



(19) 대한민국특허청(KR)
(12) 공개특허공보(A)

(11) 공개번호 10-2025-0009888
(43) 공개일자 2025년01월20일

- | | |
|---|---|
| <p>(51) 국제특허분류(Int. Cl.)
H04L 67/12 (2022.01) H04L 1/00 (2006.01)
H04L 45/745 (2022.01) H04L 69/10 (2022.01)</p> <p>(52) CPC특허분류
H04L 67/12 (2022.05)
H04L 1/0061 (2013.01)</p> <p>(21) 출원번호 10-2024-0006612</p> <p>(22) 출원일자 2024년01월16일
심사청구일자 2024년01월16일</p> <p>(30) 우선권주장
1020230089667 2023년07월11일 대한민국(KR)</p> | <p>(71) 출원인
고려대학교 산학협력단
서울특별시 성북구 안암로 145, 고려대학교 (안암동5가)</p> <p>(72) 발명자
최원미
서울특별시 동대문구 약령사로 11-1, 403호</p> <p>유연호
서울특별시 성북구 고려대로24가길 14-6, 102호
(뒷면에 계속)</p> <p>(74) 대리인
김등용</p> |
|---|---|

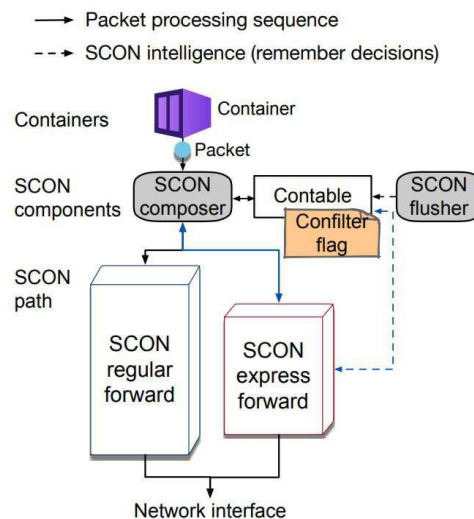
전체 청구항 수 : 총 13 항

(54) 발명의 명칭 IoT 환경에서 컨테이너의 패킷 처리 경량화 장치 및 방법

(57) 요약

IoT(Internet of Things, 사물인터넷) 환경에서 컨테이너(container)의 패킷 처리 경량화 장치 및 방법이 개시된다. 상기 패킷 처리 방법은 IoT 장치에 컨테이너 환경을 제공하는 서버에 의해 수행되고, 상기 IoT 장치에 할당된 컨테이너로부터 패킷을 수신하는 단계, 상기 패킷을 처리할 패킷 처리 경로가 결정되는 단계, 결정된 패킷 처리 경로에 따라, 상기 패킷에 대한 처리가 수행되는 단계, 및 패킷 처리가 완료된 패킷이 외부 네트워크로 전송되는 단계를 포함한다.

대표도 - 도5



(52) CPC특허분류

H04L 45/745 (2022.05)

H04L 69/10 (2022.05)

(72) 발명자

양경식

서울특별시 동대문구 무학로45길 40, 401호(용두동)

유혁

서울특별시 용산구 한강대로 205, A-803호

이 발명을 지원한 국가연구개발사업

과제고유번호	1711179534
과제번호	2023R1A2C3004145
부처명	과학기술정보통신부
과제관리(전문)기관명	한국연구재단
연구사업명	개인기초연구(과기정통부)
연구과제명	이기종 GPU 환경에서 분산 학습 자동공급 및 스케줄링 기술 연구
과제수행기관명	고려대학교
연구기간	2023.03.01 ~ 2024.02.29

이 발명을 지원한 국가연구개발사업

과제고유번호	1345375069
과제번호	2021R1A6A1A13044830
부처명	교육부
과제관리(전문)기관명	한국연구재단
연구사업명	이공학학술연구기반구축
연구과제명	융합소프트웨어연구소
과제수행기관명	고려대학교
연구기간	2023.03.01 ~ 2024.02.29

이 발명을 지원한 국가연구개발사업

과제고유번호	2710006652
과제번호	II201819
부처명	과학기술정보통신부
과제관리(전문)기관명	정보통신기획평가원
연구사업명	디지털기술선도핵심인재양성(R&D)
연구과제명	ICT명품인재양성(고려대학교)
과제수행기관명	고려대학교산학협력단
연구기간	2024.01.01 ~ 2024.12.31

명세서

청구범위

청구항 1

IoT(Internet of Things) 장치에 컨테이너(container) 환경을 제공하는 서버에 의해 수행되는 패킷 처리 방법에 있어서,

상기 IoT 장치에 할당된 컨테이너로부터 패킷을 수신하는 단계;

상기 패킷을 처리할 패킷 처리 경로가 결정되는 단계;

결정된 패킷 처리 경로에 따라, 상기 패킷에 대한 처리가 수행되는 단계; 및

패킷 처리가 완료된 패킷이 외부 네트워크로 전송되는 단계를 포함하는 패킷 처리 방법.

청구항 2

제1항에 있어서,

상기 패킷 처리 경로를 결정하는 단계는 패킷 처리 결과가 저장되어 있는 테이블을 이용하여 수행되는,

패킷 처리 방법.

청구항 3

제2항에 있어서,

상기 패킷 처리 경로를 결정하는 단계는,

상기 패킷의 연결 정보가 이전에 수신된 패킷의 연결 정보와 동일하지 않는 경우 제1 패킷 처리 경로를 상기 패킷 처리 경로로 결정하고,

상기 패킷의 연결 정보가 이전에 수신된 패킷의 연결 정보와 동일한 경우 제2 패킷 처리 경로를 상기 패킷 처리 경로로 결정하는,

패킷 처리 방법.

청구항 4

제3항에 있어서,

상기 연결 정보는 소스 IP 어드레스, 목적지 IP 어드레스, 소스 포트 어드레스, 목적지 포트 어드레스, 및 L4 프로토콜 ID를 포함하는,

패킷 처리 방법.

청구항 5

제3항에 있어서,

상기 제1 패킷 처리 경로에 따른 패킷 처리 과정은 넷필터(Netfilter), 라우팅 룩업(routing lookup), 및 헤더 처리를 포함하는,

패킷 처리 방법.

청구항 6

제3항에 있어서,

상기 제2 패킷 처리 경로에 따른 패킷 처리 과정은 상기 테이블에 저장되어 있는 라우팅 룩업 결과, TTL 값, 및 MTU 크기를 상기 패킷의 패킷 헤더에 기재하고, 체크섬 동작을 포함하는 헤더 처리를 수행하는,

패킷 처리 방법.

청구항 7

제6항에 있어서,

상기 제2 패킷 처리 경로에 따른 패킷 처리 과정은, 상기 테이블에 저장되어 있는 넷필터 수행의 결과가 폐기(discard)인 경우, 상기 패킷을 드랍하는,

패킷 처리 방법.

청구항 8

제6항에 있어서,

상기 제2 패킷 처리 경로에 따른 패킷 처리 과정은, 상기 테이블에 저장되어 있는 넷필터 수행의 결과가 폐기이고 상기 패킷에 대한 상기 패킷 헤더 처리의 결과가 통과(pass)인 경우, 상기 패킷을 네트워크 인터페이스로 전송하는,

패킷 처리 방법.

청구항 9

제6항에 있어서,

상기 제2 패킷 처리 경로에 따른 패킷 처리 과정은, 상기 패킷에 대한 상기 패킷 헤더 처리의 결과가 오류(error)인 경우, 상기 패킷을 드랍하는,

패킷 처리 방법.

청구항 10

제5항에 있어서,

상기 제1 패킷 처리 경로에 따른 패킷 처리 과정의 결과는 상기 테이블에 업데이트되는,

패킷 처리 방법.

청구항 11

제10항에 있어서,

상기 패킷 처리 방법은,

상기 컨테이너가 폐기되는 경우, 상기 테이블에 저장되어 있는 상기 컨테이너에 대응하는 정보를 삭제하는 단계를 더 포함하는,

패킷 처리 방법.

청구항 12

제10항에 있어서,

상기 패킷 처리 방법은,

상기 테이블에 저장되어 있는 정보 중 미리 정해진 시간 주기 동안 히트(hit)되지 않은 연결(connection)에 대한 정보를 삭제하는 단계를 더 포함하는,

패킷 처리 방법.

청구항 13

제9항에 있어서,

상기 테이블에 저장되어 있는 정보 중 상기 헤더 처리 결과가 오류인 연결에 대한 정보를 삭제하는 단계를 더

포함하는,
패킷 처리 방법.

발명의 설명

기술 분야

[0001] 본 발명은 IoT(Internet of Things, 사물인터넷) 장치에서 컨테이너(container)의 패킷 처리 경량화 기법에 관한 것으로, 특히 동일한 네트워크 연결 내 패킷들에 대하여 반복적인 네트워크 통신 과정에서 발생하는 불필요한 CPU 처리 병목을 해결하기 위해 패킷 전송에 필요한 정보를 재활용하여 패킷 처리 과정을 경량화하는 방법 및 장치에 관한 것이다.

배경 기술

- [0002] IoT(Internet of Things, 사물인터넷) 장치
- [0003] IoT 장치는 한정적인 컴퓨팅 자원을 보유한 장치에 센서, 소프트웨어, 통신 기술을 연결하여 인터넷에 연결하고 데이터를 수집, 저장, 처리, 및 분석할 수 있도록 하는 장치이다. 주로 스마트 팩토리, 스마트 홈, 스마트 팜과 같은 대량의 데이터가 발생하는 스마트 산업에 사용되며 데이터를 수집 및 분석을 기반으로 실시간 피드백 제어를 수행한다. 이런 제어 메커니즘은 짧은 대기 시간과 고성능 통신을 요구한다.
- [0004] 컨테이너 가상화
- [0005] 컨테이너 가상화는 하나의 호스트 운영체제를 공유하면서 독립적인 사용자 공간을 제공하여 응용 프로그램과 그에 필요한 라이브러리를 패키징하여 실행하는 방식이다. 컨테이너 기술을 통해, 관리자가 서비스를 효율적으로 관리 및 배포할 수 있으며, 하나의 물리 자원을 여러 개의 가상화된 자원으로 분리하여 전체 자원 활용률을 높일 수 있다. 이러한 장점으로 IoT 장치는 여러 센서에서 수집한 데이터를 효율적으로 관리하기 위해 컨테이너 기술을 도입한다.
- [0006] 컨테이너화된 응용 프로그램이 네트워크 통신을 위한 패킷 처리 과정
- [0007] 운영체제 위에서 동작하는 프로세스는 다른 호스트와 통신하기 위해 커널 영역에서 여러 계층의 패킷 처리과정을 거친다. 바로 운영체제 위에서 실행되는 프로세스는 시스템 콜을 통해 커널 영역에서 소켓을 생성하고, 트랜스포트 계층과 네트워크 계층(IP 계층), 데이터링크 계층을 통해 패킷 헤더를 추가한 후, 물리 인터페이스를 통해 패킷을 전송 매체로 전송한다. 한편, 컨테이너화된 프로세스는 가상화 인터페이스를 통해 호스트와 연결되어 있기 때문에 추가 호스트 계층을 거쳐서 패킷을 전송한다. 데이터링크 계층을 통과한 패킷은 호스트의 커널 영역에서 IP 계층과 데이터링크 계층을 통해 호스트 주소로 변환하거나 추가한 후, 패킷을 전송 매체로 전송한다.
- [0008] 본 명세서에서 사용되는 용어의 정의는 다음과 같다.
- [0009] 패킷: 컴퓨터 간의 통신을 위해 전송되는 데이터 블록의 단위로, 프로세스가 전송하는 데이터 이외에 데이터를 식별하기 위한 번호와 출발지/목적지 주소가 기록되며 에러 체크 비트도 포함된다.
- [0010] 네트워크 연결: 두 개의 물리 서버에서 실행하는 두 프로세스의 통신을 하나의 네트워크 연결이라고 한다. 일반적으로 연결 지향적인 TCP 프로토콜을 사용하여 이루어지며, 네트워크 연결을 통해 프로세스 간에 데이터를 주고받는다. UDP 프로토콜에서 유니캐스트 통신도 두 프로세스 간에 데이터를 주고받기 때문에 하나의 네트워크 연결을 이룰 수 있다. 본 발명에서는 TCP 패킷 헤더 정보들 중 5-튜플(5-tuple), 즉 소스 IP 어드레스(source IP address), 목적지 IP 어드레스(destination IP address), 소스 포트 어드레스(source port address), 목적지 포트 어드레스(destination port address), 및 트랜스포트 계층 프로토콜 식별자(L4 protocol identifier)이 같다면, 하나의 TCP 네트워크 연결로 간주한다.
- [0011] 운영체제: 컴퓨터 시스템에서 프로세스와 하드웨어 사이의 인터페이스 역할을 하는 소프트웨어로, 프로세스의 명령어를 하드웨어로 전달하고 필요한 자원을 제공해 준다. 여러 프로세스가 동시에 실행되고 원활하게 작동하기 위해, 운영체제는 스케줄링을 통해 CPU 자원을 할당하고, 락 또는 세마포어를 통해 메모리 및 파일 시스템 접근제어를 관리한다.

선행기술문헌

특허문헌

- [0012] (특허문헌 0001) 대한민국 등록특허 제2479438호 (2022.12.20. 공고)
(특허문헌 0002) 대한민국 공개특허 제2014-0126656호 (2014.10.31. 공개)

발명의 내용

해결하려는 과제

- [0013] 본 발명이 이루고자 하는 기술적인 과제는 IoT 장치에서 컨테이너 기술을 적용한 프로그램이 네트워크 통신을 수행하는 과정에서 발생하는 긴 패킷 처리로 인한 높은 CPU 연산을 줄여 CPU 병목을 해결하는 것이다.
- [0014] 컨테이너 기술은 독립적인 환경에서 응용프로그램이 동작할 수 있도록 지원해주는 기술이다. 그러므로, 컨테이너 내부에서 보낸 패킷은 외부로 전송하기 위해 기존 커널 네트워크 스택 외에 호스트 커널에서 추가적인 네트워크 스택을 거쳐게 된다. 그 결과, 커널 영역에서 호스트 위에 실행하는 프로세스보다 높은 CPU 연산이 요구되는데, 이는 한정적인 컴퓨팅 자원을 가진 IoT 장치에서 큰 오버헤드를 발생시킨다. 응용 프로그램이 사용가능한 CPU 자원이 줄어들고 네트워크 성능 저하의 원인이 되기 때문이다.
- [0015] 기존 연구들은 컨테이너의 네트워크 성능을 향상시키기 위해 크게 두 가지 방법을 사용하였다. 첫째 방법은 패킷 처리과정을 하드웨어 또는 사용자 공간으로 전환하여 커널 네트워크 스택의 일부를 우회하는 것이다. 이러한 방법은 패킷 처리 가속화를 지원하는 특수한 하드웨어가 필요하거나 기존 응용 프로그램의 동작을 수정해야 하는 제약이 있다. 이는, 안드로이드(Android)나 라스베리 파이(Raspberry Pi)와 같은 다양한 하드웨어 환경에서 작동하며 MQTT, CoAP와 같은 다양한 프로토콜을 사용하는 IoT 장치에는 부합하지 않는다. 두 번째 방법은 여러 코어의 CPU 자원을 동시에 사용하여 초당 패킷 처리 개수를 증가시키는 것이다. 이러한 방법은 컴퓨팅 자원이 풍부한 클라우드 환경에는 적합하지만 한정적인 컴퓨팅 자원을 가지고 확장하기 어려운 IoT 장치에는 적합하지 않다.

과제의 해결 수단

- [0016] 본 발명의 일 실시예에 따른 패킷 처리 방법은 IoT(Internet of Things, 사물인터넷) 장치에 컨테이너(container) 환경을 제공하는 서버에 의해 수행되는 패킷 처리 방법으로, 상기 IoT 장치에 할당된 컨테이너로부터 패킷을 수신하는 단계, 상기 패킷을 처리할 패킷 처리 경로가 결정되는 단계, 결정된 패킷 처리 경로에 따라, 상기 패킷에 대한 처리가 수행되는 단계, 및 패킷 처리가 완료된 패킷이 외부 네트워크로 전송되는 단계를 포함한다.

발명의 효과

- [0017] 본 발명은 IoT 장치에서 컨테이너 네트워크 통신 과정에서 발생한 CPU 병목을 해결하여 패킷 처리 지연 시간을 줄이고 네트워크 성능을 향상시킨다. 패킷 처리 과정에서, 패킷 헤더 처리에 필요한 연산을 메모리에 저장하여 재사용함으로써 호스트 영역에서 발생하는 추가적인 네트워킹 스택의 오버헤드를 해결할 수 있다. 이러한 기법은 오직 커널 레벨에서 수행되기 때문에 별도 하드웨어 지원이나 응용 프로그램의 수정이 필요하지 않는다.
- [0018] 따라서 본 발명은 반복적인 네트워크 트래픽이 발생하는 IoT 장치에서 컨테이너 기술을 활용할 때 패킷 처리 과정의 CPU 자원 사용을 줄이고 네트워크 성능을 높이는 데 효과적인 기법이다.

도면의 간단한 설명

- [0019] 본 발명의 상세한 설명에서 인용되는 도면을 보다 충분히 이해하기 위하여 각 도면의 상세한 설명이 제공된다.
- 도 1은 데이터베이스 리눅스와 컨테이너의 상세한 패킷 처리 시퀀스를 도시한다.
- 도 2는 IoT 장치의 컨테이너의 CPU 사용량, 네트워크 처리량, 및 지연을 도시한다.
- 도 3은 IoT 장치에 대한 기존 가속 기술의 부적절함을 도시한다.
- 도 4는 IoT 컨테이너의 어려움의 주요 원인 분석 결과를 도시한다.

- 도 5는 제안 기법(SCON)의 디자인을 도시한다.
- 도 6은 제안 기법의 패킷 처리 경로를 도시한다.
- 도 7은 제안 기법 테이블 구조를 도시한다.
- 도 8은 HTTP 트래픽 시나리오를 도시한다.
- 도 9는 MQTT 트래픽 시나리오를 도시한다.
- 도 10은 SoftIRQ에서의 CPU 사용량을 도시한다.
- 도 11은 네트워크 처리량을 도시한다.
- 도 12는 패킷 당 지연을 도시한다.
- 도 13은 패킷과 함수 심볼에 대한 처리 시간을 도시한다.
- 도 14는 IoT 트래픽의 99% 테일 FCT를 도시한다.
- 도 15는 SoftIRQ(UDP)에 대한 CPU 사용량을 도시한다.
- 도 16은 네트워크 처리량(UDP)을 도시한다.
- 도 17은 본 발명의 일 실시예에 따른 패킷 처리 방법을 설명하기 위한 흐름도이다.

발명을 실시하기 위한 구체적인 내용

- [0020] 본 명세서에 개시되어 있는 본 발명의 개념에 따른 실시예들에 대해서 특정한 구조적 또는 기능적 설명들은 단지 본 발명의 개념에 따른 실시예들을 설명하기 위한 목적으로 예시된 것으로서, 본 발명의 개념에 따른 실시예들은 다양한 형태로 실시될 수 있으며 본 명세서에 설명된 실시예들에 한정되지 않는다.
- [0021] 본 발명의 개념에 따른 실시예들은 다양한 변경들을 가할 수 있고 여러 가지 형태들을 가질 수 있으므로 실시예들을 도면에 예시하고 본 명세서에서 상세하게 설명하고자 한다. 그러나, 이는 본 발명의 개념에 따른 실시예들을 특정한 개시 형태들에 대해 한정하려는 것이 아니며, 본 발명의 사상 및 기술 범위에 포함되는 모든 변경, 균등물, 또는 대체물을 포함한다.
- [0022] 제1 또는 제2 등의 용어는 다양한 구성 요소들을 설명하는데 사용될 수 있지만, 상기 구성 요소들은 상기 용어들에 의해 한정되어서는 안 된다. 상기 용어들은 하나의 구성 요소를 다른 구성 요소로부터 구별하는 목적으로만, 예컨대 본 발명의 개념에 따른 권리 범위로부터 벗어나지 않은 채, 제1 구성 요소는 제2 구성 요소로 명명될 수 있고 유사하게 제2 구성 요소는 제1 구성 요소로도 명명될 수 있다.
- [0023] 어떤 구성 요소가 다른 구성 요소에 "연결되어" 있다거나 "접속되어" 있다고 언급된 때에는, 그 다른 구성 요소에 직접적으로 연결되어 있거나 또는 접속되어 있을 수도 있지만, 중간에 다른 구성 요소가 존재할 수도 있다고 이해되어야 할 것이다. 반면에, 어떤 구성 요소가 다른 구성 요소에 "직접 연결되어" 있다거나 "직접 접속되어" 있다고 언급된 때에는 중간에 다른 구성 요소가 존재하지 않는 것으로 이해되어야 할 것이다. 구성 요소들 간의 관계를 설명하는 다른 표현들, 즉 "~사이에"와 "바로 ~사이에" 또는 "~에 이웃하는"과 "~에 직접 이웃하는" 등도 마찬가지로 해석되어야 한다.
- [0024] 본 명세서에서 사용한 용어는 단지 특정한 실시예를 설명하기 위해 사용된 것으로서, 본 발명을 한정하려는 의도가 아니다. 단수의 표현은 문맥상 명백하게 다르게 뜻하지 않는 한, 복수의 표현을 포함한다. 본 명세서에서, "포함하다" 또는 "가지다" 등의 용어는 본 명세서에 기재된 특징, 숫자, 단계, 동작, 구성 요소, 부분품 또는 이들을 조합한 것이 존재함을 지정하려는 것이지, 하나 또는 그 이상의 다른 특징들이나 숫자, 단계, 동작, 구성 요소, 부분품 또는 이들을 조합한 것들의 존재 또는 부가 가능성을 미리 배제하지 않는 것으로 이해되어야 한다.
- [0025] 다르게 정의되지 않는 한, 기술적이거나 과학적인 용어를 포함해서 여기서 사용되는 모든 용어들은 본 발명이 속하는 기술 분야에서 통상의 지식을 가진 자에 의해 일반적으로 이해되는 것과 동일한 의미를 가진다. 일반적으로 사용되는 사전에 정의되어 있는 것과 같은 용어들은 관련 기술의 문맥상 가지는 의미와 일치하는 의미를 갖는 것으로 해석되어야 하며, 본 명세서에서 명백하게 정의하지 않는 한, 이상적이거나 과도하게 형식적인 의미로 해석되지 않는다.

- [0026] 이하, 본 명세서에 첨부된 도면들을 참조하여 본 발명의 실시예들을 상세히 설명한다. 그러나, 특허출원의 범위가 이러한 실시예들에 의해 제한되거나 한정되는 것은 아니다. 각 도면에 제시된 동일한 참조 부호는 동일한 부재를 나타낸다.
- [0027] 우선, IoT 장치에서의 컨테이너 네트워킹(container networking)과 패킷 처리 시퀀스(packet processing sequence)를 설명한다.
- [0028] 1. IoT 장치에서의 컨테이너 네트워킹(Container Networking in Iot Devices)
- [0029] 컨테이너는, 다른 컨테이너와의 고립을 보장하기 위해, 고유한 IP 어드레스를 갖는다. 호스트를 연결하는 네트워크 스위치 또는 라우터는 오직 호스트 머신의 IP 주소만을 인식한다. 따라서, 컨테이너에 의해 생성된 패킷을 호스트를 거쳐 전송하기 위해서, 컨테이너 플랫폼은 두 가지 기술 중 하나인 NAT(Network Address Translation) 또는 오버레이(overlay)를 활용한다.
- [0030] NAT(Network Address Translation). 각 호스트는 컨테이너 어드레스를 호스트 어드레스로 매핑하는 NAT 테이블 관리한다. 매핑 결과에 기초하여, NAT는 컨테이너 IP를 호스트 IP로 대체함으로써, 패킷 헤더를 수정한다. 따라서, 외부 네트워크를 통과하는 패킷은 호스트 어드레스를 가지고 있고, 스위치 및/또는 라우터는 이를 인식하고 호스트 간에 올바르게 전달할 수 있다.
- [0031] 오버레이(Overlay). 오버레이는 추가적인 헤더를 활용하는 기법이다. 소스 호스트(source host)로부터의 컨테이너 패킷은 추가적인 헤더(예, VXLAN 또는 IP/IP)로 캡슐화되고(encapsulated), 목적지 호스트(destination host)에서 디캡슐화된다(decapsulated). 외부 네트워크는 추가적인 헤더 내의 호스트 어드레스에 기초하여 패킷을 전달한다. 오버레이가 컨테이너 패킷을 성공적으로 호스트 간에 전달하지만, NAT에 비해, CPU 사이클과 지연 측면에서 더 큰 오버헤드를 발생시키는 것으로 알려져 있다. 오버레이는 패킷 캡슐화/디캡슐화 프로세스를 위한 오버헤드를 필요로 하기 때문이다. 또한, 패킷 사이즈가 커지면, 캡슐화된 패킷 크기가 최대 전송 단위를 초과할 때 패킷이 단편화될(fragmented) 수 있다. 이는 IoT 장치에게는 간단한 동작이 아니다. 따라서, 제한된 컴퓨팅 리소스를 가진 IoT 장치에게는, NAT가 선호되는 선택이다.
- [0032] Docker, LXC, 및 OpenVZ와 같은 다양한 컨테이너 플랫폼은 브릿지 네트워크(bridge network)를 활용한다. Open vSwitches와 같이, 브릿지의 대안을 채용하는 일부 다른 기법들이 존재하지만, 그 역할은 브릿지와 동일하다. 브릿지 네트워크에서 NAT 연산(NAT operation)은 IoT 장치와의 컨테이너 네트워킹에서 사실상 표준이기 때문에, 브릿지를 갖는 NAT에 집중한다.
- [0033] 2. 패킷 처리 시퀀스(Packet Processing Sequence)
- [0034] 도 1은 네이티브 리눅스와 컨테이너의 상세한 패킷 처리 시퀀스를 도시한다. TCP(Transmission Control Protocol) 패킷 처리를 예로 설명한다. TCP 패킷과 UDP(User Datagram Protocol) 패킷에 대한 처리 시퀀스는, 트랜스포트 계층 함수(transport layer functions)를 제외하고, 대체로 유사하다. 트랜스포트 계층의 경우, TCP가 상대적으로 UDP보다 더 복잡하다.
- [0035] 우선, 도 1a에서 네이티브 리눅스의 함수를 살펴본다. 사용자 어플리케이션(user application)으로부터의 패킷은 리눅스 네트워킹 스택(Linux networking stack)의 트랜스포트 계층의 첫번째 함수인 sock_sendmsg를 통과한다. 그런 다음 tcp_sendmsg는 패킷의 메타데이터와 페이로드의 데이터 구조(예: sk_buff)에 대한 메모리 공간을 할당하고, 원격의 목적지 호스트와의 연결을 수립(establishes)한다. 연결이 수립된 후, tcp_sendmsg는 IP 계층(ip_queue_xmit)을 호출한다. IP 계층 함수는 라우팅 룩업(routing lookup)과 넷필터 처리(Netfilter processing)를 수행한다. 다음 홉(next-hop)의 MAC 어드레스가 식별되면, 패킷은 링크 계층(dev_queue_xmit)으로 전송되고, 더 나아가 네트워크 인터페이스 계층(ndo_start_xmit)으로 전송된다. 마지막으로, 패킷은 외부 네트워크로 전송된다.
- [0036] 다음으로, 도 1b에서, 컨테이너에서의 패킷 처리 과정을 살펴본다. TCP 및 IP 계층에서의 함수는 네이티브 리눅스와 동일하다. 그러나, 컨테이너의 링크 계층에서는 dev_queue_xmit이 실제 호스트의 네트워크 장치 대신 veth로 패킷을 전송하는 veth_xmit을 호출한다. veth는 각 컨테이너에 기본으로 할당된 가상 네트워크 인터페이스(virtual network interface)로, 격리된 네트워크 주소 공간을 제공한다. 각 veth는 고유한 MAC 및 IP 어드레스를 가지기 때문에 컨테이너를 구별할 수 있다.
- [0037] veth를 통과하는 패킷은 브릿지로 전달된다. 브릿지는, 전달된 패킷에 대해 NAT를 수행하는 호스트 측 네트워크 인터페이스로 패킷을 전달하기 위하여, 컨테이너의 veth 인터페이스의 MAC 주소를 식별한다. 브릿지의 상세 동

작은 다음과 같다. 먼저, br_handle_frame에서 패킷의 MAC 주소를 식별하기 위하여 ARP 테이블 룩업(table lookup)을 수행한다. 그 다음, br_handle_frame은 패킷이 어떤 네트워크 인터페이스를 통해 전송될지를 결정하기 위하여 넷필터(Netfilter) 작업을 수행한다. 마지막으로, br_handle_frame은 패킷을 다른 IP 계층 처리(IP layer processing)로 이끄는 대응되는 네트워크 인터페이스로 전송한다. 이는 두 번째 IP 계층이라고 명명된다.

[0038] 두 번째 IP 계층에서, ip_rcv, ip_forward, 및 ip_output이 차례대로 호출된다. 이 호출 체인은, 패킷의 적절한 목적지 IP를 결정하는 라우팅 룩업(routing lookups)과 패킷을 전달할지 여부를 결정하는 넷필터(Netfilter) 동작을 포함한다. 특히, NAT는 패킷의 소스 IP 어드레스를 컨테이너의 IP 어드레스에서 호스트의 IP 어드레스로 변경하는 IP 계층의 ip_output의 Netfilter hook에 의해 수행된다. 두 번째 IP 계층에서의 NAT 이후에, 패킷이 호스트 서버의 실제 네트워크 인터페이스 전달되고(dev_queue_xmit), 마침내 패킷이 전송된다(ndo_start_xmit).

[0039] 패킷 처리 시퀀스의 분석을 통해, 도 1b에 도시된 컨테이너의 네트워킹 프로세싱 시퀀스는 도 1a의 네이티브 리눅스보다 훨씬 길다는 것을 알 수 있다.

[0040] 이하에서는, IoT 컨테이너의 어려움과 주된 원인 분석을 설명한다.

[0041] 먼저 IoT 컨테이너의 어려움은, 1) IoT 장치에서의 비효율적인 CPU 사용, 2) IoT 장치에서의 부족한 처리량과 지연, 및 3) 기존 가속화 기술의 부적절함에 있다. 그런 다음, 어려움의 근본 원인을 심층적으로 분석하고 새로운 커널 레벨 가속기(kernel-level accelerator)에 대한 접근 방식의 개요를 설명한다.

[0042] 1. 어려움(Challenges)

[0043] 실험 설정(Experiment setting). 다양한 실험을 통해 IoT 컨테이너의 어려움을 보인다. 실험에는 널리 사용되는 IoT 장치인 라즈베리 파이 4(Raspberry Pi 4)를 사용한다. 그러나, 본 발명의 권리범위가 IoT 장치의 종류에 한정되는 것은 아니며, 실시예에 따라 다양한 IoT 장치가 이용될 수 있다. 라즈베리 파이 4는 ARM Cortex-A72 64 비트 쿼드 코어@1.5GHz CPU, 8GB RAM, 그리고 1000 Mbps 이더넷 칩을 구비한다. Linux 버전 5.10을 실행하며 Docker 런타임 버전 20.10에 의해 컨테이너를 생성한다.

[0044] 라즈베리 파이는 1000 Mbps 이더넷으로 엣지 클라우드 서버(서버 머신)에 연결된다. 서버 머신은 64GB 메모리를 갖는 Intel i7-3770K 옥타 코어@3.5GHz CPU와 256GB SSD가 장착되어 있다. 이 서버는 Linux 버전 5.4를 실행한다. iperf를 사용하여 라즈베리 파이의 컨테이너로부터 서버 머신으로의 TCP 트래픽을 생성한다. iperf는 IoT 시스템 평가에서 가장 인기 있는 네트워크 벤치마크 중 하나이다. 특별히 언급되지 않는 한, 이 명세서 전체에서는 여기에서 설명한 실험 설정을 사용한다.

[0045] 비교를 위해, 동일한 IoT 장치에서 동일한 설정(예: iperf)을 사용하여 "네이티브"에 대한 실험도 진행하였다. 컨테이너에 대해 하나의 CPU 코어를 사용하여 실험 결과를 제시하는데, 이는 자원이 제한된 IoT 환경의 전형적인 사양 중 하나이다. 또한, 더 많은 코어(최대 4개의 코어)를 사용한 실험도 진행하였고, 이는 아래의 결과와 유사한 경향을 보여준다.

[0046] Challenge 1: Inefficient CPU usage in IoT device. 먼저, IoT 장치에서의 CPU 사용량(usage)을 평가한다(도 2a). TCP 패킷의 크기를 32 B에서 1 KB까지 증가시킨다(x축). 컨테이너와 네이티브 프로세스 간의 측정 결과를 비교한다.

[0047] 도 2a에서, IoT의 네이티브와 컨테이너 모두 CPU를 완전히 활용하지만, 두 경우에서의 세부적인 사용량은 상이하다. 특히, SoftIRQ 처리에 대한 컨테이너의 CPU 사용량이 네이티브에 비해 상당히 높다. 예를 들어, 1 KB 메시지를 처리할 때, 컨테이너 네트워킹은, SoftIRQ 처리에서, 네이티브에 비해 최대 2.6배 더 높은 CPU 사용량을 발생시킨다. 이러한 결과는, 시스템 호출을 처리하는 시스템 레벨과 패킷을 생성하는 사용자 레벨에서 처리에 사용 가능한 CPU 자원이 줄어들어 네트워크 처리량에 부정적인 영향을 미침을 의미한다. 이러한 비효율적인 CPU 사용은 많은 IoT 장치에 대한 중요한 도전 과제로 작용한다. 특히, 많은 IoT 장치가 배터리 구동이기 때문에 CPU 사용에 매우 민감하게 반응하는 특성이 있다.

[0048] Challenge 2: Poor throughput and latency in IoT devices. 다음으로, IoT 장치의 네트워크 처리량과 지연을 평가한다. 도 2b에서는 메시지 크기에 따른 네트워크 처리량 증가를 보여준다. 특히, 1 KB 메시지의 경우, 컨테이너의 네트워크 처리량은 네이티브에 비해 59% 감소합니다. 평균적으로, 컨테이너는 네이티브보다 44% 낮은 처리량을 가진다.

[0049] Netperf에 의해 측정된 평균 및 99% 테일 값(tail values) 관점에서, 컨테이너의 패킷 전송 지연 시간도 평가한다. 결과에서는 평균과 99% 테일 지연 값이 모두 증가하는 것을 보여준다. 특히, 도 2c와 도 2d에서, 컨테이너

는 평균 지연 시간을 평균 2배 증가시키고 99% 테일 지연 시간도 평균 2배 증가시키고 있다. IoT 장치가 실시간 통신에 자주 사용됨을 고려할 때(예: 스마트 홈 자동화, 원격 건강 모니터링, 및 공장 제어 등), 네트워크 처리량을 향상시키고 지연 시간을 줄이는 것이 매우 중요하다.

- [0050] Challenge 3: Inadequacy of existing acceleration techniques. 컨테이너는 데이터센터 관점에서 상당한 관심을 받고 있기 때문에, 다양한 네트워킹 가속화 연구가 수행되고 있다. 그러나 다음과 같은 이유로 인해 이러한 연구들은 IoT 장치에는 부적절하다.
- [0051] 첫째, 기존 연구들은 이종 IoT 장치(heterogeneous IoT devices)를 위한 상호 운용성(interoperability)이 부족하다. 네트워킹 가속화를 위해, 몇몇 연구는 패킷 처리에서 커널 의미론(kernel semantics)을 변경했는데(예: 커널 바이패스(kernel bypassing) 및 사용자 레벨 처리(user-level processing)), 모든 기존 응용 프로그램의 소켓 API를 수정하여야 한다. 이는, 이러한 기술을 다양한 이종 IoT 응용 프로그램에 적용하기 위해 상당한 개발 노력이 필요하다는 것을 의미한다.
- [0052] 다른 연구들은 네트워크 프로세싱을 오프로드하기 위해 추가 하드웨어(예: RDMA NIC 또는 DPDK NIC)를 사용하도록 제안하였다. 그러나, 이러한 NICs가 IoT 장치에서 쉽게 사용 가능한지 여부가 명확하지 않다. 따라서, 이러한 기술은 네트워킹 성능을 향상시킬 수 있지만, 필요한 하드웨어 지원이 부족한 이종 IoT 장치 간의 상호 운용성을 제공하지 못할 수 있다.
- [0053] 둘째, 이러한 연구들은 IoT 트래픽을 충분히 가속화하지 못한다. 네트워킹 스택의 일부를 우회하면서 특별한 하드웨어가 필요하지 않는 최신이자 대표적인 방법인 오픈 소스 가속화 기술인 Slim의 성능을 실험하였다. IoT 장치에서는 Slim을 iperf를 활용하여 성능을 평가하는데, Slim과 호환되는 유일한 벤치마크로 확인되었기 때문이다. Slim에는 기존 Linux 응용 프로그램을 실행하기 위한 사용자 지정 라이브러리가 포함되어 있다. 그러나, 패킷 처리 의미론의 변경(호스트 우회)으로 인해 이 라이브러리는 전적으로 기존 시스템 호출(예: accept)을 지원하지 않는다. Slim에서 MQTT 라이브러리와 같은 IoT 트래픽 응용 프로그램을 실행하려고 했지만 수정이 필요하였다. 따라서 여기에서는 iperf 결과만 제시한다.
- [0054] 네이티브, Slim, 및 컨테이너 간의 처리량을 비교한다. 트래픽 크기를 두 가지 범주, 즉 1) 보편적인 클라우드 트래픽(512 B 및 1 KB)과 2) IoT 트래픽(32 B에서 256 B)으로 구분한다. 클라우드 트래픽(도 3a)의 경우, Slim은 평균적으로 컨테이너 대비 네트워크 처리량을 66% 향상시킨다. 네이티브와 비교했을 때, Slim은 평균적으로 14% 더 낮은 처리량을 보여준다. 그러나, IoT 트래픽(도 3b)의 경우, Slim은 네이티브와 비교했을 때 처리량이 떨어져 평균 59% 감소한다. 컨테이너와 비교했을 때도 Slim은 평균적으로 34% 낮은 성능을 보여준다. 이 결과는 기존 기술이 대규모이고 오래 지속되는 클라우드 트래픽을 대상으로 설계되었으며, 따라서 IoT 트래픽의 처리량을 향상시키는 데 부족하다는 것을 나타낸다. 이는 컨테이너 네트워킹에서 IoT 트래픽을 위한 새로운 가속기가 필요하다는 점을 강조한다.
- [0055] 2. Root Cause Analysis and Our Approach
- [0056] IoT 장치와 트래픽을 위한 새로운 가속기를 설계하기 위해, 비효율적이고 낮은 CPU 사용량, 네트워크 처리량, 및 지연의 근본 원인을 조사한다. ftrace를 사용하여 IoT 컨테이너 네트워킹 스택을 분석한다. 구체적으로는, 64 B TCP 패킷을 보낼 때, IoT 컨테이너의 네트워킹 스택에서 각 함수 심볼을 처리하는 데 걸리는 시간을 측정한다.
- [0057] 도 4a는, 앞서 논의한 것과 같이, 컨테이너 네트워킹에서 패킷 처리에 관여하는 주요 함수들에 대한 측정 결과를 도시한다. 결과는, 컨테이너에서 단일 TCP 패킷을 처리하는 데 필요한 시간이 네이티브보다 두 배 높다는 것을 나타낸다. 이는, 컨테이너 네트워킹 스택이 "추가" 처리를 필요로 하는데, 예를 들어 브릿지 계층(br_handle_frame), 두 번째 IP 계층(ip_rcv), 및 veth/eth(dev_queue_xmit 및 veth_xmit)와 같은 부분이 포함되어 있기 때문이다. 컨테이너 네트워킹 스택의 다른 부분인 TCP 및 IP 계층은 네이티브 환경과 유사한 양의 CPU 시간을 소비한다.
- [0058] 결과는, 추가 처리에서 IP 계층이 가장 큰 부분(63%)을 차지한다는 것을 보여준다. 따라서, IP 계층에서의 패킷 처리에 대해 추가 분석을 수행한다. 각 심볼에 사용된 CPU 사이클을 측정하고, 넷필터(Netfilter), 헤더 처리(header processing), 라우팅 룩업(routing lookup), 및 패킷 검증(packet validation)의 작업 범주로 분류한다. 각 범주는 다음과 같은 동작을 의미한다.
- [0059] · 넷필터(Netfilter): 리눅스 커널에 의해 제공하는 패킷 필터링을 의미한다. 넷필터(Netfilter)는 외부 네트워킹을 가능하게 하기 위해 컨테이너의 패킷 소스 IP 어드레스를 호스트 서버로 변경하는 NAT 처리도 포함한다.

실험에서는 기본 구성을 사용한다.

- [0060] · 헤더 처리(Header processing): TTL(Time-to-live) 필드와 체크섬 값들(checksum values)을 다시 계산하고, NAT를 위해 헤더의 크기를 확장하기 위해 패킷 헤더를 재할당한다.
- [0061] · 라우팅 룩업(Routing lookup): 전송된 패킷에 대한 대응되는 네트워크 인터페이스를 결정한다.
- [0062] · 패킷 검증(Packet validation): 리눅스 브릿지에서 패킷이 로컬인지 원격인지 체크한다.
- [0063] 도 4b를 참조하면, 넷필터(Netfilter), 헤더 처리(header processing), 및 라우팅 룩업(routing lookup)의 합이 추가 패킷 처리에서 CPU 사이클의 94%를 차지함을 알 수 있다. 심층 분석 결과는, 상술한 세 가지 동작이 많은 CPU 사이클을 소비하는 이유가 다음과 같음을 의미한다. 첫째, 넷필터(Netfilter)는 컨테이너 간의 외부 네트워킹을 위해 NAT 테이블 룩업(NAT table lookup)을 수행한다. 또한, 넷필터(Netfilter)는 방화벽 또는 보안을 위한 정책을 구현하며, 이는 정책에 따른 IP 테이블 룩업(iptables lookup)을 포함한다. 넷필터(Netfilter)가 비용이 많이 드는 스핀락 동작(spin-lock operations)이 필요하여 높은 CPU 오버헤드를 발생시킨다는 것을 발견하였다. 더구나, 추가 IP 헤더 처리는 두 번째 IP 계층 처리를 위해 수행된다. 또한, 라우팅은 모든 수신된 패킷에 대하여 적절한 목적지를 찾기 위해 데이터 구조(trie)를 조회한다.
- [0064] 넷필터(Netfilter), 헤더 처리, 및 라우팅 룩업(routing lookup)은, 컨테이너가 동일한 출발지 및 목적지 쌍을 갖는 패킷을 전송하더라도, 반복적으로 수행된다. 따라서, IoT 장치가 트래픽을 버스티하게(bursty) 전송할 때, CPU를 상당히 소모하게 되며, 처리량 및 지연에서의 저하로 이어진다.
- [0065] 따라서, 본 발명에서는 1) 넷필터(Netfilter), 2) 헤더 처리, 및 3) 라우팅 룩업(routing lookup)과 같은 세 가지 높은 오버헤드 동작에 대한 커널 가속기(kernel accelerator)를 설계한다. 제안 기법(SCON으로 명명될 수 있음)의 목표는 다음과 같다.
- [0066] · IoT 네트워크 트래픽의 네트워킹 처리량과 지연을 향상시키면서 반복적인 리소스 소모를 감소시킨다.
- [0067] · IoT 장치를 위한 어떠한 하드웨어 오프로딩 지원 없이 네트워킹 가속화 기술을 디자인한다.
- [0068] · 애플리케이션 재구현(application re-implementation)을 초래하는 커널 의미론 변경이나 API 변경을 피한다.
- [0069] 이하에서는, IoT 장치에서 효율적인 CPU 사용량과 높은 네트워크 성능을 제공하는 제안 기법(SCON)의 세부 디자인을 제시한다. 도 5는 제안 기법(SCON)의 개략적인 디자인을 도시한다. 앞서 관찰된 높은 오버헤드 동작을 제거하기 위해, 제1 패킷 처리 과정(도 5의 SCON regular forward, 제1 패킷 처리 루틴 등으로 명명될 수도 있음)와 제2 패킷 처리 과정(도 5의 SCON express forward, 제2 패킷 처리 루틴 등으로 명명될 수도 있음)로 구성되는 패킷 처리 경로(도 5의 SCON path)를 제안한다. 이들은 넷필터(Netfilter), 헤더 처리, 및 라우팅 룩업(routing lookup) 동작에서의 반복을 피하기 위해, 컨테이너 패킷 처리에 "지능(intelligence)"을 추가한다. 제안 기법(SCON)의 지능은, 최근 연결에 대한 제1 패킷 처리 과정(SCON regular forward)에서의 패킷 처리 결과(예: 라우팅 테이블 룩업 결과(routing table lookup results))를 기억하는 것이다.
- [0070] 구체적으로, 컨테이너로부터의 패킷이 리눅스 브리지로 전달될 때, 제안 기법(SCON)은 제1 패킷 처리 과정(SCON regular forward)과 제2 패킷 처리 과정(SCON express forward) 중에서 어떤 과정이 패킷을 처리할 적절한 루틴인지를 구별(또는 판단)한다. 패킷이 출발지와 목적지 컨테이너 간에 새롭게 생성된 연결에 속한다면(출발지 및 목적지 IP 및 포트 어드레스, 및 L4 프로토콜 식별자를 포함한 5-튜플(5-tuple) 정보에 의해 식별될 수 있음), 제1 패킷 처리 경로(SCON regular forward)로 이동한다. 패킷이, 제안 기법(SCON)이 최근에 이미 본 네트워크 연결에 속한다면, 제2 패킷 처리 경로(SCON express forward)로 이동한다.
- [0071] 그러나, 하드웨어 오프로딩과 API 수정 없이 모든 들어오는 패킷의 5-튜플 정보를 식별하는 것은, 리눅스 브리지에 상당한 오버헤드를 초래할 수 있다. 리눅스 브리지는 여러 컨테이너로부터의 패킷을 동시에 처리하기 때문이다. 패킷 식별 비용을 감소시키기 위해 다음과 같은 제안 기법(SCON)의 구성 요소, 즉 1) 패킷 처리를 위한 컨테이너당 구조(per-container structures)로써의 테이블(Contable이나 패킷 처리 결과 저장 테이블 등으로 명명될 수 있음) 및 플래그(Confilter flag로 명명될 수 있음), 그리고 2) 테이블(Contable)을 할당하고 관리하기 위한 테이블 생성부(도 5의 SCON composer) 및 테이블 관리부(도 5의 SCON flusher)를 제안한다. 이하에서는, 패킷 처리 경로(SCON path) 및 제안 기법(SCON)의 구성 요소를 하나씩 설명한다.
- [0072] 1. 패킷 처리 경로(SCON Path)
- [0073] 먼저, 패킷 처리 경로(SCON path)의 제1 패킷 처리 경로(SCON regular forward)를 설명한다. 여기서, 패킷은

오리지널 리눅스 네트워킹 스택의 호출 체인(call chain)을 따른다(도 6 참조). 호출 체인(call chain)을 통과하면서, 제안 기법(SCON)은 패킷을 감지하고, 다음과 같이, 넷필터(Netfilter), 라우팅 룩업(routing lookup), 및 헤더 처리로부터의 패킷 처리 결과를 수집한다.

- [0074] · 넷필터(Netfilter): NF_INET_PRE_ROUTING, NF_INET_FORWARD, 및 NF_INET_POST_ROUTING의 결과.
- [0075] · 라우팅 조회(routing lookup): 다음 홉의 IP 및 MAC 어드레스.
- [0076] · 헤더 처리: MTU 사이즈, TTL 값, 및 IP 계층(L3) 프로토콜 ID.
- [0077] 패킷이 콜 체인(call chain)을 통과한 후에, 제안 기법(SCON)은 Contable이라 명명되는 저장소에 수집된 패킷 처리 결과를 "기억(memorizes)"하며, 해당 패킷이 속한 네트워크 연결의 식별자(5-tuple 값)도 함께 저장한다.
- [0078] 다음으로, 제2 패킷 처리 경로(SCON express forward)는, 이전의 제1 패킷 처리 경로(SCON regular forward)에서 구성된 테이블(Contable)을 활용하여 반복적인 패킷 처리를 피할 수 있다. 제2 패킷 처리 경로(SCON express forward)는 두 부분, 즉 1) scon_express_forward 및 2) SCON_EXPRESS_NETFILTER로 구성된다. 오리지널과 비교하면, scon_express_forward는 패킷 헤더 처리 및 라우팅 룩업(routing lookup)에 대한 가벼운 대안(lightweight alternatives)을 제공하며, SCON_EXPRESS_NETFILTER는 넷필터(Netfilter) 처리에 대한 부분이다.
- [0079] scon_express_forward는 테이블(Contable)에 저장된 결과를 검색하고, 패킷에 필요한 헤더 정보(예: 라우팅 룩업(routing lookup) 결과, TTL 값, 및 MTU 크기)를 채운다. 또한, 필수적인 체크섬 동작(checksum operation)을 수행하고 다음 SCON_EXPRESS_NETFILTER로 전송한다.
- [0080] SCON_EXPRESS_NETFILTER에서, 제안 기법(SCON)은, 테이블(Contable)에 저장된 넷필터(Netfilter) 결정을 기억함으로써, 넷필터(Netfilter) 처리의 오버헤드(예: 구조 조회(structure lookup)를 위한 반복적인 스핀락(spin-locks))을 감소시킨다. 넷필터(Netfilter)로부터의 결정은 두 가지 동작, 즉 1) 성공(success) 및 2) 폐기(discard)이다. 성공(success)은 패킷이 즉시 나갈 준비가 되었음을 의미하며(출발지 NAT 처리 포함), 따라서 SCON_EXPRESS_NETFILTER는 패킷을 네트워크 인터페이스로 전송한다. 폐기(discard)는 시스템 관리자에 의해 설정된 보안 위반(security breach)이나 방화벽 정책(firewall polices)으로 인해 패킷이 드랍(drop)되어야 함을 의미한다.
- [0081] 패킷 처리 경로(SCON path)가 의도한 대로 작동하려면, 테이블(Contable) 구조체가, 연결 정보를 지능적으로 저장하기 위해, 적절한 시간에 할당되고 생성되어야 한다. 또한, 연결이 종료되거나 특성(characteristics)이 변경될 때(예: 연결 재설정 등), 메모리에 저장된 연결 정보를 업데이트하는 것이 필요하다. 이러한 역할을 제안 기법(SCON)의 구성 요소가 수행하며, 이하에서 상세히 설명한다.
- [0082] 2. 제안 기법의 구성 요소(SCON Components)
- [0083] 이하에서는, 제안 기법(SCON)의 4가지 구성, 즉 1) 테이블(Contable), 2) 플래그(Confilter flag), 3) 테이블 생성부(SCON composer), 및 4) 테이블 관리부(SCON flusher)를 설명한다.
- [0084] 2.1 테이블(Contable). 테이블(Contable) 구조는 도 7에 도시되어 있다. 테이블(Contable)은 연결의 패킷이 제1 패킷 처리 경로(SCON regular forward)에 의해 어떻게 처리되었는지를 기억한다. 또한, 각 패킷이 어떤 패킷 처리 경로(SCON 루틴)를 따라야 하는지 식별한다. 제안 기법(SCON)은 각 컨테이너 당 별도의 테이블(Contable)을 유지하며, 이는 연결된 목록(linked list)으로 연결된 여러 서브 테이블(도 7의 Conentry) 구조로 구성된다.
- [0085] 각 연결에 대해 서브 테이블(Conentry)이 구축되고 사용된다. 제안 기법(SCON)은 5-튜플(5-tuple)을 통해 서브 테이블(Conentry)과 연결을 모두 식별한다. 각 서브 테이블(Conentry)은 1) 라우팅 정보(즉, IP 라우팅, MAC 라우팅, L3 프로토콜 ID) 및/또는 2) 플래그(Confilter flag)를 유지한다. IP 라우팅은 패킷의 목적 네트워크 인터페이스(즉, struct net_device), TTL 값, 및 제1 패킷 처리 경로(SCON regular forward)로부터 수집한 해당 인터페이스의 MTU 크기를 저장한다. MAC 라우팅은 호스트 서버로부터 전송된 패킷을 수신하는 다음 홉 서버의 링크 계층 정보를 저장한다. L3 프로토콜 ID는 IP 헤더에 필요한 IP 프로토콜 번호이다. 플래그(Confilter flag)는 넷필터(Netfilter) 동작을 집계하며(aggregates), 패킷 헤더 처리의 일부로서 이하에서 자세히 설명한다.
- [0086] 2.2 플래그(Confilter flag). 테이블(Contable)의 플래그(Confilter flag)에는, 1) 넷필터(Netfilter)로부터의 처리 결과, 및/또는 2) 헤더 처리가 포함된다. 다시 말하면, 테이블(Contable)은 넷필터 처리 결과 및/또는 헤더 처리 결과에 대한 정보를 포함할 수 있다. 넷필터(Netfilter)는 사용자 또는 시스템 관리자가 악성 또는 필

터링 가능한 네트워크 연결에 대한 규칙을 정의하는 역할을 한다. 또한, 정상적인 네트워크 연결의 경우, 넷필터(Netfilter)는 외부 통신을 위해 출발지 NAT(source NAT)를 수행한다. 넷필터(Netfilter)로부터의 결과는 1) 성공(success) 또는 2) 폐기(discard)로, 앞서 설명한 바와 같다.

[0087] 헤더 처리 동안, 패킷 비트 내의 오류를 감지 하기 위해, 패킷 헤더 체크섬(packet header checksum)이 계산된다. 이러한 오류는 패킷당 존재하며 해당 패킷이 속한 전체 연결에 지속되지 않는다. 헤더 처리의 결과는 1) 통과(정상, normal) 또는 2) 오류(폐기, dropped)로 분류될 수 있다.

[0088] 패킷(연결)이 넷필터(Netfilter)로부터 "성공(success)"이고 헤더 처리로부터 "통과(pass)"인 경우, 제안 기법(SCON)은 플래그(Confilter flag)를 "성공(success)"으로 설정한다. 이 경우, 제1 패킷 처리 경로(SCON regular forward)는 패킷을 바로 네트워크 인터페이스로 전송한다. 만약 패킷이 넷필터(Netfilter)에서 폐기된(discarded) 경우(헤더 처리 결과에 관계없이), 제안 기법(SCON)은 플래그(Confilter flag)를 "폐기(discard)"로 설정하고 제1 패킷 처리 경로(SCON regular forward)는 패킷을 드랍(drop)한다. 마지막으로, 패킷이 헤더 처리로부터 "오류"로 간주되는 경우, 다음에 오는 패킷은 제1 패킷 처리 경로(SCON regular forward)를 통과하여 테이블(Contable)의 정보가 업데이트될 수 있다. 성공(success), 폐기(discard), 및 제1 패킷 처리 경로(SCON regular forward)로 이동을 의미하는 플래그(Confilter flag)는 정수로 정의될 수 있다(즉, 1, -1 및 0).

[0089] 2.3 테이블 생성부(SCON composer). 테이블 생성부(SCON composer)는 테이블 구조체(Contable structure)를 구성하는 역할을 수행한다. 컨테이너에서 패킷이 전송되면, 테이블 생성부(SCON composer)는, 패킷의 5-튜플(5-tuple)과 일치하는 서브 테이블(Conentry)이 있는지 확인하기 위하여, 테이블(Contable)을 조회한다. 일치하는 서브 테이블(Conentry)이 없으면, 패킷이 이전에 본 적이 없는 새로운 네트워크 연결에 속한다는 것을 나타낸다. 따라서, 테이블 생성부(SCON composer)는 새로운 서브 테이블(Conentry)을 할당하고, 테이블(Contable)에 추가하며, 이전 서브 테이블(Conentry)에 연결하고, 해당 엔트리(entries)를 초기화한다. 패킷이 제안 기법(SCON)의 정규 경로를 통과한 후, 테이블 생성부(SCON composer)가 기억된 패킷의 처리 정보로 서브 테이블(Conentry)을 업데이트한다.

[0090] 2.4 테이블 관리부(SCON flusher). 테이블 관리부(SCON flusher)는 더 이상 유효하지 않은 테이블(Contable) 또는 서브 테이블(Conentry) 구조체를 제거하는 역할을 수행한다. 이러한 구조체가 더 이상 유효하지 않은 상황은 여러 가지가 있다. 첫 번째 상황은 해당 컨테이너가 중지되거나(stopped) 제거된(removed) 경우로, 컨테이너에 할당된 테이블(Contable)이 더 이상 사용되지 않게 된다. 이러한 경우를 위해, 각 테이블(Contable)이 컨테이너의 인터페이스(즉, 브리지 포트)에 의존하도록 설계할 수 있다. 컨테이너가 중지되거나 제거될 때, 커널(kernel)에 의해 컨테이너의 인터페이스가 제거되기 때문이다. 따라서, 컨테이너가 중지 또는 제거되면, 테이블(Contable)도 함께 제거된다.

[0091] 두 번째 상황은 컨테이너가 연결에서 패킷을 더 이상 전송하지 않을 때 발생한다. 이를 감지하기 위해, 타이머가 주기적으로 서브 테이블(Conentry)의 히트(hits)의 빈도(frequency)를 체크한다. 타이머 간격(예: 10초) 내에 서브 테이블(Conentry)이 히트되지 않으면, 테이블 관리부(SCON flusher)가 활성화되어 서브 테이블(Conentry)을 해제한다. 즉, 미리 정해진 주기 동안 활용되지 않은(조회되지 않거나 히트되지 않은의 의미로 이해될 수 있음) 테이블 정보(즉, 서브 테이블)은 삭제될 수 있다.

[0092] 테이블 관리부(SCON flusher)는, 또한, 기존 연결이 새롭게 갱신되는 상황을 처리하기 위해 설계될 수 있다. TCP 연결이 닫히고(closed) 다시 시작될(refreshed) 때, 서브 테이블(Conentry)은 플러시되고(flushed) 그에 따라 업데이트되어야 한다. 이 상황을 감지하기 위해, TCP 패킷의 SYN 플래그가 모니터링된다. 이 모니터링은 특히 MQTT/HTTP와 같은 에너지 효율을 위해 짧은 수명의 메시지를 빈번하게 사용하는 IoT 디바이스에게 중요하다.

[0093] 상술한 테이블 생성부나 테이블 관리부 등은 기능 및 논리적으로 분리될 수 있음으로 나타내는 것이며, 반드시 각각의 구성이 별도의 물리적 장치로 구분되거나 별도의 코드로 작성됨을 의미하는 것이 아님을 본 발명의 기술 분야의 평균적 전문가가 용이하게 추론할 수 있을 것이다.

[0094] 또한, 본 명세서에서 "~부"라 함은, 본 발명의 기술적 사상을 수행하기 위한 하드웨어 및 상기 하드웨어를 구동하기 위한 소프트웨어의 기능적, 구조적 결합을 의미할 수 있다. 예컨대, 상기 "~부"는 소정의 코드와 상기 소정의 코드가 수행되기 위한 하드웨어 리소스의 논리적인 단위를 의미할 수 있으며, 반드시 물리적으로 연결된 코드를 의미하거나, 한 종류의 하드웨어를 의미하는 것이 아니다.

- [0095] 구현 및 평가(IMPLEMENTATION AND EVALUATION)
- [0096] 제안 기법(SCON)의 모든 구성 요소를 Debian GNU/Linux 11 (bullseye)에서 Linux 커널 버전 5.10을 사용하여 구현한다. 구현은 2K 라인의 C 코드를 포함한다. 추가로, 제안 기법(SCON)을 비활성화하는 구성 옵션(configuration option)도 함께 제공할 수 있다. 실험 설정은 앞선 기재와 동일하다.
- [0097] 두 가지 종류의 실험, 즉 1) 실제 응용 프로그램(real-world applications) 및 2) 마이크로 벤치마크(micro-benchmarks)를 수행한다. 실제 응용 프로그램으로는 제안 기법(SCON)의 효과를 1) HTTP 트래픽 시나리오 및 2) MQTT 트래픽 시나리오로 평가한다. 대부분의 IoT 트래픽은 HTTP 또는 MQTT 프로토콜을 사용하여 생성되기 때문이다. 두 시나리오 모두에서, IoT 트래픽을 전송하는 데 필요한 CPU 사이클 및 작업 완료 시간(job completion time)을 측정한다.
- [0098] 마이크로 벤치마크로는, TCP 패킷 크기를 32 B에서 1 KB로 점진적으로 증가시킴으로써 CPU 사용량 및 네트워크 처리량을 평가한다. 이러한 패킷 크기 범위에는 앞서 설명한 바와 같이 IoT 패킷과 보편적인 데이터 센터 패킷(common datacenter packets)이 모두 포함된다. 또한, 도 4와 유사한 상세 프로파일링을 사용하여 각 패킷 및 함수 심볼(function symbol)의 처리 시간을 분석한다. 또한, IoT 트래픽의 FCT(Flow Completion Time)를 측정하고 결과에서 99% 테일 값(tail values)을 제시한다. 마지막으로, UDP 프로토콜을 사용한 제안 기법(SCON)의 성능을 살펴본다. 각 실험은 최소 다섯 번 이상 진행되며 평균 값을 기록한다. 네이티브 Linux 프로세스, 컨테이너, 및 SCON을 비교한다.
- [0099] 1. 실제 응용 프로그램(Real-world Applications)
- [0100] 1.1 HTTP. 우선 제안 기법(SCON)의 HTTP 성능을 테스트한다. 이를 위해 라즈베리 파이(Raspberry Pi)에서 Nginx (버전 1.23.2)를 실행하여 HTTP 트래픽을 생성한다. 반대편에서는, Intel 서버에서 wrk2 및 mpstat을 실행하여 네트워크 성능 및 CPU 사용량을 측정한다.
- [0101] IoT 장치로부터의 각 연결에 대해, wrk2는 1 개의 스레드(thread)를 생성하고 64 B HTML 파일을 요청한다. 설정에서, 컨테이너의 능력 상한에 가까운 2K requests/s로 입력을 설정한다. 또한, mpstat을 통해 CPU 사용량 및 wrk2에서 측정된 HTTP 지연을 측정한다. 여기서 측정된 지연은 요청(request)을 보내고 대응하는 응답(reply)을 받는 데 걸리는 시간이다. 각 실험은 10초 동안 진행된다.
- [0102] CPU 사용량(CPU usage). 도 8a는 CPU 사용량을 상세히 나타낸다. 실험에서 동일한 64 B 파일을 사용했지만, 네이티브(native), 컨테이너(container), 및 제안 기법(SCON)은 매우 다른 CPU 사용량을 보인다. 컨테이너와 비교하여, 제안 기법(SCON)은 CPU 사용량을 19% 감소시킨다. 특히, 제안 기법(SCON)은 Soft IRQ 처리에 소요되는 CPU 사이클을 26% 감소시킨다. 결과적으로, 필요한 처리량을 얻기 위한 전체 CPU 사용량에서 네이티브(native) 및 컨테이너(container) 간의 격차는, 제안 기법(SCON)을 사용하면, 25% 감소한다.
- [0103] HTTP 지연(HTTP latency). 도 8b는 wrk2에 의해 보고된 평균 HTTP 지연을 도시한다. 여기서 바(bars)는 평균 값을 나타내고, 위스커(whisker)는 범위를 나타낸다. 컨테이너(container)는 네이티브(native)에 비해 훨씬 뚜렷한 HTTP 지연을 나타내며, 14배 더 길어진다. 반면에, 제안 기법(SCON)은 컨테이너(container)에 비해 지연을 10배 감소시킨다. 특히, 네이티브(native)와 컨테이너(container) 간의 지연 차이는 1325%이고, 네이티브(native)와 제안 기법(SCON) 간의 지연은 31%로, 42배 향상된다.
- [0104] 1.2 MQTT. 다음으로, IoT 장치에서 널리 이용되는 데이터 스트리밍 프로토콜인 MQTT를 평가한다. MQTT는 데이터 게시자(data publishers, 예 : IoT 장치) 및 구독자(subscribers, 예 : 엣지 클라우드 서버) 사이에 개입하는 "MQTT 브로커(MQTT broker)"를 제공한다. MQTT 브로커는 다양한 데이터를 게시자로부터 받아 적절한 구독자에게 전달한다. 실험을 위해, 버전 2.0.15의 MQTT 프로토콜을 사용한다.
- [0105] 다음과 같이 MQTT 구성 요소를 실행하기 위해, MQTTLoader와 Mosquitto를 사용한다. MQTT 게시자는 라즈베리 파이(Raspberry Pi)에서 실행되며, 브로커와 구독자는 Intel 서버에서 실행된다. MQTTLoader를 사용하여 MQTT 게시자(Raspberry Pi)에서 각각 64 B 데이터로 구성된 메시지를 생성한다. 각 시도(trial) 동안, 60초 동안 100만 개의 메시지를 전송한다. 브로커는 100만 개의 메시지를 수신하고, 그런 다음 이를 구독자(Intel 서버)에게 전달한다.
- [0106] 실험을 적어도 다섯 번 반복하고 평균 값을 제시한다. CPU 사용량과 MQTT 지연을 측정한다. MQTT 지연은 게시자에 의한 메시지 생성부터 구독자에 의한 메시지 수신까지의 처리 시간을 포함한다. CPU 사용량은 mpstat에 의해 측정되고, 지연은 MQTTLoader에 의해 측정된다.

- [0107] CPU 사용량(CPU usage). 도 9a는 MQTT 트래픽의 CPU 사용량을 도시한다. 네이티브 환경과 컨테이너 환경을 비교할 때, 컨테이너 환경은 패킷 처리에 대해 훨씬 더 많은 CPU 사이클을 소비하고(SoftIRQ), 이는 2.2배 더 높다. 이러한 SoftIRQ에서의 높은 처리로 인해, 컨테이너 환경의 사용자 시간은 CPU 사이클 감소(0.87배)를 경험하며 그 결과가 나쁜 성능으로 나타난다. 반면, 제안 기법(SCON)은 컨테이너의 SoftIRQ 소비를 24% 감소시켜 MQTT 성능을 향상시킨다.
- [0108] MQTT 지연(MQTT latency). 도 9b는 MQTT 지연을 도시한다. 바(bars)로 평균 값을, 위스커(whiskers)로 범위를 표시한다. 네이티브 환경과 컨테이너 환경을 비교할 때, 컨테이너 환경의 지연은 1.5배 증가한다. 반면에, 컨테이너 환경을 제안 기법(SCON)과 비교할 때, 제안 기법(SCON)은 지연을 31% 감소시킨다. 더 나아가, 네이티브 환경과 제안 기법(SCON)을 비교할 때, 제안 기법(SCON)의 지연은 네이티브 환경보다 5% 높을 뿐이다(최소 2%). 결과는, 제안 기법(SCON)이 MQTT 지연을 크게 개선하여 IoT 디바이스의 실제 응용 프로그램 시나리오를 크게 도와준다는 것을 의미한다.
- [0109] 2. Micro-benchmarks
- [0110] 2.1 CPU 사용량(CPU usage). iperf를 사용하여 TCP 패킷을 생성하고 mpstat을 사용하여 CPU 사용량(%)을 측정한다. 상술한 바와 유사하게, 패킷 크기를 1) IoT 트래픽(32 B에서 256 B, 도 10a)과 2) 보편적인 클라우드 트래픽(512 B 및 1 KB, 도 10b)으로 분류한다. 도 10의 각 바(bars)는 패킷 크기(x축)별 SoftIRQ 처리에 대한 평균 CPU 사이클(y축)을 나타낸다.
- [0111] CPU 사용량에 대해서는, SoftIRQ 부분에 집중한다. 왜냐하면, 이는 제안 기법(SCON)이 개선하려는 패킷 처리 오버헤드에 해당하기 때문이다. 앞서 언급한 대로, 모든 실험은 하나의 CPU 코어에서 수행되며, 모든 경우에 CPU 코어가 완전히 활용된다(100%). 도 10에서 제시된 SoftIRQ 사용량을, 100%(하나의 코어)의 총 CPU 사용량에서 빼줌으로써, 주로 사용자에게 의해 활용되는 CPU 사용량을 얻을 수 있다. 따라서, SoftIRQ 사용을 최소화하는 것은 IoT 응용 프로그램에 더 많은 CPU 사이클을 확보하는 데 도움이 된다.
- [0112] 도 10은 패킷 크기가 증가함에 따라 모든 경우에 SoftIRQ의 CPU 사용량이 증가함을 보여준다. 이는, 패킷 크기가 증가함에 따라 사용자 수준에서 생성되는 패킷 수와 시스템 수준에서 처리되는 시스템 호출 수가 감소하기 때문이다. 도 10a는 또한, 제안 기법(SCON)이 컨테이너에 비해 SoftIRQ에서 소비되는 CPU 사이클을 평균 20% 감소시킨 것을 보여준다. 특히, 64 B 패킷의 경우 CPU 사용량이 26% 감소한다. 보편적인 클라우드 트래픽(도 10b)의 경우에도, 제안 기법(SCON)은 SoftIRQ 처리에서 컨테이너에 비해 15%에서 17% 감소를 보여준다.
- [0113] 2.2 네트워크 처리량(Network throughput). 측정된 네트워크 처리량(Mbps)은 도 11에 도시되어 있다. 제안 기법(SCON)은 컨테이너의 네트워크 처리량을 향상시킨다. IoT 트래픽의 경우, 도 11a에서 제안 기법(SCON)은 컨테이너에 비해 평균 12% 높은 네트워크 처리량을 보여준다. 또한, 제안 기법(SCON)은 네이티브와 컨테이너 간의 차이를 평균 19% 감소시킨다. 64 B 패킷을 처리할 때, 제안 기법(SCON)은 컨테이너의 처리량을 최대 18% 향상시키고 컨테이너와 네이티브 간의 차이를 최대 23% 줄인다. 보편적인 클라우드 트래픽의 경우, 제안 기법(SCON)은 도 11b에서 평균 37% 높은 네트워크 처리량을 보여주며 512 B 패킷에서 최대 45%까지 향상된다. 또한, 네이티브와 컨테이너 간의 네트워크 처리량 차이를 평균 40% 및 최대 47%로 감소시킨다. 이러한 결과는, 제안 기법(SCON)이 CPU 사용량을 효과적으로 감소시키고 컨테이너 네트워킹 처리량을 IoT 및 클라우드 트래픽 모두에서 향상시키는 효과적인 역할을 하는 것을 보여준다.
- [0114] 2.3 패킷 당 지연(Per packet latency). 패킷 크기를 32 B에서 1 KB로 변경하여 Netperf를 사용하여 패킷당 지연 시간을 측정한다. 도 12a와 도 12b는 각각 10 초 동안 IoT 클라이언트에서 Linux 엡지 서버로 TCP 패킷을 보내는 데 걸리는 평균 및 99% 테일 지연(tail latency)을 도시한다. 제안 기법(SCON)은 평균 지연을 컨테이너에 비해 평균 25% 감소시키고, 네이티브 및 컨테이너 간의 차이를 평균 49% 감소시킨다. 99% 테일 지연(tail latency)의 경우, 제안 기법(SCON)은 평균적으로 컨테이너의 지연을 32% 감소시키고, 네이티브 프로세스와 컨테이너 간의 차이를 평균 66%로 최소화한다.
- [0115] 2.4 함수 심볼 당 처리 시간(Processing time per function symbol). 여기서는, 앞서 보여준 것과 동일한 프로파일링을 수행하여 제안 기법(SCON)의 영향을 더 자세히 살펴본다. 64 B 크기의 TCP 패킷을 보낼 때, IoT의 컨테이너의 네트워킹 스택(networking stack)에서 각 함수를 처리하는 데 걸리는 시간을 측정한다. 도 13은 네이티브, 컨테이너, 및 제안 기법(SCON)에서 평균적인 패킷 실행 시간을 도시한다. 각 바(bar)는, ip_rcv, br_handle_frame, 및 veth_xmit와 같은 프로파일링한 함수들을 포함한다.
- [0116] 제안 기법(SCON)은, 컨테이너와 비교하여, 단일 패킷을 처리하는 데 소요되는 전체 시간을 43% 감소시킨다. 컨

테이너의 CPU 사용량이 네이티브 대비 99% 증가하는 반면, SCON과 네이티브 간의 차이는 20%로, 5배의 개선을 나타낸다. 보다 구체적으로, 제안 기법(SCON)은 ip_rcv에서 소요되는 시간을 $22\mu s$ 절약하면서, 다른 심볼에서 소요되는 시간을 컨테이너와 유사하게 유지한다. 이는, 제2 패킷 처리 경로(SCON express forward)가 기존 컨테이너 네트워킹 스택에서 발생하는 추가 및 반복적인 패킷 처리로 인한 오버헤드를 성공적으로 감소시킴을 의미한다.

[0117] 2.5 플로우 완료 시간(Flow Completion Time, FCT). 많은 IoT 패킷의 수행을 위한 시간인 FCT를 측정한다. 라즈베리 파이(Raspberry Pi)에서 트래픽 생성기를 사용하여 IoT 트래픽을 생성한다. 패킷은 센서 데이터의 IoT 플로우의 크기 분포에 따라 생성된다. 각 시도(trial)에서, Intel 서버로 TCP 연결을 통해 100개의 IoT 트래픽 플로우를 전송한다. 각 IoT 플로우는 분포에 따라 여러 패킷으로 구성된다. 한 플로우를 전송한 후에, 해당 플로우를 완료하는 데 걸리는 시간(FCT)을 측정한다. 100개의 FCTs로부터, 99% 테일 값(tail value)을 계산한다. 측정을 적어도 다섯 번 수행하고 평균값을 제시한다. 측정 결과는 도 14에 도시되어 있다. 제안 기법(SCON)과 컨테이너를 비교할 때, 제안 기법(SCON)은 99% 테일 FCT를 20% 감소시킨다. 더 나아가, 제안 기법(SCON)은 네이티브와 컨테이너 간의 차이를 12ms에서 2.9ms로 축소하여 4.1배 감소시킨다.

[0118] 2.6 UDP에서의 제안 기법(SCON on UDP). IoT 도메인에서, 저전력 통신(low-power communication, 예컨대 CoAP)을 용이하게 하기 위하여, 경량 프로토콜(lightweight protocol)로써 UDP가 활용된다. 제안 기법(SCON)은 IP 계층에 대한 최적화 기술이므로, 제안 기법(SCON)은 어떠한 수정 없이도 UDP와 호환된다. 여기서, TCP 평가와 유사한 설정을 사용하여, 그러나 iperf로부터 UDP 트래픽을 생성하여, CPU 사용량 및 네트워크 처리량을 측정한다.

[0119] 먼저, 도 15a와 도 15b에 나와 있는 IoT 트래픽 및 클라우드 트래픽에 대한 SoftIRQ의 CPU 사용량 결과를 설명한다. 메시지 크기(x-축)가 증가함에 따라 IoT와 클라우드 트래픽 모두, 제안 기법(SCON)이 SoftIRQ에서 소비되는 CPU 사이클을 평균 33% 줄인다는 것을 보여준다. 또한, 32 B에서 1 KB로 패킷 크기가 증가할 때, 컨테이너는 CPU 사이클 소비에서 1.7% 증가만 경험하며, 이는 무시할 수 있는 수준이다.

[0120] 다음으로, 도 16a와 도 16b는 각각 IoT 및 클라우드 트래픽에 대한 UDP의 네트워크 처리량을 도시한다. IoT 트래픽의 경우(도 16a), 제안 기법(SCON)은 평균적으로 컨테이너보다 32% 높은 처리량을 나타낸다. 또한, 제안 기법(SCON)은 네이티브 프로세스와 컨테이너 간의 처리량 차이를 평균 39% 줄인다. 보편적인 클라우드 트래픽의 경우(도 16b), 제안 기법(SCON)은 평균적으로 처리량을 29% 향상시킨다. 제안 기법(SCON)을 사용하면, 네이티브 프로세스와 컨테이너 간의 처리량 차이가 평균 28% 감소한다. 이러한 결과는, 제안 기법(SCON)이 UDP 트래픽뿐만 아니라 TCP 트래픽에 대한 CPU 사용량과 처리량을 효과적으로 개선한다는 것을 나타낸다.

[0121] 도 17은 본 발명의 일 실시예에 따른, 패킷 처리 방법을 설명하기 위한 흐름도이다.

[0122] 도 17을 참조하면, 적어도 프로세서(processor) 및/또는 메모리(memory)를 포함하는 컴퓨팅 장치에 의해 수행되는 (컨테이너의) 패킷 처리 방법은 (컨테이너의) 패킷 경량화 방법 등으로 명명될 수도 있다. 컴퓨팅 장치는 PC(Personal Computer), 서버(server), 클라우드 서버, 엣지 클라우드 서버 등을 포함할 수 있다. 실시예에 따라, 컴퓨팅 장치는 패킷 처리 장치나 패킷 경량화 장치 등으로 명명될 수도 있다. 또한, 컴퓨팅 장치는 물리적으로 구분되는 2 이상의 장치에 의해 구현되는 것도 가능하다.

[0123] 패킷 처리 장치는 라즈베리 파이와 같은 IoT 장치에 컨테이너 환경을 제공하며, IoT 장치로부터 소정의 패킷을 수신한 후 패킷 처리를 수행하여 외부 네트워크로 패킷을 전송하는 역할을 수행할 수 있다. IoT 장치 또한 적어도 프로세서 및/또는 메모리를 포함하는 컴퓨팅 장치를 의미할 수 있다.

[0124] 또한, 패킷 처리 장치와 복수의 IoT 장치들은 스마트 팩토리 시스템, 스마트 팜 시스템, 스마트 홈 시스템 등과 같은 스마트 시스템을 구성할 수 있다.

[0125] 이하에서, 패킷 처리 방법을 설명함에 있어, 앞선 기재와 중복되는 내용에 관하여는 그 구체적인 기재를 생략하기로 한다.

[0126] 우선, IoT 장치에 할당된 컨테이너로부터 패킷이 수신된다(S110). 패킷은 TCP 패킷 및/또는 UDP 패킷일 수 있다. 여기서, 패킷은 IoT 장치로부터 수신된 패킷에 대하여 컨테이너가 패킷 처리를 수행한 후의 결과물을 의미할 수도 있고, 컨테이너에서의 패킷 처리는 상술하였으므로 이에 대한 기재는 생략한다. 또한, 컨테이너로부터의 패킷은 (리눅스) 브릿지를 통해 수신될 수 있다. 또한, IoT 장치는, 예컨대, 온도, 습도, 풍속, 조도 등에 대한 감지 결과(센싱 데이터)를 패킷 처리 장치로 송신하는 소정의 컴퓨팅 장치를 의미할 수 있다.

- [0127] 수신된 패킷에 대하여, 패킷 처리 경로 또는 패킷 처리 과정이 결정된다(S120). 패킷 처리 경로는 제1 패킷 처리 경로와 제2 패킷 처리 경로 중 어느 하나로 결정될 수 있다. 패킷 처리 경로는, 패킷 처리 장치에 미리 저장되어 있는 테이블(패킷 처리 결과 저장 테이블)에 기초하여 결정될 수 있다. 즉, 수신된 패킷의 연결과 동일한 연결 정보를 갖는 패킷에 대한 정보가 테이블에 미리 저장되어 있다면 수신된 패킷은 제2 패킷 처리 경로로 전달되고, 수신된 패킷의 연결과 동일한 연결 정보를 갖는 패킷에 대한 정보가 테이블에 저장되어 있지 않다면 수신된 패킷은 제1 패킷 처리 경로로 전달될 수 있다. 여기서, 동일한 연결이라 함은, 패킷의 소스 IP 어드레스, 목적지 IP 어드레스, 소스 포트 어드레스, 목적지 포트 어드레스, 및 L4 프로토콜 식별자가 동일한 패킷을 의미한다.
- [0128] S120 단계에서, 제1 패킷 처리 경로가 수신된 패킷을 처리할 경로로 결정된 경우, 패킷 처리 장치는 제1 패킷 처리 과정을 수행하여 수신된 패킷을 처리한다(S130). S120 단계에서, 제2 패킷 처리 경로가 수신된 패킷을 처리할 경로로 결정된 경우, 패킷 처리 장치는 제2 패킷 처리 과정을 수행하여 수신된 패킷을 처리한다(S140).
- [0129] 구체적으로, 제1 패킷 처리 과정은, 예컨대, 오리지널 리눅스 네트워킹 스택의 호출 체인을 따를 수 있다. 제1 패킷 처리 과정 중에 넷필터, 라우팅 룩업, 및 헤더 처리의 결과는 수집될 수 있다.
- [0130] 제2 패킷 처리 과정은, 테이블을 이용하여, 반복적으로 수행되는 동작을 생략한다. 구체적으로, 제2 패킷 처리 과정은, 패킷에 필요한 헤더 정보(예컨대, 라우팅 룩업 결과, TTL 값, MTU 크기)를 패킷 헤더에 기재한다. 또한, 제2 패킷 처리 과정은 체크섬 동작을 더 포함할 수 있다.
- [0131] 또한, 제2 패킷 처리 과정은, 테이블에 저장되어 있는 넷필터 결정(넷필터 수행의 결과)에 따라 패킷을 네트워크 인터페이스로 전송하거나 드랍시키는 동작을 포함할 수 있다. 즉, 테이블에 포함된 넷필터 결정이 성공(success)라면 해당 패킷을 네트워크 인터페이스로 전송하고, 테이블에 포함된 넷필터 결정이 폐기(discard)라면 해당 패킷을 드랍할 수 있다.
- [0132] 실시예에 따라, 테이블에 포함된 넷필터 결과는 "성공"이고, 헤더 처리 결과는 "통과"인 경우, 해당 패킷은 네트워크 인터페이스로 전송될 수 있다. 테이블에 포함된 넷필터 결과가 "폐기"인 경우, 헤더 처리 결과와 무관하게, 해당 패킷은 드랍될 수 있다. 또한, 패킷에 대한 헤더 처리 결과가 "오류"인 경우, 해당 패킷은 드랍되고, 해당 패킷과 동일한 연결 정보를 갖는 다음 패킷은 제1 패킷 처리 경로를 통과하도록 테이블이 업데이트될 수 있다(예컨대, 해당 패킷에 대한 정보를 삭제).
- [0133] 제1 패킷 처리 과정 또는 제2 패킷 처리 과정을 수행한 후, 패킷은 외부 네트워크로 전송된다(S150). 네트워크 인터페이스로 전송된 패킷은 물리적인 NIC(Network Interface Card)를 통해 외부 네트워크로 전송될 수 있다.
- [0134] 테이블 업데이트가 수행된다(S160). 도 17에서, S160 단계는 S150 단계 이후에 수행되는 것으로 도시되어 있으나, 실시예에 따라, S160 단계는 S130 단계 이후에 수행되는 것도 가능하다. 업데이트는, 새로운 연결 정보를 갖는 패킷에 대한 정보의 추가, 컨테이너가 폐기되거나 해당 연결에 대한 정보가 히트된지 소정 기간이 경과된 경우의 정보 삭제, 헤더 처리 결과가 오류인 경우의 정보 삭제 등을 포함한다.
- [0135] 이상에서 설명된 장치는 하드웨어 구성 요소, 소프트웨어 구성 요소, 및/또는 하드웨어 구성 요소 및 소프트웨어 구성 요소의 집합으로 구현될 수 있다. 예를 들어, 실시예들에서 설명된 장치 및 구성 요소는, 예를 들어, 프로세서, 컨트롤러, ALU(Arithmetic Logic Unit), 디지털 신호 프로세서(Digital Signal Processor), 마이크로컴퓨터, FPA(Field Programmable array), PLU(Programmable Logic Unit), 마이크로프로세서, 또는 명령(instruction)을 실행하고 응답할 수 있는 다른 어떠한 장치와 같이, 하나 이상의 범용 컴퓨터 또는 특수 목적 컴퓨터를 이용하여 구현될 수 있다. 처리 장치는 운영 체제(Operation System, OS) 및 상기 운영 체제 상에서 수행되는 하나 이상의 소프트웨어 애플리케이션을 수행할 수 있다. 또한, 처리 장치는 소프트웨어의 실행에 응답하여, 데이터를 접근, 저장, 조작, 처리 및 생성할 수도 있다. 이해의 편의를 위하여, 처리 장치는 하나가 사용되는 것으로 설명된 경우도 있지만, 해당 기술 분야에서 통상의 지식을 가진 자는, 처리 장치가 복수 개의 처리 요소(Processing Element) 및/또는 복수 유형의 처리 요소를 포함할 수 있음을 알 수 있다. 예를 들어, 처리 장치는 복수 개의 프로세서 또는 하나의 프로세서 및 하나의 컨트롤러를 포함할 수 있다. 또한, 병렬 프로세서(Parallel Processor)와 같은, 다른 처리 구성(Processing Configuration)도 가능하다.
- [0136] 소프트웨어는 컴퓨터 프로그램(Computer Program), 코드(Code), 명령(Instruction), 또는 이들 중 하나 이상의 조합을 포함할 수 있으며, 원하는 대로 동작하도록 처리 장치를 구성하거나 독립적으로 또는 결합적으로(Collectively) 처리 장치를 명령할 수 있다. 소프트웨어 및/또는 데이터는, 처리 장치에 의하여 해석되거나 처리 장치에 명령 또는 데이터를 제공하기 위하여, 어떤 유형의 기계, 구성 요소(Component), 물리적 장치, 가상

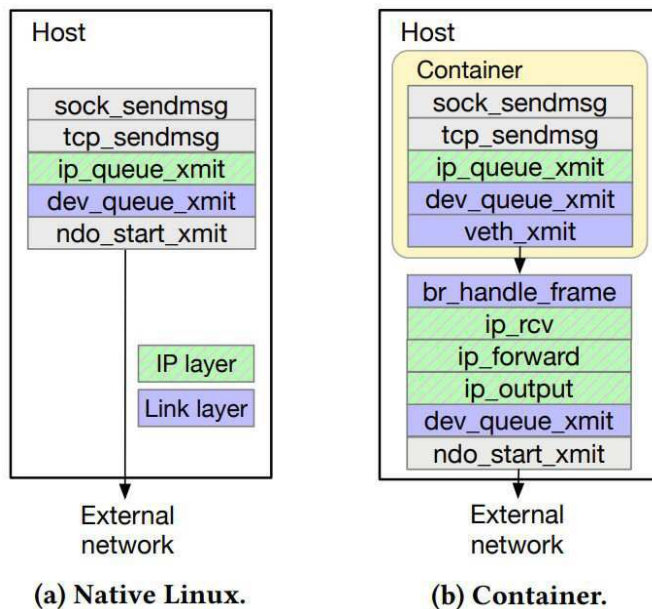
장치(Virtual Equipment), 컴퓨터 저장 매체 또는 장치, 또는 전송되는 신호 파(Signal Wave)에 영구적으로, 또는 일시적으로 구체화(Embody)될 수 있다. 소프트웨어는 네트워크로 연결된 컴퓨터 시스템 상에 분산되어서, 분산된 방법으로 저장되거나 실행될 수도 있다. 소프트웨어 및 데이터는 하나 이상의 컴퓨터 판독 가능 기록 매체에 저장될 수 있다.

[0137] 실시예에 따른 방법은 다양한 컴퓨터 수단을 통하여 수행될 수 있는 프로그램 명령 형태로 구현되어 컴퓨터 판독 가능 매체에 기록될 수 있다. 상기 컴퓨터 판독 가능 매체는 프로그램 명령, 데이터 파일, 데이터 구조 등을 단독으로 또는 조합하여 포함할 수 있다. 상기 매체에 기록되는 프로그램 명령은 실시예를 위하여 특별히 설계되고 구성된 것들이거나 컴퓨터 소프트웨어 당업자에게 공지되어 사용 가능한 것일 수도 있다. 컴퓨터 판독 가능 기록 매체의 예에는 하드 디스크, 플로피 디스크 및 자기 테이프와 같은 자기 매체(Magnetic Media), CD-ROM, DVD와 같은 광기록 매체(Optical Media), 플롭티컬 디스크(Floptical Disk)와 같은 자기-광 매체(Magneto-optical Media), 롬(ROM), 램(RAM), 플래시 메모리 등과 같은 프로그램 명령을 저장하고 수행하도록 특별히 구성된 하드웨어 장치가 포함된다. 프로그램 명령의 예에는 컴파일러에 의해 만들어지는 것과 같은 기계어 코드뿐만 아니라 인터프리터 등을 사용해서 컴퓨터에 의해서 실행될 수 있는 고급 언어 코드를 포함한다. 상기된 하드웨어 장치는 실시예의 동작을 수행하기 위해 하나 이상의 소프트웨어 모듈로서 작동하도록 구성될 수 있으며, 그 역도 마찬가지이다.

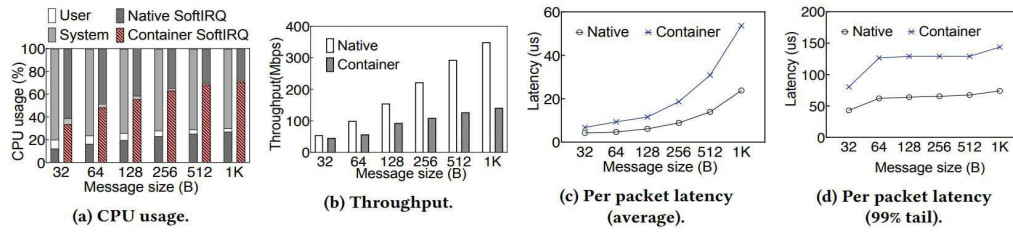
[0138] 본 발명은 도면에 도시된 실시예를 참고로 설명되었으나 이는 예시적인 것에 불과하며, 본 기술 분야의 통상의 지식을 가진 자라면 이로부터 다양한 변형 및 균등한 타 실시예가 가능하다는 점을 이해할 것이다. 예를 들어, 설명된 기술들이 설명된 방법과 다른 순서로 수행되거나, 및/또는 설명된 시스템, 구조, 장치, 회로 등의 구성 요소들이 설명된 방법과 다른 형태로 결합 또는 조합되거나, 다른 구성 요소 또는 균등물에 의하여 대치되거나 치환되더라도 적절한 결과가 달성될 수 있다. 따라서, 본 발명의 진정한 기술적 보호 범위는 첨부된 등록청구범위의 기술적 사상에 의해 정해져야 할 것이다.

도면

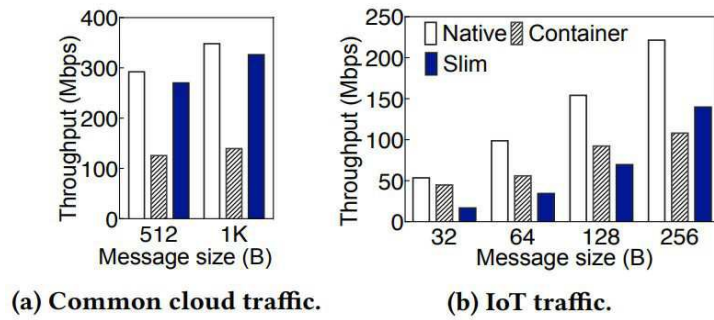
도면1



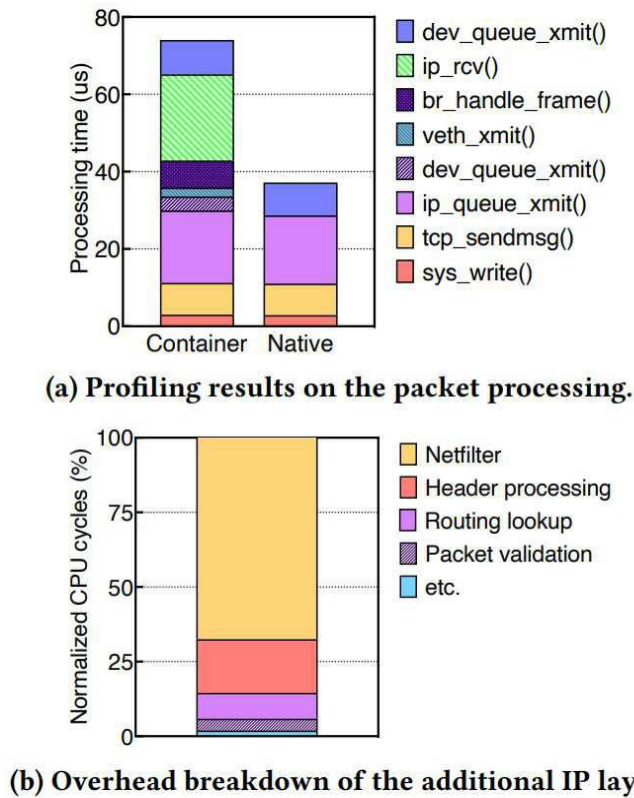
도면2



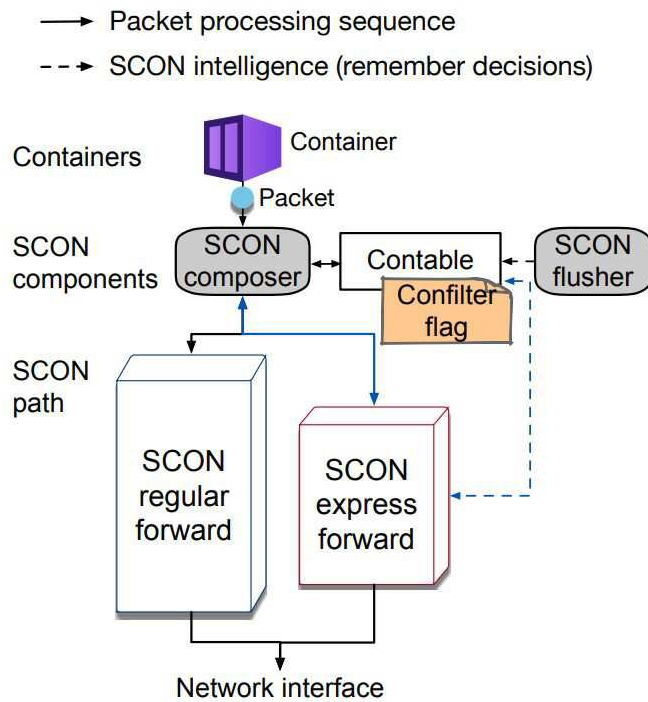
도면3



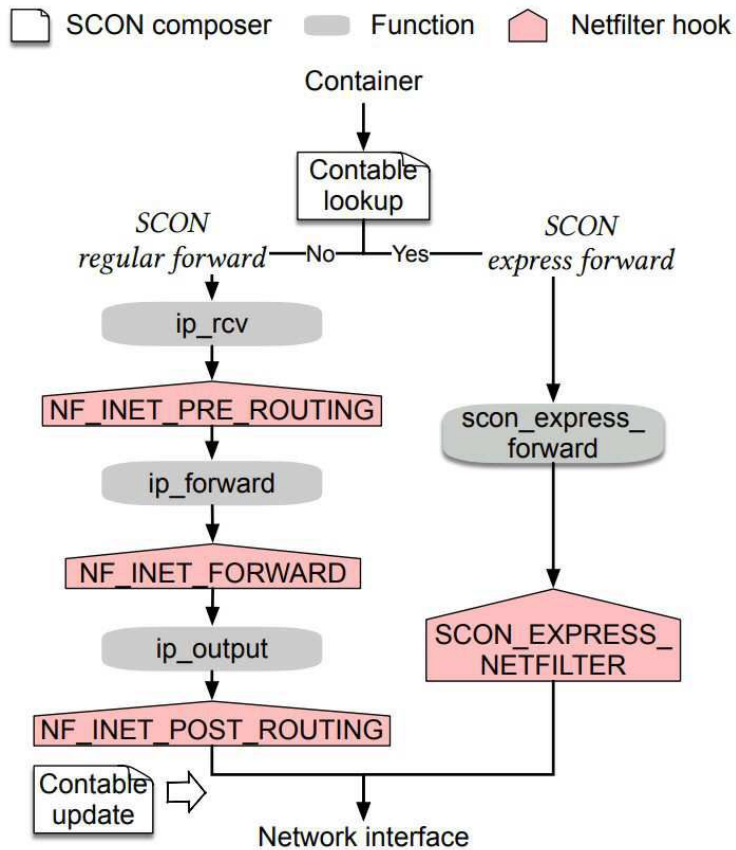
도면4



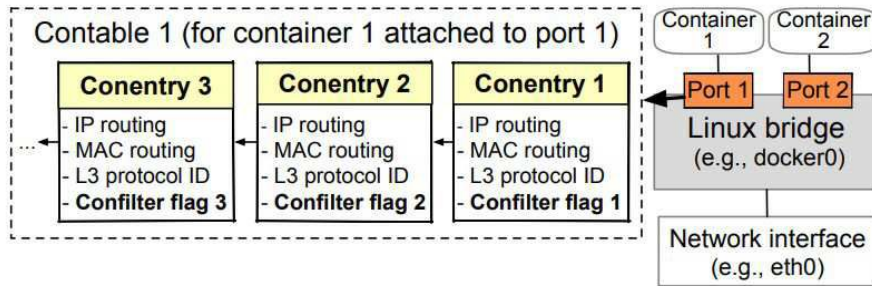
도면5



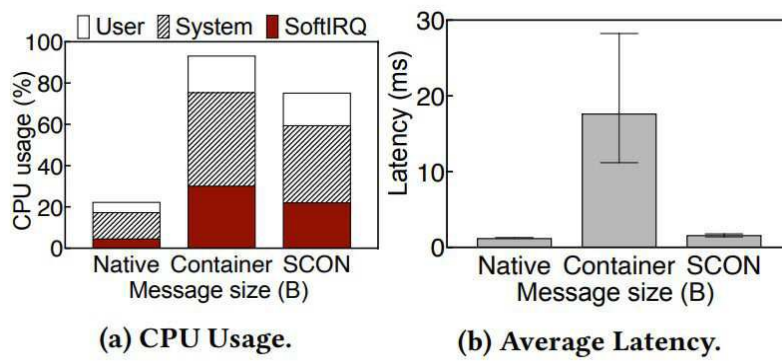
도면6



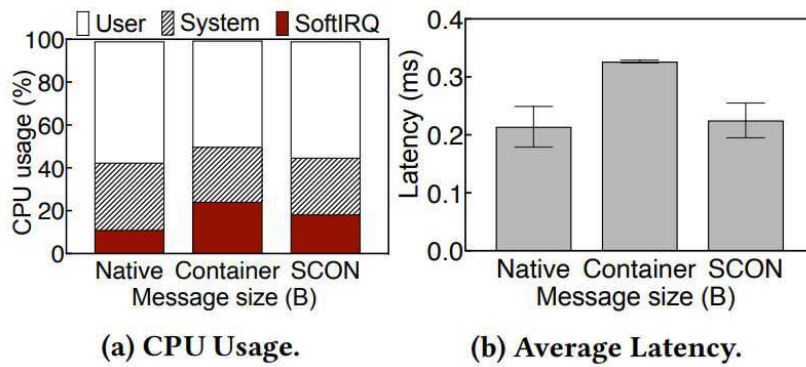
도면7



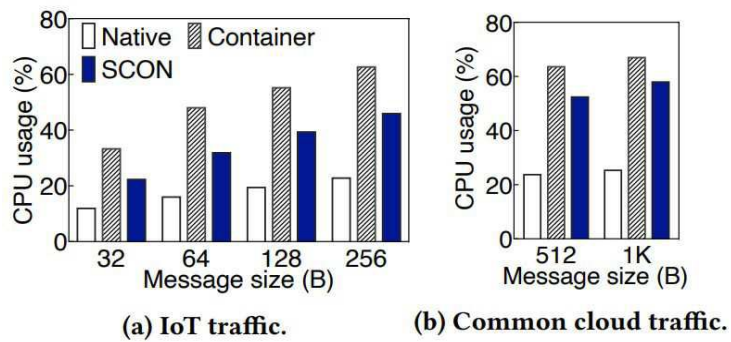
도면8



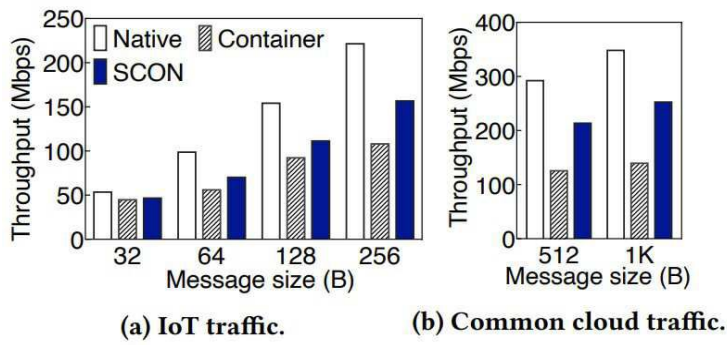
도면9



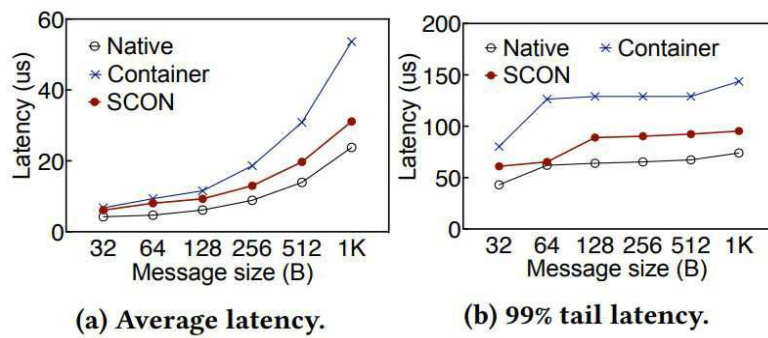
도면10



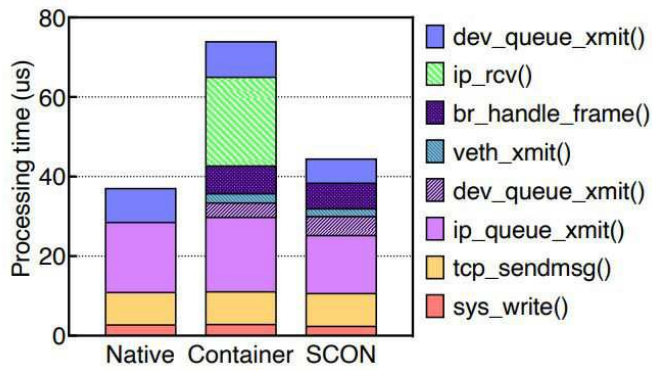
도면11



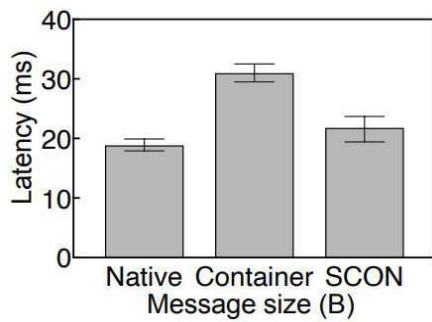
도면12



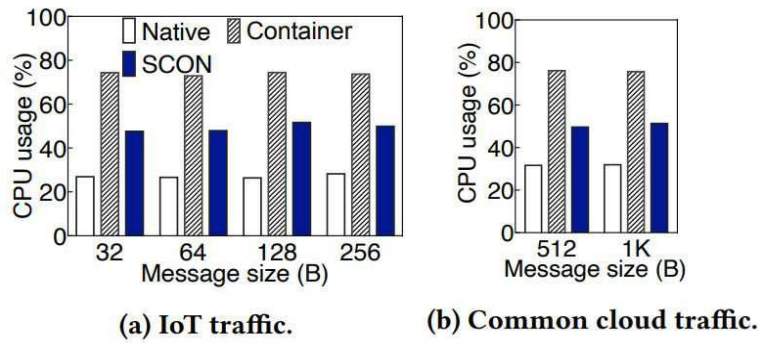
도면13



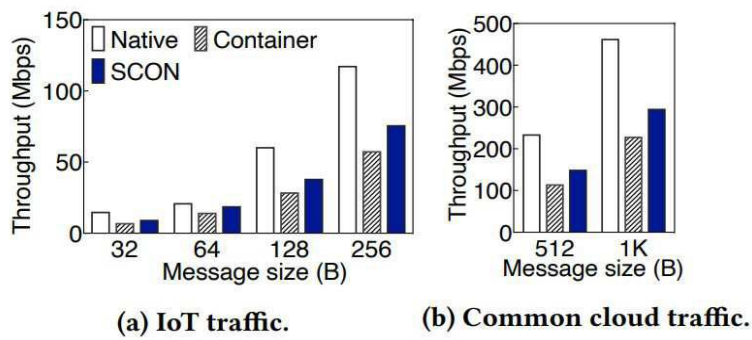
도면14



도면15



도면16



도면17

