OpenThread 사용 가이드

2025 년 5 월

경북대학교 사물인터넷표준연구실

김현수 (howeve18@gmail.com),

이동재 (leedongjae625@gmail.com)

요 약

사물인터넷(IoT) 기술의 확산과 함께 수많은 저전력 기기들이 안정적으로 연결될 수 있는 무선 네트워크의 필요성이 커지고 있다. Thread 는 이러한 요구를 충족하기 위해 설계된 IPv6 기반의 저전력 메쉬 네트워크 프로토콜로 보안성, 확장성, 자가 치유 능력 등을 갖춘 것이 특징이다. 이 기술 보고서는 OpenThread 사이트(https://openthread.io/?hl=ko)를 토대로, Thread 의 핵심 개념과 구조에 대해서 소개하고, Google 이 개발한 오픈소스 구현체인 OpenThread 의 특징과 활용 방안을 함께 다룬다.

목 차

1. 서론

본 기술 보고서에서는 사물인터넷(IoT) 환경에서 수많은 저전력 기기를 안정적으로 연결하기 위해 설계된 IPv6 기반 저전력 메쉬 네트워크 프로토콜인 Thread 의 핵심 개념과 구조를 소개하고, 라우터·종단 장치의 역할 분담, IPv6 주소 지정 방식, 네트워크 탐색 및 형성 과정을 설명하며, Google 이 개발한 오픈소스 구현체인 OpenThread 의 주요 특징과 실제 활용 방안을 다룬다.

2. Thread 에 대하여

2.1 Thread 란?

Thread 는 IEEE 802.15.4-2006 기반의 무선 메시 네트워크(WPAN)에서 저전력 사물 인터넷(IoT) 기기를 위해 설계된 IPv6 기반 네트워킹 프로토콜이다. 이 프로토콜은 ZigBee, Z-Wave, Bluetooth LE 와 같은 다른 802.15 메시 네트워크 기술들과는 독립적으로 작동하며, 다음과 같은 주요 특징을 갖고 있다.

첫째, 간단한 설치 및 작동을 통해 사용자 친화적인 단순성을 제공한다. 둘째, 네트워크 내 모든 장치를 인증하고 모든 통신을 암호화하여 높은 수준의 보안을 보장한다. 셋째, 단일 장애 지점이 없는 자체 복구 메시 구조와 스프레드 스펙트럼 기술을 활용하여 신뢰성과 간섭에 대한 저항성을 높였다. 넷째, 저전력 장치는 절전 모드로 수년간 유지되며 배터리로도 안정적으로 작동할 수 있어 에너지 효율성이 뛰어나다. 마지막으로, Thread 네트워크는 수백 개의 장치로 손쉽게 확장 가능하여 유연한 확장성을 제공한다.

따라서 OpenThread 를 애플리케이션에 적용하기 위해서는 이러한 Thread 의 기본 개념과 특성을 먼저 이해하는 것이 중요하다.

2.2 노드 역할 및 유형

Thread 네트워크에서 노드는 크게 두 가지 역할로 구분된다: 라우터와 종단 장치(End Device)이다.

라우터는 네트워크의 중추적인 역할을 수행하며, 패킷을 다른 기기로 전달하는 중계 기능을

제공한다. 또한, 네트워크에 새로 가입하려는 기기에게 보안 구성을 지원하여 안전한 네트워크 환경을 조성한다. 라우터는 항상 트랜시버(송수신 장치)를 활성화시켜 네트워크의 지속적인 연결과 안정성을 유지한다.

반면, 종단 장치는 주로 하나의 라우터와만 통신하며, 다른 기기의 패킷을 전달하지 않는다. 이 장치는 에너지 효율성을 고려해 필요할 때 트랜시버를 비활성화할 수 있으며, 이러한 특성 덕분에 배터리로 오랜 시간 작동하는 저전력 환경에 적합하다.

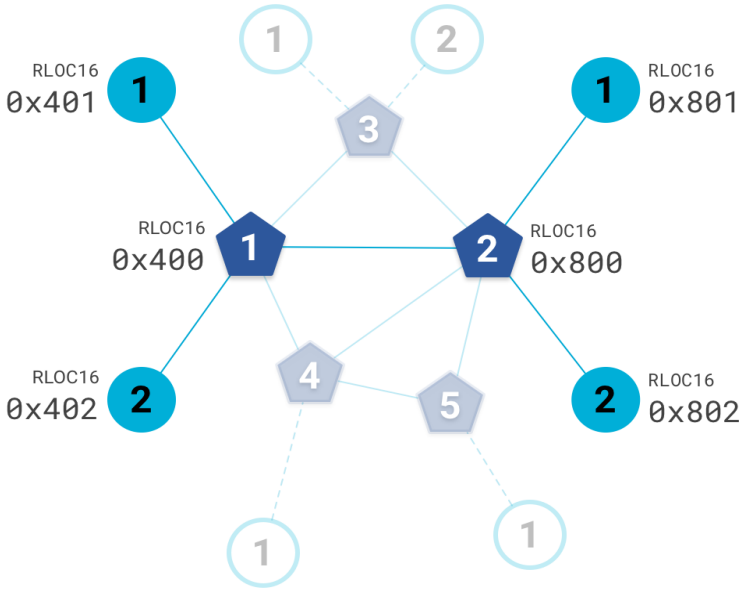
라우터와 종단 장치는 상위-하위 관계를 이루며, 종단 장치는 반드시 하나의 라우터에 연결되며, 라우터는 항상 상위 요소로 작동한다.

2.3 IPv6 주소 지정

Thread 네트워크에서는 각 장치를 식별하고 서로 통신하기 위해 다양한 유형의 주소를 사용한다. 특히 유니캐스트 주소는 단일 스레드 기기를 식별하는 데 사용되며, 주소의 범위(scope)에 따라 기능과 사용 목적이 다르다.

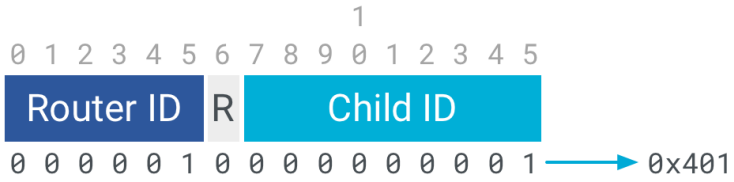
스레드 네트워크에서 유니캐스트 주소는 다음 세 가지 범위로 나뉜다.

1. **링크-로컬(Link-Local)**: 단일 무선 전송으로 도달 가능한 인터페이스 간의 통신에 사용되며, 접두사는 fe80::/16 이다.
2. **메시-로컬(Mesh-Local)**: 동일한 Thread 네트워크 내 모든 장치 간 통신을 위해 사용되며, 접두사는 fd00::/8 이다.
3. **전역(Global)**: 네트워크 외부와의 통신에 사용된다.



<그림 1> Thread 네트워크 내 RLOC16 주소 구성 예시

이 주소 체계에서 중요한 개념 중 하나는 라우팅 로케이터(RLOC, Routing Locator)이다. RLOC 는 Thread 네트워크 내 노드의 위치를 나타내는 IPv6 주소로, 각 장치의 라우터 ID 와 하위 ID 조합에 의해 생성된다. 라우터 자신을 가리키는 RLOC 의 하위 ID 는 항상 0 이며, 하위 장치는 라우터 ID 를 상위 요소로 가진다.



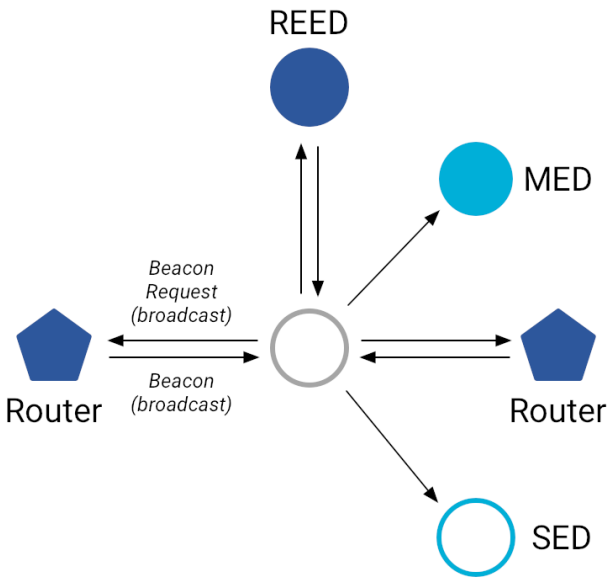
<그림 2> RLOC16 비트 필드 구조

RLOC 주소의 마지막 16 비트를 RLOC16 이라고 하며, 이는 IPv6 주소의 인터페이스 식별자(IID)의 일부로 사용된다. Thread 에서는 0000:00ff:fe00:RLOC16 형식의 IID 를 메시 로컬 프리픽스와 결합하여 전체 RLOC 주소를 생성한다. 이 구조를 통해 Thread 는 네트워크 토폴로지 상에서 각 장치의 위치를 명확히 식별하고 효율적인 라우팅을 수행할 수 있다.

2.4 네트워크 탐색 및 형성

Thread 네트워크는 2 바이트 PAN ID, 8 바이트 XPAN ID, 사람이 읽을 수 있는 네트워크 이름 세 가지 고유 식별자로 구분된다. 예를 들어 PAN ID 가 0xBEEF, XPAN ID 가 0xBEEF1111CAFE2222, 네트워크 이름이 yourThreadCafe 인 네트워크가 있을 수 있다.

새 Thread 네트워크를 생성하거나 기존 네트워크를 검색하려면 기기는 먼저 무선 범위 내 802.15.4 네트워크에 대해 활성 스캔을 수행한다. 이 과정에서 특정 채널에서 비콘 요청(Beacon Request)을 브로드캐스트하면, 범위 내의 라우터 또는 라우터 사용 가능 종단 기기(REED)가 PAN ID, XPAN ID, 네트워크 이름을 포함한 비콘을 브로드캐스트해 응답한다. 이러한 단계를 모든 채널에서 반복해 주변의 모든 네트워크를 탐지한다. 아래 그림과 같다

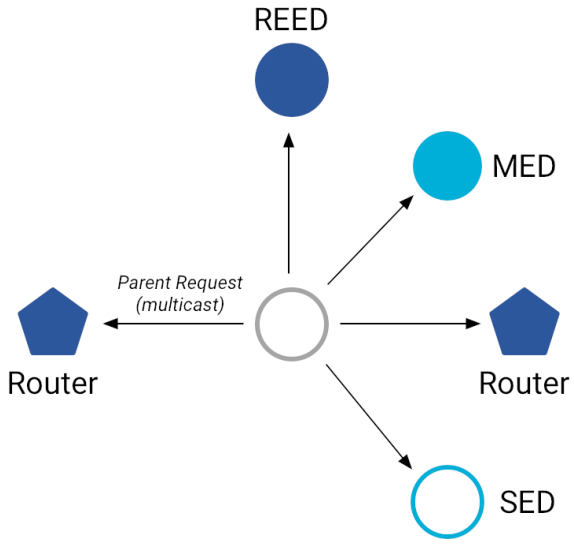


<그림 3> Thread 네트워크 활성 스캔 절차

링크 설정과 네트워크 정보 전파는 Mesh Link Establishment(MLE) 프로토콜을 통해 이뤄진다. MLE 는 주변 기기를 검색하고(Discover), 링크 품질을 평가하며(Quality), 실제 연결을 설정하고(Establish), 기기 유형·프레임 카운터·타임아웃 등 링크 매개변수를 협상하는 역할을 수행한다. 이 과정에서 리더 데이터(Leader RLOC, Partition ID, Partition weight), 네트워크 데이터(on-mesh prefixes, 주소 자동 구성, 경로 정보), 경로 전파 정보가 함께 공유된다. 경로 전파는 RIP 과 유사한 거리 벡터 방식으로 동작한다. 다만 MLE 는 커미셔닝을 통해 네트워크 자격 증명을 획득한 이후에만 진행된다

2.4.1 부모 요청

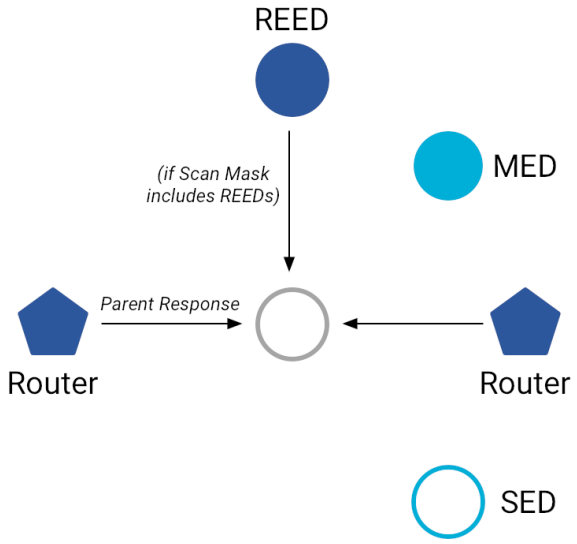
하위 기기는 부모 요청(Parent Request) 단계에서 멀티캐스트 형식의 부모 요청 메시지를 전송해 이웃 라우터 및 REED 를 검색한다. 이 메시지에는 Mode(기기 특성), Challenge(재전송 공격 방지), Scan Mask(라우터 전용 또는 라우터+REED 포함 여부)가 포함된다. 아래 그림을 참조한다.



<그림 4> Thread 네트워크의 Parent Request 단계

2.4.2 부모 응답

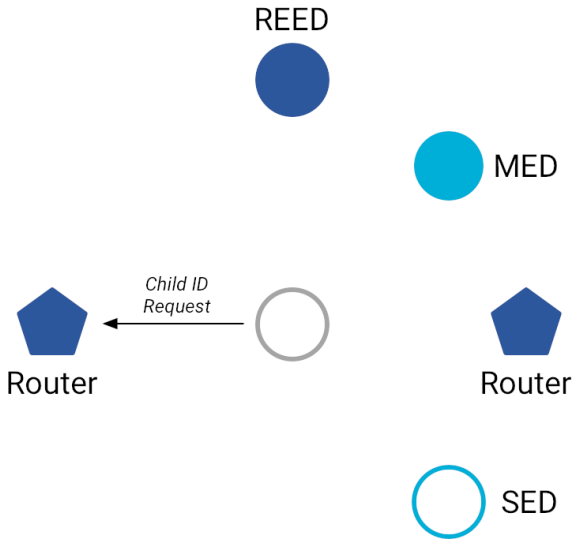
이어서 이웃 라우터 또는 REED 는 부모 응답(Parent Response) 메시지를 유니캐스트로 보내 링크 설정 정보를 제공한다. 메시지에는 Version(프로토콜 버전), Response(요청 Challenge 복사본), Link/MLE Frame Counters, Source Address(RLOC16), Link Margin(수신 신호 품질), Connectivity(연결 수준), Leader Data, Challenge(하위 ID 요청 시 검증용) 등이 포함된다. 아래 그림을 참조한다.



<그림 4> Thread 네트워크의 Parent Response 단계

2.4.3 하위 ID 요청

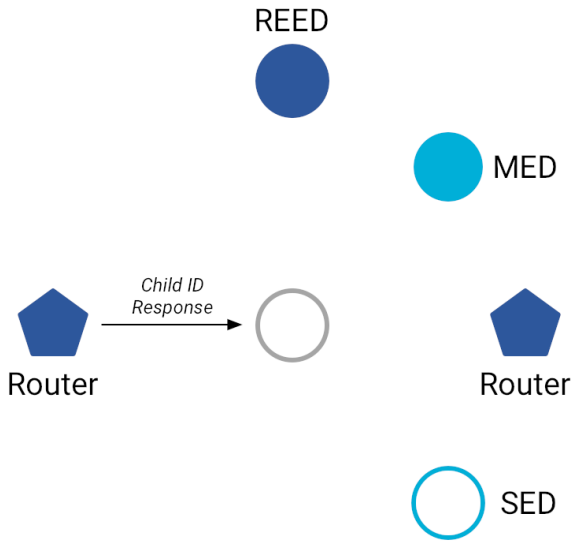
하위 기기는 선택한 부모에게 하위 ID 요청(Child ID Request) 메시지를 유니캐스트로 전송해 자식-부모 링크를 설정한다. 메시지에는 Version, Response Challenge, Link/MLE Frame Counters, Mode(기기 종류), Timeout(비활성 시간 제한), Address Registration(MED/SED 의 IPv6 주소 등록) 등이 포함된다. REED 대상인 경우 수락 전에 라우터로 업그레이드된다. 아래 그림을 참조한다.



<그림 5> Thread 네트워크의 Child ID Request 단계

2.4.4 하위 ID 요청

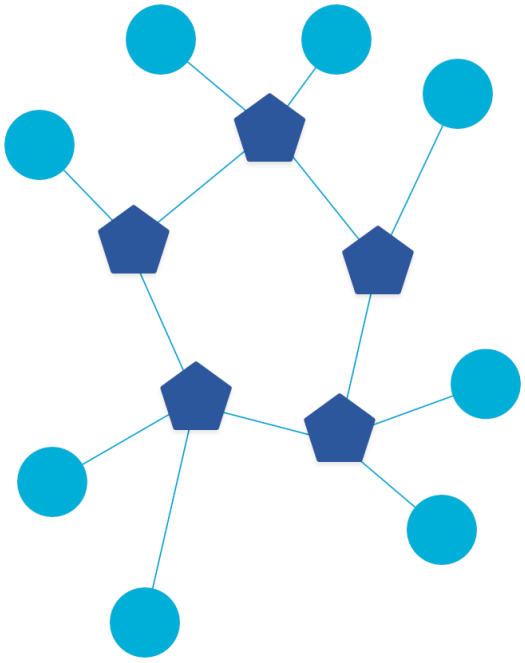
부모 기기는 하위 ID 응답(Child ID Response) 메시지를 유니캐스트로 보내 링크 설정을 확인한다. 메시지에는 Source Address(Parent RLOC16), Address16(Child RLOC16), Leader Data, Network Data, Route 정보, Timeout, Address Registration 확인 등이 포함된다. 아래 그림을 참조한다.



<그림 6> Thread 네트워크의 Child ID Response 단계

2.5 라우터 탐색

Thread 네트워크에서는 모든 라우터가 연결 지배 집합(Connected Dominating Set, CDS)을 형성해야 한다. 즉, 네트워크 내의 어떤 두 라우터도 라우터 전용 경로를 통해 반드시 연결되어 있어야 하며, CDS 내의 모든 라우터가 서로 도달 가능한 상태를 유지해야 한다. 또한, 최종 기기(End Device)는 반드시 직접 라우터에 연결되어야 하며, 분산 알고리즘은 이러한 조건을 만족하면서도 최소 수준의 중복만을 허용하도록 CDS 를 동적으로 관리한다. 네트워크 상태가 변할 때마다 알고리즘은 라우터 수가 16 개 미만으로 떨어지면 범위를 확장하고 경로 다양성을 높이기 위해 새 라우터를 추가하며, 반대로 라우터 수가 32 개를 넘으면 라우팅 상태를 줄이고 다른 구역에 새 라우터를 배치할 수 있도록 기존 라우터를 삭제한다.

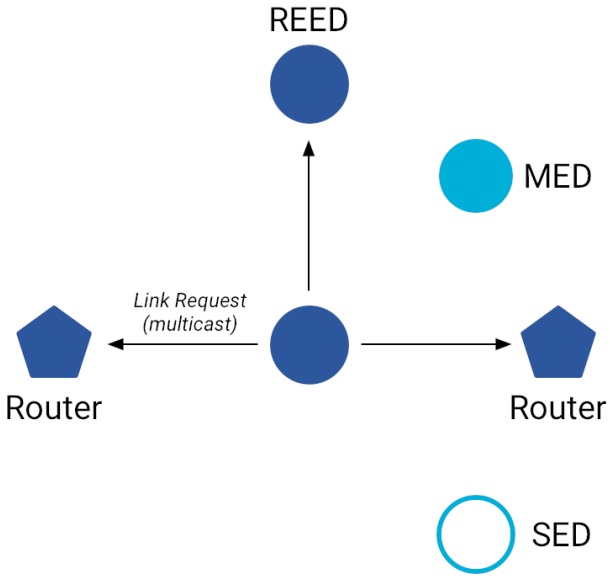


<그림 7> 연결된 지배 집합의 예

한편, Thread 네트워크에 연결된 최종 기기는 필요에 따라 라우터로 업그레이드될 수 있다. 기기가 라우터가 되기로 결정하면 먼저 리더에게 주소 요청 메시지를 보내 라우터 ID 를 요청하고, 리더가 이를 승인하면 할당받은 ID 로 라우터로 전환된다. 이어서 MLE(Mesh Link Establishment) 프로토콜을 통해 인접한 라우터들과 양방향 링크를 설정한다. 구체적으로 새 라우터는 멀티캐스트 방식으로 링크 요청 메시지를 전송하고, 인접 라우터들은 링크 수락 및 요청 메시지로 응답한다. 마지막으로 새 라우터가 각 라우터로부터 받은 요청에 대해 유니캐스트 링크 수락을 보내면, 양쪽 모두에서 완전한 라우터–라우터 연결이 수립된다.

2.5.1 연결 요청

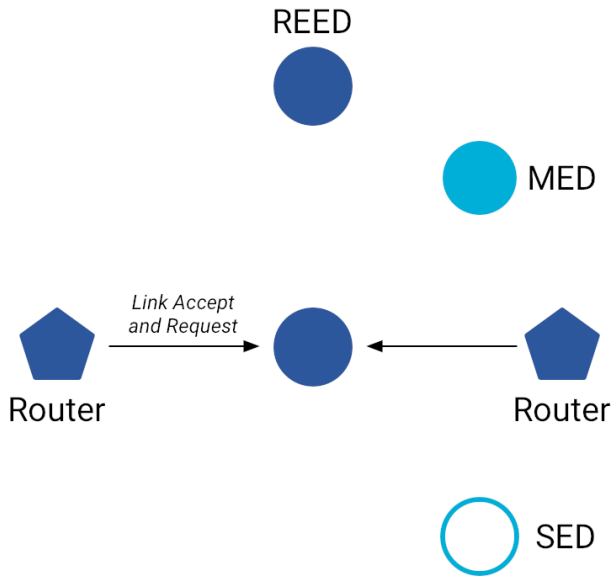
연결 요청은 라우터가 스레드 네트워크의 다른 모든 라우터에 보내는 요청이다. 처름 라우터가 되면 기기는 ff02::2 에 멀티캐스트 링크 요청을 보낸다. 나중에 MLE 광고를 통해 다른 라우터를 발견한 후 기기는 unicast 링크 요청을 전송한다.



<그림 8> Thread 네트워크 Link Request 단계

2.5.2 연결 수락 및 요청

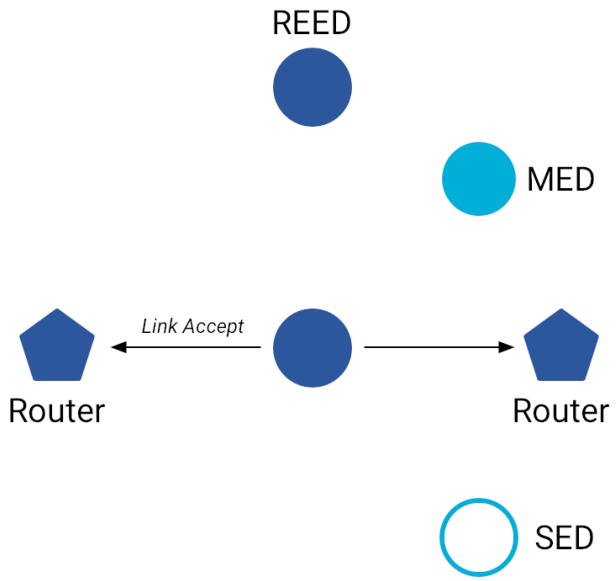
연결 수락 및 요청은 연결 수락 및 연결 요청 메시지의 조합이다. 스레드는 MLE 링크 요청 프로세스에서 이 최적화를 사용하여 메시지 수를 4 개에서 3 개로 줄인다.



<그림 9> Thread 네트워크 Link Accept and Request 단계

2.5.3 연결 수락

Link Accept 는 인접 라우터의 Link Request 에 대한 unicast 응답으로, 자체에 관한 정보를 제공하고 인접 라우터의 링크를 수락합니다.



<그림 10> Thread 네트워크 Link Accept 단계

라우터가 REED 로 다운그레이드되면 기존의 라우터-라우터 링크가 해제되고, 기기는 MLE Attach 프로세스를 시작하여 자식-부모 링크를 재설정한다. 이 과정의 세부 절차는 "기존 네트워크에 참여(Joining an Existing Network)" 항목을 참조하면 된다.

한편, 경우에 따라 단방향 수신 링크를 설정해야 할 수도 있다. 예컨대, 라우터 재설정 후에도 인접한 라우터가 여전히 유효한 수신 링크를 유지하고 있는 경우, 재설정된 라우터는 Link Request 메시지를 전송하여 라우터-라우터 연결을 복원한다. 또한 멀티캐스트 전송의 안정성을 높이기 위해 엔드 디바이스가 자신의 상위 라우터가 아닌 인접 라우터와 수신 링크를 맺는 경우도 있으며, 멀티캐스트 라우팅에 대한 자세한 내용은 이후 섹션에서 다룬다.

3. Open Thread 에 대하여

3.1 Open Thread 란?

Nest 에서 출시한 OpenThread 는 Thread 네트워킹 프로토콜의 오픈 소스 구현이다. Nest 는 커넥티드 홈용 제품 개발을 가속화하기 위해 Nest 제품에 사용되는 기술을 개발자가 광범위하게 사용할 수 있도록 OpenThread 를 출시했다.

스레드 사양은 홈 애플리케이션을 위한 IPv6 기반의 안정적이고 안전하며 저전력 무선 기기 간 통신 프로토콜을 정의한다. OpenThread 은 IPv6, 6LoWPAN, MAC 보안이 적용된 IEEE 802.15.4, 메시 링크 설정, 메시 라우팅을 비롯한 모든 Thread 네트워킹 레이어를 구현한다.

3.2 Docker 를 사용한 시뮬레이션

3.2.1 필요한 항목

Docker, Linux, 네트워크 라우팅에 관한 기본 지식

3.2.2 Docker 설정

Linux, MAC OC X 또는 Windows 시스템에서 Docker 를 사용하도록 설계되었다. 권장 환경은 Linux 이다.

3.2.3 Docker 이미지 가져오기

Docker 가 설치되면 터미널 창을 열고 openthread/environment Docker 이미지를 가져온다. 이 이미지에는 이 Codelab 에서 사용할 수 있는 사전 빌드된 OpenThread 및 Open Thread 데몬이 포함되어 있다.

```
$ docker pull openthread/environment:latest
```

터미널 창의 이미지에서 Docker 컨테이너를 시작하고 bash 셸에 연결한다.

```
$ docker run --name codelab_otsim_ctnr -it --rm ₩
  --sysctl net.ipv6.conf.all.disable_ipv6=0 ₩
  --cap-add=net_admin openthread/environment bash
```

--rm 옵션은 컨테이너를 종료할 때 컨테이너를 삭제한다. 컨테이너를 삭제 않으려면 이 옵션을 사용하지 않는다.

--sysctl net.ipv6.conf.all.disable_ipv6=0

- 컨테이너 내에서 IPv6 를 사용 설정한다.

--cap-add=net_admin

- IP 경로 추가와 같은 네트워크 관련 작업을 실행할 수 있는 NET_ADMIN 기능을 사용 설정한다.

컨테이너에 들어가면 다음과 비슷한 프롬프트가 표시된다.

root@c0f3912a74ff:/#

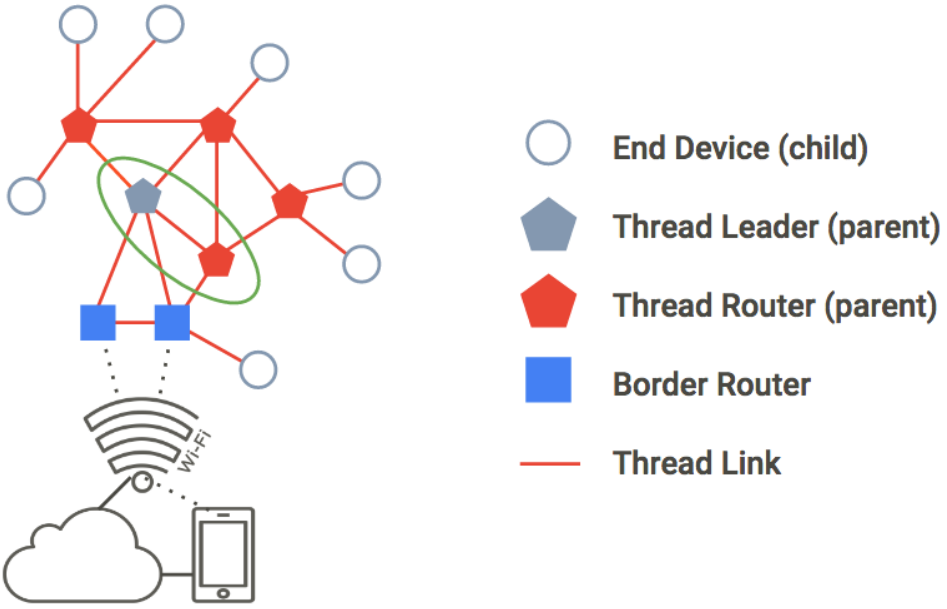
위의 예에서 c0f3912a74ff 는 컨테이너 ID 입니다. Docker 컨테이너 인스턴스의 컨테이너 ID 는 이 Codelab 의 메시지에 표시된 것과 다릅니다.

3.2.4 스레드 네트워크 시뮬레이션

사용할 예시 애플리케이션은 기본 명령줄 인터페이스 (CLI)를 통해 OpenThread 구성 및 관리 인터페이스를 노출하는 최소한의 OpenThread 애플리케이션을 보여준다.

이 연습에서는 에뮬레이션된 다른 스레드 기기에서 에뮬레이션된 스레드 기기를 핑하는 데 필요한 최소한의 단계를 안내한다.

아래 그림은 기본적인 스레드 네트워크 토폴로지를 보여준다. 이 연습에서는 녹색 원 안에 있는 두 개의 노드, 즉 스레드 리더와 스레드 라우터를 에뮬레이션하고, 두 노드 사이에 연결이 하나 있도록 설정한다.



3.2.5 Docker 설정

3.2.5.1 노드 1 시작

아직 하지 않았다면 터미널 창에서 Docker 컨테이너를 시작하고 bash 셸에 연결한다.

```
$ docker run --name codelab_otsim_ctnr -it --rm ₩
  --sysctl net.ipv6.conf.all.disable_ipv6=0 ₩
  --cap-add=net_admin openthread/environment bash
```

Docker 컨테이너에서 ot-cli-ftd 바이너리를 사용하여 에뮬레이션된 스레드 기기의 CLI 프로세스를 생성한다.

```
root@c0f3912a74ff:/# /openthread/build/examples/apps/cli/ot-cli-ftd 1
```

이 바이너리는 OpenThread 기기를 구현한다. IEEE 802.15.4 무선 드라이버는 UDP 위에 구현된다. (IEEE 802.15.4 프레임은 UDP 페이로드 내에서 전달됨).

1의 인수는 '팩토리 할당'의 최하위 비트를 나타내는 파일 설명자이다. 에뮬레이션된 기기의 IEEE EUI-64 이 값은 IEEE 802.15.4 무선 에뮬레이션 (포트 = 9000 + 파일 설명자)을 위해 UDP 포트에 바인딩할 때도 사용된다. 이 Codelab 에서 에뮬레이션된 스레드 기기의 각 인스턴스는 다른 파일 설명자를 사용한다.

새 운영 데이터 세트를 만들고 활성 데이터 세트로 커밋한다. 운영 데이터 세트는 생성 중인 스레드 네트워크의 구성이다.

```
> dataset init new
Done
> dataset
Active Timestamp: 1
Channel: 20
Channel Mask: 07fff800
Ext PAN ID: d6263b6d857647da
Mesh Local Prefix: fd61:2344:9a52:ede0/64
Network Key: e4344ca17d1dca2a33f064992f31f786
Network Name: OpenThread-c169
PAN ID: 0xc169
PSKc: ebb4f2f8a68026fc55bcf3d7be3e6fe4
Security Policy: 0, onrcb
Done
```

이 데이터 세트를 활성 데이터 세트로 커밋한다.

```
> dataset commit active
Done
```

IPv6 인터페이스를 불러온다.

```
> ifconfig up
Done
```

스레드 프로토콜 작업을 시작한다.

```
> thread start
Done
```

몇 초 동안 기다렸다가 기기가 스레드 리더가 되었는지 확인한다. 리더는 라우터 ID 할당을 관리하는 기기이다.

```
> state
leader
Done
```

노드 1 의 스레드 인터페이스에 할당된 IPv6 주소를 확인한다. (출력은 다름).

```
> ipaddr
fd61:2344:9a52:ede0:0:ff:fe00:fc00
fd61:2344:9a52:ede0:0:ff:fe00:5000
fd61:2344:9a52:ede0:d041:c5ba:a7bc:5ce6
fe80:0:0:0:94da:92ea:1353:4f3b
Done
```

특정 IPv6 주소 유형을 확인한다.

fd 로 시작 = Mesh-local

fe80(으)로 시작 = link-local

메시-로컬 주소 유형은 다음과 같이 추가로 분류된다..

ff:fe00 포함 = 라우터 로케이터 (RLOC)

ff:fe0 = 엔드포인트 식별자 (EID)를 포함하지 않음

콘솔 출력에서 EID 를 식별하여 나중에 사용할 수 있도록 기록해둔다. 위의 샘플 출력에서 EID 는 다음과 같다.

```
fd61:2344:9a52:ede0:d041:c5ba:a7bc:5ce6
```

3.2.5.2 노드 2 시작

새 터미널을 열고 현재 실행 중인 Docker 컨테이너에서 노드 2 에 사용할 bash 셸을 실행한다.

```
$ docker exec -it codelab_otsim_ctnr bash
```

새 bash 프롬프트에서 2 인수를 사용하여 CLI 프로세스를 생성한다. 다음은 두 번째로 에뮬레이션된 스레드 기기다.

```
root@c0f3912a74ff:/# /openthread/build/examples/apps/cli/ot-cli-ftd 2
```

노드 1 의 운영 데이터 세트와 동일한 값을 사용하여 스레드 네트워크 키와 PAN ID 를 구성한다.

```
> dataset networkkey e4344ca17d1dca2a33f064992f31f786
Done
> dataset panid 0xc169
Done
```

이 데이터 세트를 활성 데이터 세트로 커밋한다.

```
> dataset commit active
Done
```

IPv6 인터페이스를 불러온다.

```
> ifconfig up
Done
```

스레드 프로토콜 작업을 시작한다.

```
> thread start
Done
```

기기가 자체적으로 하위 요소로 초기화된다. 스레드 하위 요소는 상위 기기와만 유니캐스트 트래픽을 전송하고 수신하는 스레드 기기인 최종 기기와 동일한다.

```
> state
child
Done
```

2 분 이내에 상태가 child 에서 router 로 전환되는 것을 확인할 수 있다. 스레드 라우터는 스레드 기기 간에 트래픽을 라우팅할 수 있다. 상위 관리자라고도 한다.

```
> state
router
Done
```

3.2.6 네트워크 확인

메시 네트워크를 쉽게 확인할 수 있는 방법은 라우터 테이블을 확인하는 것이다.

3.2.6.1 연결 확인

Node 2 에서 RLOC16 을 가져온다. RLOC16 은 기기 RLOC IPv6 주소의 마지막 16 비트이다.

```
> rloc16
5800
Done
```

노드 1 에서 라우터 테이블에서 노드 2 의 RLOC16 을 확인한다. 먼저 노드 2 가 라우터 상태로 전환되었는지 확인한다.

```
> router table
| ID | RLOC16 | Next Hop | Path Cost | LQ In  | LQ Out  | Age | Extended MAC    |
+----+--------+----------+-----------+--------+-------+---+------------------+
| 20 | 0x5000 |       63 |         0 |      0 |     0 |   0 | 96da92ea13534f3b |
| 22 | 0x5800 |       63 |         0 |      3 |     3 |  23 | 5a4eb647eb6bc66c |
```

노드 2 의 RLOC 0x5800 가 테이블에 있으므로 메시에 연결되었는지 확인할 수 있습니다.

3.2.6.2 노드 2 에서 노드 1 핑

에뮬레이션된 두 스레드 기기 간의 연결을 확인한다. 노드 2 에서 ping 는 노드 1 에 할당된 EID 이다.

```
> ping fd61:2344:9a52:ede0:d041:c5ba:a7bc:5ce6
```

```
> 16 bytes from fd61:2344:9a52:ede0:d041:c5ba:a7bc:5ce6: icmp_seq=1 hlim=64 time=12ms
```

enter 를 눌러 > CLI 프롬프트로 돌아갑니다.

3.2.7 네트워크 테스트

이제 에뮬레이션된 두 스레드 기기 간에 핑을 성공적으로 마칠 수 있으므로 노드 하나를 오프라인으로 전환하여 메시 네트워크를 테스트한다.

노드 1 로 돌아가서 스레드를 중지한다.

```
> thread stop
Done
```

노드 2 로 전환하고 상태를 확인한다. 2 분 이내에 노드 2 는 리더 (노드 1)가 오프라인 상태임을 감지하고 노드 2 가 네트워크의 leader 로 전환되는 것을 확인할 수 있다.

```
> state
router
Done
...
> state
leader
Done
```

확인이 완료되면 Docker bash 프롬프트로 다시 종료하기 전에 스레드를 중지하고 Node 2 를 초기화한다. 이 실습에서 사용한 스레드 네트워크 사용자 인증 정보가 다음 실습으로 이전되지 않도록 초기화가 수행된다.

```
> thread stop
Done
> factoryreset
>
> exit
root@c0f3912a74ff:/#
```

factoryreset 명령어 다음에 > 프롬프트를 다시 표시하려면 enter 를 몇 번 눌러야 할 수 있다. Docker 컨테이너를 종료하면 안된다.

또한 초기화하고 Node 1 을 종료합니다.

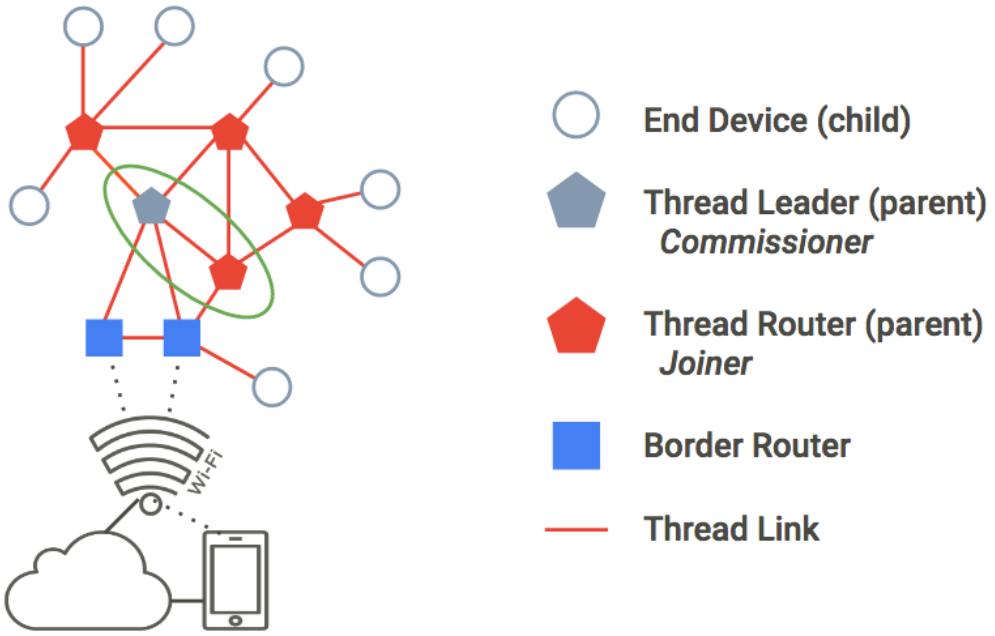
```
> factoryreset
>
> exit
root@c0f3912a74ff:/#
```

3.2.8 커니셔닝으로 노드 인증

이전 연습에서는 두 개의 시뮬레이션된 기기와 확인된 연결을 사용하여 스레드 네트워크를 설정했다. 하지만 이 경우 인증되지 않은 IPv6 링크-로컬 트래픽만 기기 간에 전달될 수 있다. 노드 간 전역 IPv6 트래픽 (및 스레드 보더 라우터를 통한 인터넷)을 라우팅하려면 노드를 인증해야 한다.

인증하려면 하나의 기기가 커미셔닝 역할을 해야 한다. 커미셔너는 현재 새로운 스레드 디바이스에 대해 선택된 인증 서버이며, 디바이스가 네트워크에 참여하는 데 필요한 네트워크 크리덴셜을 제공하는 승인자이다.

이 연습에서는 이전과 동일한 2 노드 토폴로지를 사용한다. 인증의 경우 스레드 리더가 커미셔너 역할을 하고 스레드 라우터는 조인자 역할을 한다.



3.2.9 커미셔닝 실행자 역할 시작하기

노드 1 에서 커미셔닝 역학을 시작한다.

```
> commissioner start
```

```
Done
```

J01NME Joiner 사용자 인증 정보와 함께 모든 Joiner (와일드 카드 사용)가 네트워크에 커미셔닝하도록 허용한다. Joiner 는 관리자가 의뢰한 스레드 네트워크에 추가하는 기기이다.

```
> commissioner joiner add * J01NME
```

```
Done
```

3.2.10 연결자 역할 시작

두 번째 터미널 창의 Docker 컨테이너에서 새 CLI 프로세스를 생성한다. Node 2 이다.

```
root@c0f3912a74ff:/# /openthread/build/examples/apps/cli/ot-cli-ftd 2
```

노드 2 에서 J01NME Joiner 사용자 인증 정보를 사용하여 Joiner 역할을 사용 설정합니다.

```
> ifconfig up

Done

> joiner start J01NME

Done
Join success
```

Joiner 로서 기기 (노드 2)는 커미셔닝원 (노드 1)을 통해 자체 인증을 성공적으로 수행하고 스레드 네트워크 사용자 인증 정보를 수신했다.

이제 노드 2 가 인증되었으므로 스레드를 시작한다.

```
> thread start

Done
```

3.2.11 네트워크 인증 확인

노드 2 의 state 를 확인하여 네트워크에 연결되었는지 확인합니다. 2 분 내에 노드 2 가 child 에서 router 로 전환됩니다.

```
> state

child

Done
```

```
...
> state
router
Done
```

4. 결론

본 보고서에서는 IoT 시대에 적합한 메쉬 네트워크 프로토콜인 Thread 의 개념과 그 구성 요소들에 대해 심층적으로 다루었다. 먼저 Thread 의 기본 정의와 구조를 통해, 기존 무선 네트워크와 차별화되는 점, 즉 자체 복구(self-healing) 기능, 저전력 설계, 그리고 보안 중심의 구조를 확인할 수 있었다. 특히 Thread 네트워크에서의 노드 역할과 유형(라우터, 엔드 디바이스, 리더 등)에 따라 네트워크 내 기능이 분산되고, 이를 기반으로 보다 유연한 네트워크 구성이 가능하다는 점이 강조되었다.

IPv6 기반 주소 지정 방식은 Thread 가 기존 인터넷 인프라와의 높은 호환성을 유지하면서도 확장 가능한 네트워크 환경을 구성할 수 있는 핵심적인 요소이며, 노드 간 통신 및 라우팅 효율성 향상에도 크게 기여한다. 아울러 네트워크 탐색 및 형성과정(부모 요청/응답, 하위 ID 요청 등)을 통해 각 노드가 어떻게 네트워크에 참여하고 역할을 부여받는지를 상세히 분석하였다. 이 과정에서 Thread 의 네트워크 자율 구성 능력과 안정적인 운영 구조를 확인할 수 있었다.

또한, Thread 네트워크 내 라우터 간의 연결 절차(연결 요청, 수락 등)를 살펴보며, Thread 가 지닌 메쉬 네트워크의 강점을 더욱 구체적으로 이해할 수 있었다. 이는 특히 대규모 IoT 환경에서 노드 간의 연결성을 유지하고 통신 장애를 최소화하는 데 중요한 기제로 작용한다.

마지막으로, OpenThread 를 활용한 Docker 기반 시뮬레이션 환경 구축 방법을 소개함으로써, 실제 하드웨어 없이도 Thread 네트워크를 가상으로 구성하고 테스트할 수 있는 방법론을 제시하였다. 이는 개발 초기 단계에서의 빠른 프로토타이핑, 기능 검증, 오류 추적 등에 매우 효과적으로 활용될 수 있다.

앞으로 스마트홈, 헬스케어, 산업 자동화, 스마트시티와 같은 다양한 분야에서 Thread 기반의

네트워크 활용 가능성은 매우 클 것으로 전망된다. 본 보고서가 Thread 의 원리와 구조를 이해하고, 실무에 적용하기 위한 기초 자료로서 활용되기를 기대한다. 나아가 OpenThread 와 같은 오픈소스 플랫폼을 통해 실제 개발 환경에 적극적으로 도입하고 확장할 수 있는 기반을 마련하는 데 도움이 되기를 바란다.

참고 문헌

[1] OpenThead, https://openthread.io/?hl=ko