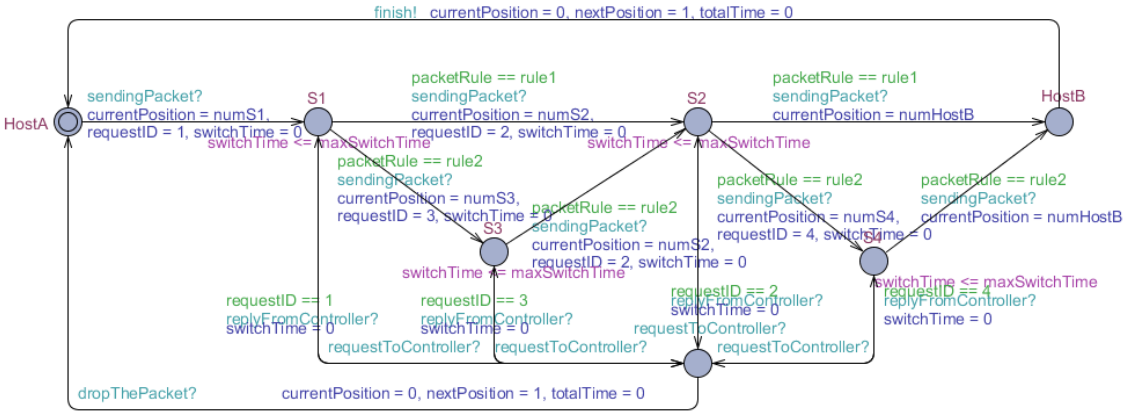
Modeling and Verification for SDN by UPPAAL

목 차

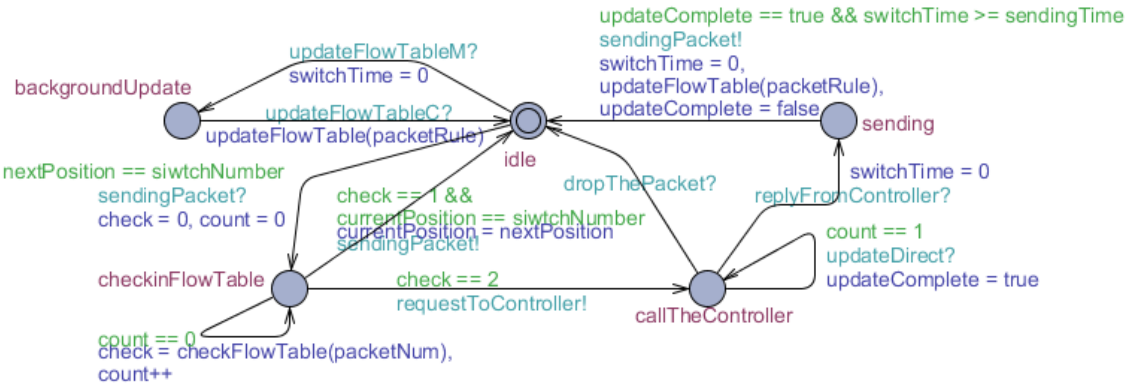
1. 모델 설명

<모델1 네트워크 토폴로지>



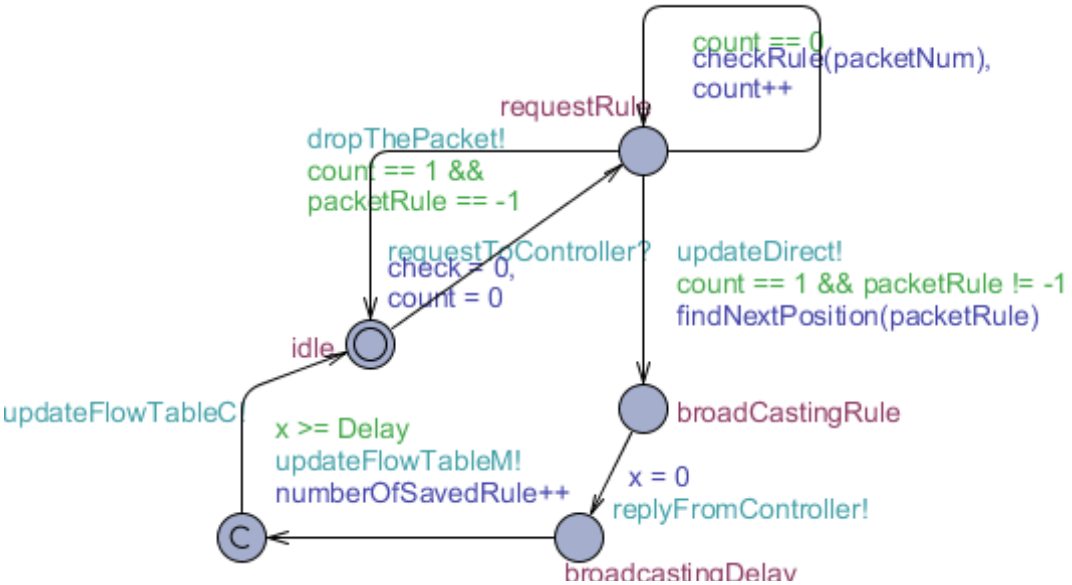
이 모델은 패킷이 전송되면서 거치는 스위치들을 직접 확인할 수 있도록 해준다. 또한 네트워크 전반의 정보를 가지고 있어, 속성 검증 또한 이 모델을 통해 진행한다. currentPosition 변수는 현재의 패킷 위치를 저장하고 있고, packetRule은 패킷이 어떠한 룰을 따르는지를 저장한다. 이 값은 컨트롤러에 의해 변경되거나, 룰에 대한 정보를 가지고 있는 스위치에 의해 변경된다. nextPosition은 패킷이 어떤 룰을 따르냐에 따라 다음 경로가 달라지므로, 룰에 따라 다음 이동할 스위치에 대한 정보를 저장한다. 각 스위치는 패킷을 maxSwitchTime이라는 시간 이내에 처리해야하며, 스위치에서 패킷이 머무르는 시간은 switchTime 변수로 측정한다.

<모델2 스위치>



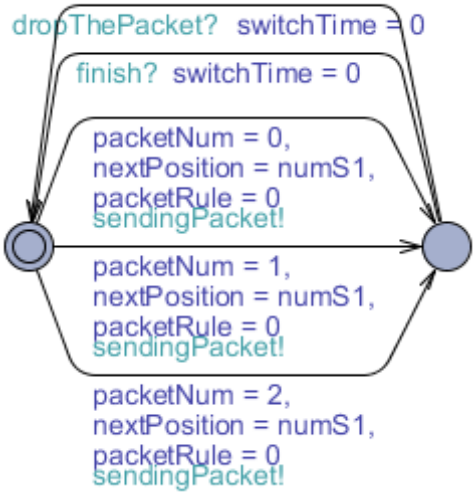
스위치 모델은 총 두 가지의 기능을 가지고 있다. 첫 번째로는 컨트롤러가 브로드캐스팅 하는 정보를 받아 플로우 테이블을 업데이트 하는 일이고, 다른 하나는 패킷을 전송하는 일이다. check 변수는 스위치가 패킷을 받았을 때, 드랍을 시킬 것인지 다음 스위치를 찾아 전송할 것인지를 결정하는 변수이다. sendingTime은 패킷 전송 시 걸리는 시간에 대한 변수이며, 이 값은 패킷이 스위치에 머무를 수 있는 최대 시간보다는 작거나 같아야 한다.

<모델3 컨트롤러>



컨트롤러의 동작을 정의하였다. broadcastingDelay 상태에서 브로드캐스팅 하기 전에 Delay의 값만큼 대기함으로써 물리적인 거리로 인해 컨트롤러와 스위치 간의 딜레이를 표현하였다. requestRule 상태에서 이 패킷이 드랍이 될 것인지, 아니면 룰이 적용되어 전송될 것인지 판단된다.

<모델4 HostA> - trigger



HostA에 연결된 스위치는 S1 밖에 없으므로, HostA는 nextPosition이 S1일 수 밖에 없다. 따라서 패킷 헤더를 랜덤으로 생성할 때, nextPosition을 S1의 ID로 초기화하여 패킷을 생성한다. 패킷 전송이 완료되면 finish 채널을 받고, 패킷이 드랍될 경우, dropThePacket 채널로 그 사실을 알 수 있다.

2. 모델 시나리오

1) 정상적으로 작동하기 위한 초기화를 했을 시,

Delay값을 1로 주고, sendingTime의 값을 2로 초기화해둔다. 스위치별 패킷을 처리하기 위해 대기하는 최대 시간인 maxSwitchTime의 값은 3으로 초기화한다. currentPosition은 현재 패킷의 위치를 나타내며, nextPosition은 패킷이 룰을 적용하였을 때, 이동해야할 다음 스위치의 위치를 의미한다. 또한 각 위치별 ID를 부여하였다. HostA는 0, S1은 1, S2는 2, S3은 3, S4는 4 그리고 HostB는 5의 값을 가진다.

HostA가 패킷헤더가 0인 패킷을 생성하고, S1에게 패킷을 보낸다고 신호를 보낸다. 이때, nextPosition의 값은 S1의 ID 값으로 설정한다. 패킷을 보낸다는 신호를 받은 S1은 nextPosition이 자신의 스위치 ID와 같은지 확인한다. 같은 경우, 패킷을 받고, currentPosition의 값을 자신의 ID 값인 1로 업데이트한다. checkFlowTable함수를 이용하여 플로우 테이블을 검사한다. 현재는 플로우 테이블이 비어있는 상태이므로, checkFlowTable 함수는 2값을 반환하여 check 변수에 2라는 값을 대입한다. 그로 인해 해당 패킷에 대한 룰이 플로우 테이블에 없다는 사실을 알게 되어 컨트롤러에게 룰을 요청하게 된다. 요청을 받은 컨트롤러는 checkRule 함수를 이용하여 해당 패킷이 어떤 룰을 적용해야하는지 판단한다. 패킷 헤더가 0은 룰1을 적용하므로, findNextPosition 함수를 이용하여 현재 위치에 따른 다음 위치를 찾아 nextPosition 변수에 저장해둔다. 그 후, 요청한 스위치에게 reply를 보내고, broadcastingDelay 상태에서 Delay 값만큼 대기하게 된다. 현재 시스템에서는 Delay값을 1로 설정하였으므로, 1만큼 대기한다. 컨트롤러의 reply를 받은 S1은 updateComplete 값을 true로 바꾼 후, 패킷을 보내기 위한 다음 상태인 sending 상태로 넘어간다. 패킷을 전송하는데 걸리는 시간을 측정하기 위해 switchTime을 0으로 초기화한다. 초기화해놨던 sendingTime이 2이므로, 2의 시간만큼 대기한다. 컨트롤러의 대기시간인 1이 지난 후, 컨트롤러는 해당 패킷에 대한 룰을 연결되어 있는 모든 스위치들에게 브로드캐스팅한다. 브로드캐스팅을 받은 S2, S3, S4는 내부의 updateFlowTable 함수로 플로우 테이블을 업데이트 하여, 패킷 헤더 0은 룰 1을 적용한다는 사실을 인지하게 된다. sendingTime인 2가 모두 지나면, S1은 nextPosition의 값을 보고 해당 스위치에게 패킷을 보낸다. 룰1을 적용했을 때, nextPosition은 S2이므로, S2는 자신의 ID와 nextPosition의 값이 같은 지 확인하고, 같으면 해당 패킷을 받으면서 currentPosition의 값을 자신의 ID인 2로 업데이트한다. 플로우 테이블을 확인하기 위한 함수인 checkFlowTable로 플로우 테이블을 확인한다. 해당 룰이 플로우 테이블에 존재하므로, 함수는 check 값에 1을 반환한다. 다음 목적지인 HostB에 패킷을 보내고, currentPosition을 HostB의 ID인 5로 업데이트한다. HostB가 패킷을 받으면, HostA에 패킷을 잘받았다는 의미로 finish라는 신호를 보내며, currentPosition과 nextPosition을 다시 초기화한다.

HostA가 다시 패킷 헤더가 0번인 패킷을 생성할 시, nextPosition의 값을 보고, S1으로 패킷을 보내고, S1은 플로우 테이블에서 검사하여 다음 스위치가 S2라는 사실을 인지하고 바로 패킷을 전송한다. S2 역시 플로우 테이블을 검사한 후, HostB에 패킷을 전송하고, HostB는 HostA에 finish 신호를 보내며, 앞으로 계속 패킷 헤더가 0번인 패킷은 플로우 테

이블 검사 후, 컨트롤러에 별다른 요청없이 HostA -> S1 -> S2 -> HostB의 순서로 패킷을 전송한다.

HostA가 패킷헤더가 1인 패킷을 생성하고, S1에게 패킷을 보낸다고 신호를 보낸다. 이때, nextPosition의 값은 S1의 ID 값으로 설정한다. 패킷을 보낸다는 신호를 받은 S1은 nextPosition이 자신의 스위치 ID와 같은지 확인한다. 같은 경우, 패킷을 받고, currentPosition의 값을 자신의 ID 값인 1로 업데이트한다. checkFlowTable함수를 이용하여 플로우 테이블을 검사한다. 플로우 테이블에는 패킷 헤더가 0인 룰1에 대한 정보가 있으므로, 패킷 헤더 1에 대한 정보가 없으므로 checkFlowTable 함수는 2값을 반환하여 check 변수에 2라는 값을 대입한다. 그로 인해 해당 패킷에 대한 룰이 플로우 테이블에 없다는 사실을 알게 되어 컨트롤러에게 룰을 요청하게 된다. 요청을 받은 컨트롤러는 checkRule 함수를 이용하여 해당 패킷이 어떤 룰을 적용해야하는지 판단한다. 패킷 헤더가 1인 경우는 룰2를 적용하므로, findNextPosition 함수를 이용하여 현재 위치에 따른 다음 위치를 찾아 nextPosition 변수에 저장해둔다. 그 후, 요청한 스위치에게 reply를 보내고, broadcastingDelay 상태에서 Delay 값만큼 대기하게 된다. 현재 시스템에서는 Delay값을 1로 설정하였으므로, 1만큼 대기한다. 컨트롤러의 reply를 받은 S1은 updateComplete 값을 true로 바꾼 후, 패킷을 보내기 위한 다음 상태인 sending 상태로 넘어간다. 패킷을 전송하는데 걸리는 시간을 측정하기 위해 switchTime을 0으로 초기화한다. 초기화해놨던 sendingTime이 2이므로, 2의 시간만큼 대기한다. 컨트롤러의 대기시간인 1이 지난 후, 컨트롤러는 해당 패킷에 대한 룰을 연결되어 있는 모든 스위치들에게 브로드캐스팅한다. 브로드캐스팅을 받은 S2, S3, S4는 내부의 updateFlowTable 함수로 플로우 테이블을 업데이트하여, 패킷 헤더 2는 룰 2를 적용한다는 사실을 인지하게 된다. sendingTime인 2가 모두 지나면, S1은 nextPosition의 값을 보고 해당 스위치에게 패킷을 보낸다. 룰2를 적용했을 때, nextPosition은 S3이므로, S3은 자신의 ID와 nextPosition의 값이 같은 지 확인하고, 같으면 해당 패킷을 받으면서 currentPosition의 값을 자신의 ID인 2로 업데이트한다. 플로우 테이블을 확인하기 위한 함수인 checkFlowTable로 플로우 테이블을 확인한다. 해당 룰이 플로우 테이블에 존재하므로, 함수는 check 값에 1을 반환한다. 다음 목적지인 S2로 패킷을 보내고, currentPosition을 S2의 ID인 2로 업데이트한다. S2는 플로우테이블을 검사하여, S2에서 룰2를 적용한 패킷의 경우, 다음 위치가 S4임을 알아 nextPosition을 S4의 ID인 4로 업데이트를 한다. S2가 패킷을 S4로 전송하고, S4는 nextPosition과 S4의 ID가 일치한지 확인한 후, 일치하므로 패킷을 받는다. S4가 플로우 테이블을 확인하여, 다음 목적지를 찾는다. 다음 목적지는 HostB이므로, HostB의 ID인 5값을 nextPosition에 대입하고, HostB에 패킷을 보낸다. HostB가 패킷을 받으면, HostA에 패킷을 잘받았다는 의미로 finish라는 신호를 보내며, currentPosition과 nextPosition을 다시 초기화한다.

HostB가 다시 패킷 헤더가 1번인 패킷을 생성할 시, nextPosition의 값을 보고, S1으로 패킷을 보내고, S1은 플로우 테이블에서 검사하여 다음 스위치가 S3이라는 사실을 인지하고 바로 패킷을 전송한다. S3 역시 플로우 테이블을 검사한 후, S2로 패킷을 전송하고, S2는 플로우 테이블 검사 후 S4로 패킷을 보내게 된다. 최종적으로 S4는 플로우 테이블을 검사하여, HostB에 패킷을 전송하고, HostB는 HostA에 finish 신호를 보내며, 앞으로 계속 패킷 헤더가 1번인 패킷은 플로우 테이블 검사 후, 컨트롤러에 별다른 요청없이 HostA -> S1 -> S3 -> S2 -> S4 -> HostB의 순서로 패킷을 전송한다.

HostA가 헤더가 2인 패킷을 생성하여 S1으로 패킷을 보낼 시, S1은 nextPosition이 자신인지를 확인한 후 패킷을 받는다. 패킷을 받은 후, 플로우 테이블을 검사한다. 플로우 테이블에는 현재 패킷 헤더가 0인 룰1과 패킷 헤더가 1인 룰2에 대한 정보 밖에 없으므로, 컨트롤러에게 이 패킷에 대한 정보를 요청하게 된다. 컨틀로러는 checkRule 함수를 통해, 패킷 헤더가 2인 경우는 드랍시켜야한다는 사실을 알고 스위치에게 패킷을 드랍하라고 한다. S1은 패킷 헤더가 2인 패킷을 드랍시킨다.

[시나리오2]
1) 컨트롤러의 브로드캐스팅이 도착하기 전에, 패킷 전송이 이루어져 컨틀롤러에 해당 패킷에 대해 두 번의 요청을 하여 오류가 발생하도록 초기화를 했을 경우,

Delay값을 2로 주고, sendingTime의 값을 1로 초기화해둔다. 스위치별 패킷을 처리하기 위해 대기하는 최대 시간인 maxSwitchTime의 값은 3으로 초기화한다.

HostA가 패킷헤더가 0인 패킷을 생성하고, S1에게 패킷을 보낸다고 신호를 보낸다. 이때, nextPosition의 값은 S1의 ID 값으로 설정한다. 패킷을 보낸다는 신호를 받은 S1은 nextPosition이 자신의 스위치 ID와 같은지 확인한다. 같은 경우, 패킷을 받고, currentPosition의 값을 자신의 ID 값인 1로 업데이트한다. checkFlowTable함수를 이용하여 플로우 테이블을 검사한다. 현재는 플로우 테이블이 비어있는 상태이므로, checkFlowTable 함수는 2값을 반환하여 check 변수에 2라는 값을 대입한다. 그로 인해 해당 패킷에 대한 룰이 플로우 테이블에 없다는 사실을 알게 되어 컨트롤러에게 룰을 요청하게 된다. 요청을 받은 컨트롤러는 checkRule 함수를 이용하여 해당 패킷이 어떤 룰을 적용해야하는지 판단한다. 패킷 헤더가 0은 룰1을 적용하므로, findNextPosition 함수를 이용하여 현재 위치에 따른 다음 위치를 찾아 nextPosition 변수에 저장해둔다. 그 후, 요청한 스위치에게 reply를 보내고, broadcastingDelay 상태에서 Delay 값만큼 대기하게 된다. 현재 시스템에서는 Delay값을 2로 설정하였으므로, 2만큼 대기한다. 컨트롤러의 reply를 받은 S1은 updateComplete 값을 true로 바꾼 후, 패킷을 보내기 위한 다음 상태인 sending 상태로 넘어간다. 패킷을 전송하는데 걸리는 시간을 측정하기 위해 switchTime을 0으로 초기화한다. 초기화해놨던 sendingTime이 1이므로, 1의 시간만큼 대기한다. sendingTime이 모두 지난 후, S1은 S로 패킷을 전송한다. 패킷을 받은 S2는 플로우 테이블을 검사하여, 해당 패킷에 대한 룰이 있는지 검사를 한다. 아직 delay로 인해 룰1에 대한 컨트롤러의 브로드캐스팅이 도달하지 못하였으므로, S2의 플로우 테이블은 비어 있는 상태이다. 따라서 S2는 컨트롤러에 해당 패킷에 대한 룰을 요청하게 된다. 컨트롤러는 연속적으로 같은 패킷의 요청으로 인해 deadlock에 빠져버린다.

3. 검증 및 쿼리

[검증 결과]

No loops : 루프가 발생하도록 하는 룰이 존재하지 않아야 한다.

* A[] Topology.totalTime <= 15

>> 패킷이 이동할 수 있는 가장 큰 경로를 고려했을 때, 최대 15 만큼이 소요된다. 즉, 패킷이 생성된 이후로 전체 시간이 15를 넘는 경우가 발생하지 않는다면, 루프가 없다고 판단할 수 있다.

Egress : 토폴로지 내의 모든 룰에 대하여, 룰을 따르는 패킷은 반드시 제한시간 내에 목적지인 HostB에 올바르게 도착해야만 한다.

* A<> Topology.HostB imply (packetRule == 1 or packetRule == 2) imply ((packetRule == 1 imply Topology.totalTime <= 9) or (packetRule == 2 imply Topology.totalTime <= 15))

>>HostB에 도착했다면, 룰1이거나 룰2이다. 룰1이면 totalTime 9 이내에, 룰2라면 totalTime 15 이내에 도착한 것이다. 따라서 Egress 속성은 위와 같은 쿼리문으로 검증할 수 있다.

Waypointing : 룰 2를 따르는 패킷은 반드시 S3과 S4를 경유하여야 한다.

* A[] (Topology.S3 imply packetRule == 2) and (Topology.S4 imply packetRule == 2)

>>패킷이 스위치 S3에 도착했다면, 이는 반드시 룰2를 따라야한다. 그리고 스위치 S4에 도착했을 때도 마찬가지로 룰2를 따르는 것을 의미하며 예외는 존재하지 않는다.

NoTimeDelay : 컨트롤러가 룰을 브로드캐스팅 시, 물리적인 거리 차로 인한 딜레이를 고려하여야 한다.

*A[] (Topology.S3 imply packetRule == 2) and (Topology.S4 imply packetRule == 2) and Delay < sendingTime

>> 모든 속성에 위와 같은 문장을 추가한 이유는 간단하다. 컨트롤러에서 스위치들에 브로드캐스팅 할 때의 딜레이가 패킷이 전송되는 시간보다 크다면, 미처 컨트롤러의 브로드캐스팅을 받지 못한 스위치는 해당 패킷에 대해 다시 룰을 요청할 수 있다. 컨트롤러는 이미 해당 룰에 대해 브로드캐스팅을 했으므로 다시 요청을 받는 것이 정의되어 있지 않다. 따라서 컨트롤러는 요청을 처리하지 못하고, 데드락 상태에 빠지게 된다. 따라서 NoLoop, Egress, Waypointing 속성은 'Delay < sendingTime'을 만족했을 시에만 검증이 된 것이다. Delay>=sendingTime인 경우는 위 속성에 대해 신뢰성을 보장하지 못한다.

1) sendingTime > Delay : NoLoop, Egress, Waypointing 모두 만족한다.
2) sendingTime <= Delay : 딜레이로 인해 컨트롤러의 룰 정보를 미처 받지 못한 스위치가 컨트롤러에게 룰을 요청할 시, deadlock이 걸려버려 위에서 정의한 모든 속성을 만족시키지 못한다.