13. Distributed Algorithms

이 장에서는 **Maude**를 사용해 여러 교과서 수준의 **분산 알고리즘(distributed algorithms)**을 형식적으로 모델링하고 분석하는 방법을 다룬다.

분산 알고리즘은 여러 개의 **노드(nodes)**가 메시지 전달(message passing)을 통해 공동의 목표를 달성하는 프로토콜을 의미한다.

- **Section 13.1**: 분산 데이터베이스에서의 **2단계 커밋(two-phase commit)** 프로토콜을 모델링 및 분석하는 방법과, 노드 실패 및 복구 모델링 기법을 다룸.
- **Sections 13.2~13.4**:
  - 13.2 : 분산 상호 배제(distributed mutual exclusion)
  - 13.3 : 분산 리더 선출(distributed leader election)
  - 13.4 : 분산 합의(distributed consensus)

이러한 알고리즘들은 클라우드 컴퓨팅 및 무선 시스템의 핵심 구성요소로, 예를 들어 **Two-Phase Commit**, **Leader Election**, **Paxos Consensus** 알고리즘은 Google의 **Megastore** 인프라(Gmail, Google+, AppEngine 등)에 사용된다.

13.1 Atomicity of Distributed Transactions: Two-Phase Commit

개념 요약

- **트랜잭션(Transaction)**

  데이터베이스에서 일련의 연산을 하나의 단일 작업으로 간주한다.

  → 모든 연산이 **전부(commit)** 반영되거나, **전혀(commit 안 됨)** 반영되지 않아야 함.

  즉, **원자성(Atomicity)** 보장.

- **다중 데이터베이스 또는 복제 데이터베이스(multiple / replicated databases)**

  여러 노드에 걸친 트랜잭션의 경우, 모든 사이트가 커밋해야만 전체 트랜잭션이 유효하다.

  하나라도 실패하면 전체를 **abort** 해야 한다.

  → 이를 달성하기 위한 프로토콜이 **Two-Phase Commit (2PC)**.

예시 : 분산 트랜잭션

고급 여행사가 고객 **X**를 위해 다음 예약을 수행하는 경우:

```
reserve(X, OSL-CDG, KLM, Dec 6 to 15);
reserve(X, Ritz, Imperial Suite, Dec 6 to 15);
reserve(X, Chez M, dinner, Dec 9);
pay(X, 6000, MasterCard, 1234567891234567, 11/17, ...);
```

→ 항공, 호텔, 식당, 결제 등 여러 사이트에 걸친 **분산 트랜잭션**.

→ 일부 예약이 실패하면 전체 트랜잭션은 **abort** 되어야 한다. (**Atomic transaction principle**)

복제 데이터베이스 (Replicated Databases)

- **복제 이유**

  데이터 손실 방지 및 고가용성 확보.

  예: 은행, 세무기관, 대학 등은 단일 DB로 운영하면 위험.

- **웹 서비스의 필요성**

  검색엔진, SNS, 온라인 예약, 결제 시스템 등은 **언제 어디서나 (available anywhere, all the time)** 작동해야 하므로 복제가 필수.

- **일관성(Consistency)**

  복제본 간 데이터가 동일해야 하지만, **고가용성(availability)**, **네트워크 장애 허용성(failure tolerance)**과의 트레이드오프 존재.

  - 검색엔진/SNS : 일부 불일치 허용 가능
  - 은행/경매시스템 : 불일치 불가 → 모든 복제본에 commit되어야 함

핵심 요약

- 분산 트랜잭션 및 복제 데이터 트랜잭션 모두 **"모든 노드에서 성공(Commit All)"** 또는 **"모든 노드에서 취소(Abort All)"** 되어야 함.
- 이 원자성을 보장하는 대표적 방법이 **Two-Phase Commit Protocol**이다.

13.1.1 The Two-Phase Commit Protocol

개요

- **2PC (Two-Phase Commit)** 프로토콜은 분산된 여러 데이터베이스가 하나의 트랜잭션을 수행할 때, **모두 커밋(commit)** 하거나 **모두 취소(abort)** 되도록 보장하는 알고리즘이다.
- 즉, 트랜잭션의 **원자성(Atomicity)**을 보장하기 위한 대표적인 분산 프로토콜.

동작 구조

2PC는 하나의 노드를 **Coordinator(조정자)**로 선정하고, 나머지는 **Participants(참여자)**가 된다.

전체 과정은 두 단계(Phase)로 나뉜다.

- **Phase 1 — Prepare (준비 단계)**
  1. **Coordinator**는 모든 참여 노드에게 "commit 준비를 하라(prepare for commit)"는 메시지를 보냄.
  2. 각 **Participant**는 로컬 트랜잭션을 완료하고, 로그 기록 및 복구 정보를 저장한 후,
     - 커밋이 가능하면 **"OK" 또는 "ready to commit"** 신호를 보냄.
     - 실패 시 **"not OK" 또는 "cannot commit"** 신호를 보냄.
  3. Coordinator가 일정 시간 안에 응답을 받지 못하면 **not OK**로 간주.
- **Phase 2 — Commit (커밋 단계)**
  1. 모든 Participant가 **OK**를 보냈고, Coordinator도 **OK**라고 판단하면:
     - Coordinator는 모든 노드에 **commit 메시지**를 보냄.
     - 각 노드는 실제로 데이터베이스에 변경 사항을 적용(로그 기록 후 반영).
  2. 하나라도 **not OK**가 있으면:

- Coordinator는 **rollback(undo)** 메시지를 보내고,
- 각 노드는 트랜잭션의 변경 사항을 취소.

특징

- 트랜잭션 중에는 실제 데이터베이스가 변경되지 않으며, **모든 노드가 성공(OK)**을 보고해야만 최종적으로 물리적 커밋이 이루어진다.
- 하나라도 실패하면 전체 트랜잭션이 **Abort** 된다.

  → “All or Nothing” 원칙.

예시: 전 세계 온라인 경매

- 노르웨이와 바누아투의 두 입찰자가 동시에 같은 상품에 입찰.
- 노르웨이의 입찰이 커밋되기 전에 모든 복제본이 이를 승인해야 함.
- 그러나 바누아투 근처의 복제본이 충돌(conflict)을 감지하고 **거부(veto)** 하면,

  → **두 입찰 모두 커밋되지 않음**,

  → 결과적으로 **아무도 상품을 얻지 못함.**

요약 문장

2PC는 분산 시스템에서 트랜잭션의 원자성을 보장하기 위해 “모두 커밋하거나, 모두 취소한다(All or Nothing)”는 원칙을 구현하는 프로토콜이다.

13.1.2 Abstraction

핵심 내용

- 2PC를 분석할 때 중요한 것은 **각 데이터베이스가 갱신되었는가(업데이트 여부)**이지, **데이터의 실제 내용(content)**은 중요하지 않다.

  따라서 데이터의 구체적인 내용은 **추상화(abstraction)**하여 무시할 수 있다.

시간(Time)과 타임아웃(Timeout) 처리

- 2PC에서는 “코디네이터가 일정 시간 내에 응답을 받지 못하면 not OK로 간주한다.”

  → 이를 모델링하려면 타이머(timer)를 사용할 수 있으나, **시간 요소를 포함하면 복잡성이 증가한다.**
- 따라서 시간과 메시지 손실 감지의 세부 사항은 생략하고, **prepare for commit 메시지는 항상 응답을 받는다**고 가정한다.

  타임아웃이 발생하는 경우는 **not OK 메시지를 받은 것으로 처리**한다.

추상화의 목적

- 시간, 메시지 손실, 데이터 읽기·쓰기 같은 세부 동작을 생략함으로써 2PC 프로토콜의 핵심 동작(커밋/어보트 결정)에 집중할 수 있다.
- 즉, **시스템의 실제 구현 세부 사항이 아닌 논리적 흐름만을 모델링**한다.

한 줄 요약

2PC 분석에서는 시간과 데이터 내용을 제외하고, 각 데이터베이스가 커밋되었는지 여부만을 단순화해 모델링한다.

13.1.3 Assumptions

핵심 요약

- 2PC(2-Phase Commit) 설명에는 **암묵적인 가정**이 존재한다.
- 구체적인 가정은 다음과 같다:
  1. **통신(communication)**은 **순서가 없을 수 있음(unordered)**.
  2. **coodinator가 될 노드**는 **모든 다른 노드들을 알고 있음**.
  3. **통신은 신뢰할 수 없을 수 있음(unreliable)**.
  4. **노드는 충돌(crash)할 수 있으며, 복구(recover) 가능**.
- 실제 실행 가능한 **형식적 명세(formal specification)**는 이러한 가정들을 모두 **명시적으로 드러내어 모델링**한다.

13.1.4 Specification and Analysis of 2PC in Maude

핵심 요약

- Maude를 사용하여 **2PC를 형식적으로 정의(formally specify)**하고 **분석(analyze)**하는 방법을 제시한다.
- 분석은 단계적으로 이루어진다:
  1. **통신 및 사이트 실패가 없는 환경**에서 2PC를 명세하고 분석.
  2. **13.1.4.3**: **메시지 손실(message loss)**이 존재할 때의 2PC 분석.
  3. **13.1.4.4**: **사이트 실패(site failure)** 와 **복구(recovery)**를 모델링하는 일반적 기법 제시.

13.1.4.1 Maude Specification of 2PC Without Failures

이 절에서는 **실패가 없는 환경에서의 2단계 커밋(2PC)** 을 **Maude** 언어로 형식적으로 명세한다.

데이터베이스 노드 정의

각 노드는 다음과 같은 형태로 정의된다

```
class 2PCDB | updated : Bool, state : CommitState, veto : Bool,
              otherNodes : OidSet, coordState : CoordState .
```

- `updated` : DB가 실제로 커밋을 수행했을 경우 `true`
- `state` : 노드의 내부 상태 (`initial`, `ready`, `abort`)
- `coordState` : 코디네이터 상태 (`notCoord`, `waitFor`)
- `otherNodes` : 다른 노드들의 집합
- `veto` : 거부 응답(`notOK`)을 받았을 때 `true`

메시지 정의

```
ops prepare OK notOK abort commit : -> MsgContent [ctor] .
msg startCommit : Oid -> Msg .
```

- `startCommit` : 2PC를 시작하는 트리거
- `prepare` : 코디네이터가 모든 노드에 보냄
- `OK` / `notOK` : 노드의 응답
- `commit` / `abort` : 최종 결정 전파

단계별 동작

1. **Prepare 단계**

   코디네이터는 `startCommit`을 받으면 `prepare` 메시지를 다른 노드에 multicast하고, 응답을 기다리는 상태로 전환한다:

   ```
   rl [prepareReq]:
     startCommit(O)
     < O : 2PCDB | state : initial, otherNodes : OS >
     =>
     < O : 2PCDB | coordState : waitFor(OS) >
     multicast prepare from O to OS .
   ```

2. **노드의 응답**

   각 노드는 `prepare`를 받고 다음 중 하나로 응답한다:

   ```
   rl [ok]:
     (msg prepare from O to O')
     < O' : 2PCDB | state : initial >
     =>
     < O' : 2PCDB | state : ready >
     (msg OK from O' to O) .

   rl [notOK]:
     (msg prepare from O to O')
     < O' : 2PCDB | state : initial >
     =>
     < O' : 2PCDB | state : abort >
     (msg notOK from O' to O) .
   ```

3. **코디네이터의 자체 투표**

   ```
   rl [coordNotOk]:
     < O : 2PCDB | state : initial, coordState : waitFor(OS) >
     =>
     < O : 2PCDB | state : abort, veto : true > .

   rl [coordOk]:
   ```

```
  < O : 2PCDB | state : initial, coordState : waitFor(OS) >
  ⇒
  < O : 2PCDB | state : ready > .
```

4. **응답 수집 (recOK / recNotOK)**

   코디네이터는 모든 노드의 응답을 읽으며, `notOK` 응답이 하나라도 있으면 `veto = true`로 설정한다.

```
rl [recOK]:
  (msg OK from O' to O)
  < O : 2PCDB | coordState : waitFor(O'; OS) >
  ⇒
  < O : 2PCDB | coordState : waitFor(OS) > .

rl [recNotOK]:
  (msg notOK from O' to O)
  < O : 2PCDB | coordState : waitFor(O'; OS) >
  ⇒
  < O : 2PCDB | coordState : waitFor(OS), veto : true > .
```

   - `recOK` : 응답이 긍정적이면 다음 노드 응답을 기다림
   - `recNotOK` : 응답이 부정적이면 `veto`를 활성화시켜 전체 abort로 이어질 수 있음

5. **최종 결정 및 전파**

   모든 응답을 받은 후, 코디네이터는 결정을 내리고 모든 노드에 전송한다:

```
rl [commitAll]:
  < O : 2PCDB | coordState : waitFor(none),
    otherNodes : OS, veto : false >
  ⇒
  < O : 2PCDB | coordState : notCoord, updated : true >
  (multicast commit from O to OS) .

rl [abortAll]:
  < O : 2PCDB | coordState : waitFor(none),
    otherNodes : OS, veto : true >
  ⇒
  < O : 2PCDB | coordState : notCoord, updated : false >
  (multicast abort from O to OS) .
```

6. **노드의 최종 갱신**

   각 노드는 코디네이터의 결정을 받고 실제 DB에 반영한다:

```
rl [recAbort]:
  (msg abort from O to O')
  < O' : 2PCDB | >
  ⇒
  < O' : 2PCDB | updated : false > .
```

```
rl [recCommit]:
  (msg commit from O to O')
  < O' : 2PCDB | >
  ⇒
  < O' : 2PCDB | updated : true > .
```

13.1.4.2 Analyzing 2PC Without Message Loss

분석 목적

- 메시지 손실 규칙은 명세에 포함되어 있지 않으므로, **신뢰할 수 있는 환경(reliable setting)**에서 2PC 프로토콜을 검증한다.
- `TEST-2PC` 모듈은 다섯 개의 데이터베이스(`a`, `b`, `c`, `d`, `e`)로 구성된 초기 상태를 정의함.
- `"a"`가 코디네이터로서 `startCommit("a")`로 트랜잭션을 시작한다.

초기 설정 예시

각 노드는 다음 상태로 시작한다:

```
<a : 2PCDB | updated:false, state:initial, otherNodes:b,c,d,e, coordState:notCoord, veto:false>
```

- 모든 노드가 아직 업데이트되지 않았고(`updated:false`), 준비 상태(`initial`)이며, `"a"`가 커밋 절차를 시작한다.

Rewrite 결과 (`frew init .`)

결과:

```
"a": ready
"b": abort
"c": ready
"d": abort
"e": ready
```

- `"b"`와 `"d"`는 커밋 불가능 → 트랜잭션 중단.
- 어떤 데이터베이스도 실제로 업데이트되지 않음 (`updated:false`).

일관성 검증

- 단일 실행 결과만 보는 rewrite 대신, **검색(search) 명령**으로 "잘못된 최종 상태(bad final state)"가 존재하는지 검사한다.
- 검사 1: 일부만 업데이트된 경우

  ```
  (search [1] init =>! < O:2PCDB | updated:false >
                      < O':2PCDB | updated:true > ...)
  ```

  결과: **No solution**

  → 일부 노드만 커밋된 불일치 상태 불가능.

- 검사 2: 어떤 노드는 abort인데 다른 노드는 commit한 경우

  ```
  (search [1] init =>! < O:2PCDB | state:abort >
                      < O':2PCDB | updated:true > ...)
  ```

  결과: **No solution**

  → abort가 발생했을 때는 어떤 노드도 업데이트하지 않음.

- 검사 3: 모두 ready인데 커밋이 이루어지지 않은 경우

  ```
  (search [1] init =>! < all 5 nodes : state:ready, updated:false > ...)
  ```

  결과: **No solution**

  → 모두 ready이면 실제로 커밋이 이루어짐.

결론

- `init` 상태에서 시작했을 때, **모든 2PC 정당성 조건(atomicity, consistency, progress)**이 만족됨.
- 그러나 이는 단지 해당 초기 상태에 대한 시뮬레이션이며, **2PC의 완전한 증명(formal proof)** 은 아니다.
- 즉, **메시지 손실이 없는 환경에서는 2PC가 올바르게 동작함을 강하게 시사**하지만, 다른 초기 상태에 대한 추가 분석이 필요하다.

13.1.4.3 Analyzing 2PC wiht Unreliable Communication

개요

이번에는 **메시지 손실이 발생할 수 있는 상황**에서 2PC를 분석한다.

단, `prepare` 요청은 항상 응답을 받는다고 가정하므로, `prepare`, `OK`, `notOK` 메시지 손실은 고려하지 않는다.

새로운 모델 정의

`abort` 또는 `commit` 메시지가 손실될 수 있도록 규칙을 추가한 모듈을 정의한다:

```
crl [lose-abortCommit] : msg MC from O to O' => none
  if MC == abort or MC == commit .
```

즉, 커밋 또는 중단 메시지가 전송 도중 사라질 수 있다.

초기 테스트 결과

명령어:

```
(frew init .)
```

결과:

```
"a", "c", "e" → ready 상태
"b", "d" → abort 상태
```

→ 일관성은 유지된 듯 보인다.

불일치 상태 탐색

검색 명령어:

```
(search [1] init =>! < O : 2PCDB | updated : false >
                    < O' : 2PCDB | updated : true > C:Configuration .)
```

결과:

```
Solution 1
... ; O':Oid --> "a" ; O:Oid --> "e"
```

- `"a"` → `"e"` 로의 커밋 메시지가 손실된 경우 불일치 상태 도달 가능.

의미

- 불일치 상태가 발생할 수 있음이 확인되었다.
- 이 결과를 분석하는 이유는:
  1. 이 동작이 실제 2PC의 결함인지, 단순히 모델링 오류인지 확인하기 위해
  2. 프로토콜의 결함을 명확히 이해하기 위해서이다.

추가 분석 방법

Full Maude는 경로를 직접 보여줄 수 없으므로, **Section 10.2.4.1**의 방법을 사용해 Full Maude 모듈을 **core Maude 모듈로 변환**한 뒤, 다시 검색하여 불일치 상태로 가는 경로를 확인한다.

그 경로에 따르면:

- 모든 노드가 커밋할 준비는 되어 있었으나,
- `"a"` → `"e"` 로 가는 **commit 메시지가 손실**되어 `"e"` 노드는 **데이터베이스를 업데이트하지 않음.**

결론 요약

메시지 손실이 있는 환경에서는 2PC 프로토콜이 일관성을 잃을 수 있다.

특히 커밋 메시지가 손실되면 일부 노드가 업데이트하지 못해 불일치 상태가 발생한다.

13.1.4.4 Modeling Process Failure and Recovery

프로세스 실패의 종류

- **프로세스(Process)** = 서버, 데이터베이스 등.
- 실패는 여러 이유로 발생할 수 있음 (소프트웨어 업그레이드, 하드웨어 장애 등).
- 실패 형태는 크게 두 가지:
  1. **Omission Failure (Crash Failure)**

     → 프로세스가 **응답하지 않게 되는 경우** (일반적인 '다운' 상태)
  2. **Byzantine Failure**

     → 프로세스가 **엉뚱한 값/메시지**를 보내는 경우 (악의적이거나 오류 전파)
- 여기서는 crash failure 중심으로 다룸 → 즉, "응답 불능 상태" 모델링.

Maude에서 실패한 프로세스를 표현하는 방법

- 방법 1 : 모든 클래스에 `failed : Bool` 속성을 추가하고 `failed : false` / `failed : true` 로 상태 표현.

  → 단점: 모든 rewrite rule마다 이 속성을 명시해야 해서 번거로움.

- 방법 2 : **(실제 사용된 방법)**

  "실패한 객체 전용 클래스"를 새로 정의함.

  - 예: 정상 노드는 `2PCDB`, 실패한 노드는 `Failed2PCDB`
  - 실패 시 `2PCDB` → `Failed2PCDB` 로 객체 변환

  ```
  class Failed2PCDB | updated : Bool, state : CommitState,
                      veto : Bool, otherNodes : OidSet,
                      coordState : CoordState .
  ```

  → 실패 시에도 노드의 정보(상태, 참여자 목록 등)는 유지됨.

실패와 복구의 모델링 (Rewrite Rules)

- 실패 규칙

  노드가 언제든 실패할 수 있도록 정의 :

  ```
  rl [nodeFailure1] :
    < O : 2PCDB | updated : B, state : S, otherNodes : OS,
                  coordState : CS, veto : B2 >
  =>
    < O : Failed2PCDB | updated : B, state : S, otherNodes : OS,
                        coordState : CS, veto : B2 > .
  ```

  → 즉, 객체의 클래스만 `2PCDB` → `Failed2PCDB` 로 바뀜.

- 복구 규칙

  실패한 노드가 복구되면 다시 원래 클래스로 전환:

  ```
  rl [nodeRecovery1] :
    < O : Failed2PCDB | updated : B, state : S, otherNodes : OS,
                        coordState : CS, veto : B2 >
  =>
    < O : 2PCDB | updated : B, state : S, otherNodes : OS,
                  coordState : CS, veto : B2 > .
  ```

실패한 노드의 동작 제한

실패한 노드는 메시지를 받더라도 반응하지 않음.

단, `recover` 메시지만 예외적으로 처리.

```
rl [ignoreMsgs] :
  (msg MC from O to O') < O' : Failed2PCDB | >
```

```
⇒
 < O' : Failed2PCDB | > .
```

Fault Injection (실패 주입)

실패를 "무작위로" 일으키면 경우의 수가 너무 많아 탐색이 불가능하므로, `fail` / `recover` **메시지**를 명시적으로 넣어서 테스트 중 실패·복구 시점을 제어함.

예:

```
msg fail recover : → Msg .
```

- `fail` 메시지 → 어떤 노드가 실패함
- `recover` 메시지 → 복구함
- 메시지에 노드를 지정하지 않으면, 임의의 노드가 실패 가능

초기 상태 정의 (initWithFailures)

예시:

```
op initWithFailures : → Configuration .
eq initWithFailures = fail fail recover init .
```

→ 초기 상태에 **2번의 실패(fail fail), 1번의 복구(recover)**를 추가해서 **"2회 실패, 1회 복구" 시나리오를 자동으로 탐색**할 수 있게 함.

13.2 Distributed Mutual Exclusion

분산 상호 배제 (Distributed Mutual Exclusion)

- 개념 :
  - 여러 프로세스가 동시에 하나의 **공유 자원(shared resource)** (예: 파일, 프린터, 은행 계좌 등)에 접근하면 문제가 발생할 수 있음.
  - 따라서 **동시에 한 프로세스만** 해당 자원에 접근하도록 해야 함.
  - 이를 **상호 배제(mutual exclusion)** 라고 함.

  → 즉, 한 프로세스가 자원을 사용하는 동안 다른 프로세스는 그 자원에 접근할 수 없음.

임계 구역 (Critical Section)

- 공유 자원에 접근하는 코드 부분을 **임계 구역**이라 함.
- 예시 :

```
y := read(x);
z := y + 20;
write(x, z);
```

→ `x`가 공유 변수이므로, 이 코드 블록은 **임계 구역**에 해당함.

→ 다른 프로세스는 이 구역 실행 중에는 `x`를 변경하면 안 됨.

분산 상호 배제 알고리즘의 목표

- **메시지 통신을 통해** 여러 프로세스 간에 상호 배제를 보장.
- 보장해야 할 조건:
  1. 동시에 **하나의 프로세스만** 임계 구역을 실행할 수 있다.
  2. 어떤 프로세스든 임계 구역 진입을 원하면 **결국 진입할 수 있어야 한다.**
  3. (공정성 조건) **요청 순서대로** 임계 구역에 들어가야 한다.

프로세스의 일반 실행 순서

```
<execute outside critical section>
<request to enter critical section>  // 접근 요청
<execute in critical section>        // 자원 접근
<release critical section>           // 해제
<execute outside critical section>
```

세 가지 주요 알고리즘

1. 중앙 서버 알고리즘 (Central Server Algorithm)
   - 중앙 서버가 각 노드의 임계 구역 접근 요청을 **직접 관리**함.
   - 가장 단순하고 관리 용이하지만, 서버가 **단일 장애점(SPOF)**이 됨.
2. 토큰 링 알고리즘 (Token Ring Algorithm)
   - 중앙 서버 없이 노드들이 **논리적 링(Ring)**을 형성.
   - **토큰(Token)**이라는 권한이 링을 따라 전달되며, **토큰을 가진 노드만** 임계 구역에 진입 가능.

   단점:

   a. 특정 노드는 토큰을 오래 기다려야 할 수 있음.

   b. 아무도 임계 구역을 원하지 않아도 토큰이 계속 순환함.

   c. 임계 구역 진입 순서가 요청 순서와 다를 수 있음.

Maekawa의 투표 알고리즘 (Maekawa's Voting Algorithm)

- 서버나 토큰이 없음.
- 노드 i가 임계 구역 진입을 원하면 **자신의 투표 집합(Vi)** 내의 모든 노드에게 요청 보냄.
- Vi의 모든 노드가 허가해야 진입 가능.

조건: 모든 노드 쌍 (i, j)에 대해, Vi와 Vj는 **최소 한 개의 공통 원소**를 가져야 함.

단점:

- 중앙 서버 방식보다 메시지가 더 많음.
- 교착상태(deadlock)가 발생할 수 있음.

전제 조건

- 통신은 **신뢰할 수 있으나 순서가 보장되지 않음**.
- 노드가 **실패하지 않는** 것으로 가정.

13.2.1 Modeling the Central Server Algorithm

전체 개요

이 모델은 **한 개의 중앙 서버(Server)** 가 여러 프로세스(Node)들의 **임계 구역(Critical Section)** 접근을 관리하는 구조이다.

- 프로세스는 임계 구역에 들어가기 전, **서버에 접근 요청(requestCS)**을 보낸다.
- 서버는 현재 임계 구역이 비어 있으면 접근을 허가하고, 비어 있지 않으면 **대기열(waiting list)**에 해당 노드를 추가한다.
- 임계 구역을 마친 노드는 **releaseCS** 메시지를 보내어 서버에게 자원을 해제한다.

Node (프로세스) 정의

```
class Node | state : MutexState .
ops beforeCS waitForCS insideCS afterCS : -> MutexState [ctor] .
```

노드는 다음 4가지 상태 중 하나를 가진다:

상태	의미
`beforeCS`	임계 구역 진입 전
`waitForCS`	임계 구역 진입 요청 후 대기 중
`insideCS`	임계 구역 실행 중
`afterCS`	임계 구역 실행 완료 후

Server (중앙 서버) 정의

```
class MutexServer | nodeInCS : Bool, waiting : OidList .
```

서버는 두 속성을 가진다:

속성	의미
`nodeInCS`	현재 임계 구역에 노드가 있는지 여부 (`true/false`)
`waiting`	대기 중인 노드들의 리스트 (FIFO 순서로 처리)

메시지 유형 정의

```
ops requestCS accessGranted releaseCS : -> MsgContent [ctor] .
```

메시지	의미
`requestCS`	임계 구역 접근 요청
`accessGranted`	접근 허가
`releaseCS`	접근 종료 및 자원 반환

규칙 (Rewrite Rules) - 동작 절차

1. 프로세스가 접근 요청을 보냄

```
rl [requestAccessToCS] :
< O : Node | state : beforeCS >
⇒
< O : Node | state : waitForCS >
(msg requestCS from O to server) .
```

- 프로세스는 상태를 `waitForCS`로 바꾸고 `requestCS` 메시지를 서버로 전송한다.

2. 서버가 요청을 처리함
   - **임계 구역이 비어 있을 경우** (`nodeInCS : false`)

     ```
     rl [grantAccess] :
     (msg requestCS from O to server)
     < server : MutexServer | nodeInCS : false >
     ⇒
     < server : MutexServer | nodeInCS : true >
     (msg accessGranted from server to O) .
     ```

     → 즉시 허가 메시지를 보냄.
   - **이미 사용 중일 경우** (`nodeInCS : true`)

     ```
     rl [putInWaitQueue] :
     (msg requestCS from O to server)
     < server : MutexServer | nodeInCS : true, waiting : OL >
     ⇒
     < server : MutexServer | waiting : OL :: O > .
     ```

     → 요청 노드를 `waiting` 리스트에 추가.
3. 접근 허가를 받은 노드가 임계 구역 실행 시작

   ```
   rl [startExecutingInCS] :
   (msg accessGranted from server to O)
   < O : Node | state : waitForCS >
   ⇒
   < O : Node | state : insideCS > .
   ```
4. 실행 완료 후 서버에 자원 해제 메시지 전송

   ```
   rl [exitCS] :
   < O : Node | state : insideCS >
   ⇒
   < O : Node | state : afterCS >
   (msg releaseCS from O to server) .
   ```
5. 서버가 자원 해제 요청을 처리
   - **대기 노드가 없을 경우**

```
rl [nooneWaiting] :
(msg releaseCS from O to server)
< server : MutexServer | waiting : nil >
⇒
< server : MutexServer | nodeInCS : false > .
```

→ 임계 구역을 "비었다"(`nodeInCS : false`)로 표시.

- **대기 노드가 있을 경우**

```
rl [grantAccessToFirstWaiting] :
(msg releaseCS from O to server)
< server : MutexServer | waiting : O' :: OL >
⇒
< server : MutexServer | waiting : OL >
(msg accessGranted from server to O') .
```

→ 대기열의 **첫 번째 노드**에게 접근 허가를 보냄 (FIFO 방식).

13.2.2 Analyzing the Central Server Algorithm

초기 상태 정의 (`init(n)`)

- `init(n)`은 **노드 n개와 서버 1개**를 가지는 **초기 시스템 상태**를 정의한다.
- 각 노드는 `beforeCS` (임계 구역 진입 전) 상태로 시작한다.
- 서버는 다음 상태로 초기화된다:

```
< server : MutexServer | nodeInCS : false, waiting : nil >
```

→ 즉, 아직 어떤 노드도 임계 구역(critical section)에 들어가지 않았고, 대기 리스트도 비어 있다.

노드 생성 함수 (`generateNodes`)

- `generateNodes(n)`은 n개의 노드를 재귀적으로 생성한다.
- 각 노드는 `< node(i) : Node | state : beforeCS >` 형태로 정의된다.
- `generateNodes(0)`은 더 이상 노드가 없음을 의미하는 `none`으로 끝난다.

상호배제(Mutual Exclusion) 검증

- Maude의 `search` 명령으로 **두 개 이상의 프로세스가 동시에 임계 구역에 있는 상태**가 존재하는지 탐색한다.

```
(search [1] init(4) ⇒*
  REST:Configuration
  < O1:Oid : Node | state : insideCS >
  < O2:Oid : Node | state : insideCS > .)
```

- **결과:**

No solution

→ 즉, **두 노드가 동시에 `insideCS` 인 상태는 존재하지 않음**

→ **상호배제 성질이 만족됨**

추가 성질

- 각 노드는 결국 언젠가 자신의 임계 구역에 진입할 수 있다 (**진입 가능성 보장**).
- 그러나, **요청한 순서대로 임계 구역에 진입한다는 보장은 없다** (FIFO 공정성 미보장).
- 이후 **Section 16.3.5**에서 Maude를 이용해 이러한 성질(상호배제, 순서 보존)을 구체적으로 분석하는 방법을 다룬다.

13.3 Distributed Leader Election

분산 시스템에서는 여러 노드 중 **하나를 리더(leader)**로 선출해야 하는 상황이 자주 발생한다.

예:

- 2단계 커밋 프로토콜(two-phase commit)은 리더(조정자, coordinator)가 존재함을 전제로 함.
- 항공기 시스템은 여러 컴퓨터 중 어느 것이 실제로 제어 중인지(리더 역할인지) 결정해야 함.

Leader Election Algorithm의 역할

- 리더 선출 알고리즘은 **하나의 노드를 리더로 선택하고**, **모든 노드가 그 리더에 동의**해야 한다.
- 리더가 고장(crash) 나면, **새로운 리더를 재선출**해야 함.
- 여러 노드가 동시에 리더의 고장을 감지할 수 있으므로, **여러 노드가 동시에 리더 선출을 시작할 수도 있음.**

대표적인 두 가지 알고리즘

1. **Ring-based 알고리즘**
2. **Spanning-tree-based 알고리즘**

→ 두 알고리즘의 목표: "어떤 기준(parameter)" (예: 프로세서 성능, 남은 에너지량 등)에서 가장 좋은 값을 가진 노드를 리더로 뽑는 것.

→ 단, 이 두 알고리즘은 **노드 또는 통신 장애**를 견디지 못함.

Bully Algorithm

- **Bully 알고리즘**은 노드의 **고장 및 복구(failure & recovery)**를 처리할 수 있는 잘 알려진 리더 선출 알고리즘이다.
- 하지만 다음 조건이 필요하다.
  - **실시간 기능(real-time features)**: 타임아웃(timeout), 시간 제한 통신(time-bounded communication)
- 이유: **비동기 분산 시스템(asynchronous system)**에서는 "응답 지연"과 "노드 고장"을 구분할 수 없기 때문에 타임아웃 기반의 고장 감지가 불가능하다.

13.3.1 A Ring-based Leader Election Algorithm

이 알고리즘은 **Chang과 Roberts**가 제안한 것으로, 모든 노드가 논리적인 **링 형태**로 배열되어 있고, 각 노드는 **다음 노드(next node)**를 알고 있다.

알고리즘 동작 요약

1. **선거 시작**
   - 리더가 고장났다고 감지한 노드가 새로운 선거를 시작한다.
   - 자신의 **값(value)**과 **ID**를 담은 **election 메시지**를 다음 노드로 보낸다.
2. **메시지 비교 및 전달**
   - 메시지를 받은 노드는 받은 값과 자신의 값을 비교한다.
     - 받은 값이 더 크면 → **그대로 전달**
     - 자신의 값이 더 크면 → **자신의 값과 ID로 새로운 메시지를 생성해 전달**
3. **리더 결정**
   - 노드가 **자신의 ID가 담긴 election 메시지**를 받게 되면, 자신이 링 내에서 가장 큰 값을 가진 노드임을 알게 되고, **리더(leader)**로 결정된다.
4. **리더 알림**
   - 새로운 리더는 자신의 ID를 담은 **leader 메시지**를 다음 노드로 보낸다.
   - 다른 노드는 리더의 ID를 저장하고, 자신이 리더가 아니면 메시지를 **다음 노드로 전달**한다.

핵심 요약

링 구조에서 모든 노드가 서로의 값을 비교하며 메시지를 전달하고, 가장 큰 값을 가진 노드가 최종 리더로 선출된다.

13.3.2 A Spanning-Tree-based Algorithm for Wireless Networks

개요

- 무선 네트워크에서는 각 노드가 여러 **이웃 노드(neighbor)** 와 “1-hop” 거리로 통신할 수 있다.
- 이 환경에서는 링(ring) 기반 알고리즘이 적합하지 않기 때문에, **스패닝 트리(spanning tree)** 구조를 기반으로 리더를 선출하는 알고리즘이 사용된다.
- 각 노드는 자신의 이웃을 알고 있으며, 네트워크는 **연결된 무방향 그래프(connected undirected graph)** 로 가정한다.

알고리즘의 3단계(3 Phases)

1. **Spanning Tree 생성 (Phase 1)**
   - 시작 노드가 `election` 메시지를 이웃들에게 보냄.
   - 메시지를 처음 받은 노드는 보낸 노드를 자신의 **parent**로 저장하고, 다른 이웃에게 `election` 메시지를 전달.
   - 이미 `election` 상태인 노드는 `ack(0)` 메시지로 응답.
2. **최대값 계산 (Phase 2)**
   - 각 노드는 자신의 서브트리(subtree)에서 가장 큰 노드 값(`max`)을 부모에게 전달.

- 부모는 자식들의 `ack` 메시지를 모두 받은 후, 그 중 최대값을 자신의 `max`로 업데이트하고 자신의 부모에게 `ack(max)`를 전달.
- 루트 노드는 최종적으로 트리 전체의 최대 노드 값을 얻게 된다.

3. **리더 전파 (Phase 3)**
   - 루트 노드는 최종 리더(가장 큰 값)를 `leader(I)` 메시지로 이웃에게 전파.
   - 노드가 `leader` 메시지를 처음 받으면 그 리더를 저장하고, 이웃들에게 다시 전파.
   - 이미 리더 정보를 가진 노드는 메시지를 무시한다.

Maude 모델 요약

- 각 노드 객체(`Node`)는 다음 속성을 가짐:

  ```
  parent, max, state, leader, neighbors
  ```

- 상태(`state`)는 다음 세 가지 중 하나:
  - `idle` : 선거 전 상태
  - `waitForAck(neighbors)` : 이웃들로부터 ack 대기 중
  - `waitForLeader` : 리더 전파 대기 중
- 주요 규칙:
  - `startLeaderElection`: 시작 노드가 election 메시지를 multicast
  - `rcvElection1`: 처음 election 메시지를 받으면 parent 설정 후 전달
  - `rcvElection2`: 이미 선거 중이면 ack(0) 응답
  - `rcvAck`: ack를 받으면 max 갱신 및 대기 목록에서 제거
  - `ackParent`: 모든 ack 수신 후 부모에게 ack(max) 보냄
  - `sendLeader`: 루트가 전체 트리의 max를 결정하면 leader 메시지 전파 시작
  - `rcvLeader1/2`: leader 메시지를 처음 받으면 저장 후 전파, 이미 있으면 무시

예시 (Maude Analysis)

세 개의 노드 (1, 2, 3)가 있을 때:

- 각 노드의 초기 `max`는 자신의 ID
- 선거를 시작하면 최종적으로 **노드 3이 리더로 선출됨**

  → 가장 큰 값(3)을 가진 노드가 리더가 되는 구조

13.4 Consensus Algorithms

문제 배경

- **복제(replication)**는 가용성을 위해 필요하지만, 여러 복제 사이트 간 불일치 문제를 일으킬 수 있음.

  → 예: 경매 시스템에서 한 사이트는 **X**를 **A**에게 팔고, 다른 사이트는 **같은 X**를 **B**에게 파는 상황 발생.

해결책 1 - Two-Phase Commit (2PC)

- **2단계 커밋 프로토콜**을 사용하면 이러한 문제를 방지할 수 있다.
  - 모든 트랜잭션은 2PC가 끝나기 전까지 **커밋 금지**
  - 한 사이트가 다른 사이트의 커밋 시도를 **거부(veto)** 가능
- 장점: 일관성 유지
- 단점:
  1. 충돌하는 트랜잭션은 중단(abort)되며, 복제본들이 **누가 구매자인지 합의**하지 못함.
  2. 모든 복제 노드가 커밋 가능해야 트랜잭션이 완료됨 → 일부 노드가 다운되면 전체 트랜잭션이 실패함.
  - 해결 방향:
    - 비실패 노드들이 특정 값(예: 구매자)에 대해 **합의(reach consensus)** 해야 함.
    - 그러나 비동기 시스템에서는 **완전한 합의는 불가능**.
    - 따라서 "모든 노드" 대신 "**대다수(majority)**"가 동의하면 커밋을 승인.

합의 알고리즘의 목표

- 모든 노드가 하나의 **의미 있는 값(value)**에 합의하는 것. (예: "누가 X의 구매자인가?")

합의 알고리즘의 두 가지 조건

1. **두 노드가 다른 값에 합의하지 않도록 보장**
2. **비실패 노드들이 합의에 도달할 가능성을 보장**

- 커밋 여부 결정(commit/abort)이나 리더 선출(leader election)도 결국 "합의"의 특수한 형태다.

간단한 합의 알고리즘 예시

1. **리더 선출 (Elect a leader)**
   - 한 노드가 자신을 리더로 제안.
   - 다수 노드가 동의하면 리더가 됨.
2. **값 제안 (Propose a value)**
   - 리더가 특정 값을 모든 노드에게 제안.
3. **응답 수집 (Count replies)**
   - 과반수가 "ok" 하면 값이 선택됨.
4. **값 전파 (Send out the value)**
   - 리더가 선택된 값을 모든 노드에 알림.

- **특징**
  - 통신 지연이나 노드 실패를 허용.
  - 노드들이 **서로 다른 값에 합의하지 않음**을 보장.
  - 모든 비실패 노드가 동일한 값에 합의할 수 있음(운이 좋을 경우).
- **단점**
  - 여러 노드가 동시에 리더로 나설 수 있으며 과반을 얻지 못할 수 있음.
  - 리더가 실패할 수 있음.

개선 아이디어 - Paxos 알고리즘

- 합의 실패 시 알고리즘을 다시 실행하는 방식.
- 단순하지 않지만, 분산 시스템에서 가장 유명한 합의 프로토콜 중 하나.
- **Paxos Consensus Algorithm (Leslie Lamport)**

  → 클라우드 시스템의 핵심 프로토콜.

  → 이 책에서는 자세히 다루지 않지만, Maude로 모델링해볼 수 있는 흥미로운 주제.