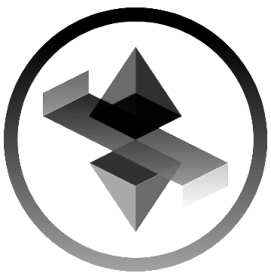
Consensus in Distributed Systems

박진형, 박찬현, 이동식, 이부형, 전창석, 홍종화
이더리움연구회 4 기 기술 리서치 분과



이더리움 연구회
korea ethereum study group

목 차

1. 개요

기술의 발달은 너무나 빨라서 사람들이 채 적응하기도 전에 또 다른 신기술들의 영향을 받는다. 특히, 4 차 산업혁명이라는 거대한 물결 속에 기술의 진보는 우리 사회에 더 큰 영향을 끼치고 있다. 인공지능, 데이터마이닝, 클라우드 컴퓨팅 등 4 차 산업을 대표하는 미래유망 신사업 분야에 높은 관심이 쏠리는 이유다. 최근에는 암호화폐의 기반 기술인 블록체인이 기존의 금융서비스 판도를 바꿀 것이라는 기대감 속에 새롭게 조명을 받고 있다. 불과 몇 년 전까지만 해도 블록체인은 '뜬구름 잡는 이야기'였지만 지금은 수많은 사람들이 연구하고 다양한 사업을 창출하는 핵심 기반이 되고 있다.

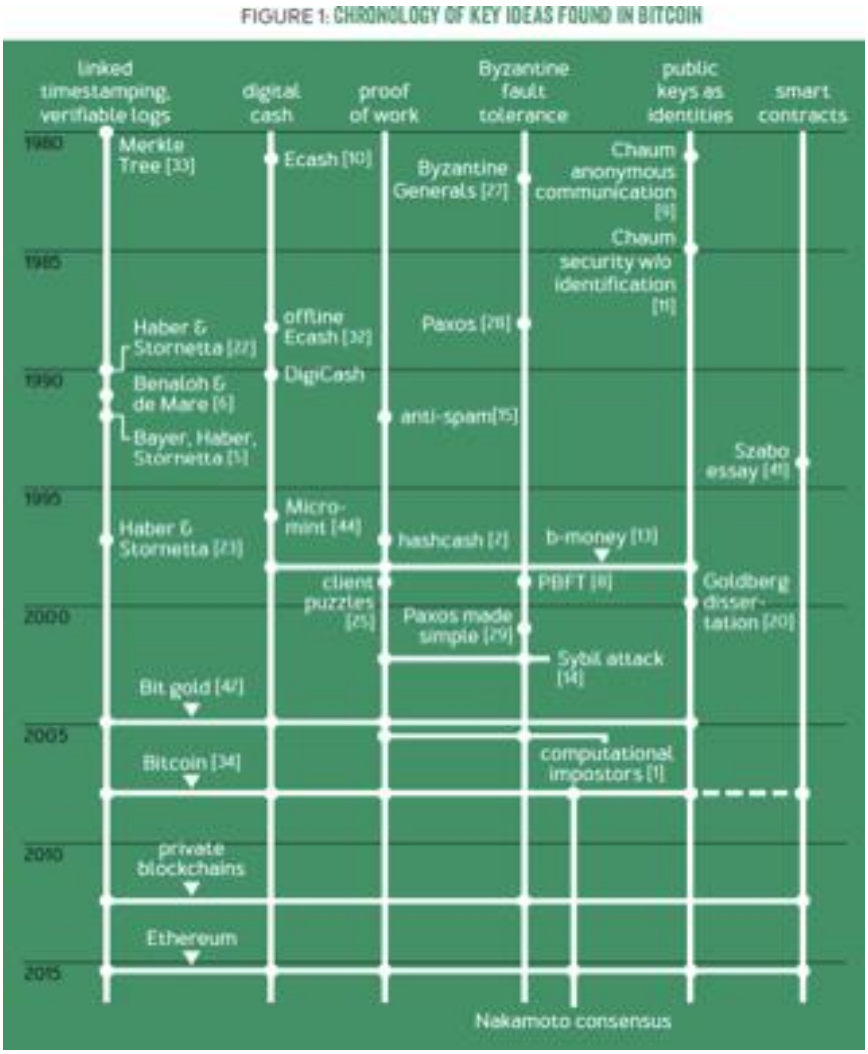
1.1 블록체인의 등장

블록체인은 **사토시 나카모토**(Satoshi Nakamoto)라는 익명의 인물이 **비트코인**: P2P **전자화폐 시스템**(Bitcoin: Peer-To-Peer Electronic Cash System) [1]이라는 9 쪽짜리 논문을 공개하면서 세상에 본격적으로 등장했다. (엄밀히 따지면 사토시 논문에는 블록체인이라는 단어는 나오지 않는다)

2008 년 10 월에 공개된 이 논문은 현재의 중앙화 금융 시스템에 대한 도전장이자, 개인 간 온라인 직접 거래(P2P)의 가능성을 보여준 획기적인 사례였다. 그 이유는 분산 컴퓨팅 분야에서 오래전부터 풀지 못했던 비잔티움 장군 문제(Byzantine Generals' problem)에 대한 실용적인 해결책을 제시했기 때문이다. 간단히 설명하자면, 사토시는 악의적인 노드가 존재할 수 있는 네트워크상에서도 합의를 이끌어낼 수 있음을 작업증명 방식(Proof of Work)을 통해 제시했다. 이 해법은 분산 컴퓨팅 과학에서 획기적인 발자취를 남기고, 신뢰할 수 있는 디지털 화폐 시대의 초석을 마련한 것으로 평가된다.

그러나, 사실 탈중앙화를 이루고자 하는 블록체인 세상은 하루아침에 탄생한 것은 아니다. 블록체인의 시작은 **사이퍼 펑크**(Cyber Punk) **운동** [2]이 벌어진 1980 년대로 거슬러 올라간다. 사이퍼펑크 운동은 정부가 개인의 사생활정보에 접근하지 못하도록 암호(cypher) 체계를 개발하고 사용해야 한다고 주장한다. 사이퍼펑크 운동은 데이비드 차움(David Chaum)의 이캐시(Ecash), 아담 백(Adam Back)의 해시캐시(HashCash), 웨이 다이(Wei Dai)의 비머니(B-money), 그리고 닉 재보(Nick

Szabo)의 비트골드(Bit Gold) 등 중요한 암호 및 거래 기술을 만드는 초석이 되었다. 즉, 블록체인은 사이퍼펑크(Cyber Punk)들의 지속적인 노력과 분산시스템에 대한 수 많은 연구의 집약체라고 볼 수 있다. 비트코인이 세상에 나오기까지의 과정은 아빈 나라야나(Arvind Narayana)과 제레미 클락 (Jeremy Clark)의 글 'Bitcoin's Academic Pedigree' [3]을 통해서도 자세히 알 수 있다. (해당 리서치는 지난 3 기 발표회 [4]에서도 확인할 수 있다.)



〈그림 1.1. 비트코인이 나오기까지의 아이디어 연대표〉

〈그림 1.1〉을 보면 알 수 있듯이 다양한 이론들이 접목되어 발전한 형태가 현재의 블록체인 기술이다. 암호학과 디지털 재화, 합의 과정, 타임스탬프, 증명 방식 등 다양한 분야의 기술들이 블록체인 기술에 영감을 주면서 발전하고 있다. 특히, 분산 시스템 하에서 블록체인 기술이 가지는 의의는 아주 큼으로, 분산 시스템과 IT 인프라 아키텍처를 소개하고자 한다.

1.2 분산 시스템 (Distributed System)

분산 시스템(Distributed System) [5,6]**이란** 기존의 중앙 집중 처리를 하던 시스템에서 탈피하여, 통신 네트워크를 통해 서로 약결합된(Loosely Coupled) 처리기들의 집합을 의미한다. 즉, 사용자는 하나의 단일 시스템으로 보지만 뒤에서는 여러 다양한 컴퓨터들로 구성되어 있다. 분산시스템의 장점은 개별 컴퓨터의 안정성이 낮아도 무관하므로 저렴한 비용의 장비로도 구축할 수 있다. 또한, 더 많은 컴퓨터를 이용해서 시스템 전체의 성능을 향상할 수 있어 확장성이 좋다는 특징을 가지고 있다. 그러나 분산시스템은 서버의 수가 증가하면 이를 운영하기 위한 구조가 점점 더 복잡해진다. 더불어 서버가 고장이 나더라도 이로 인한 영향을 최소화하기 위해서는 서버의 역할을 세세하게 검토해야 한다. 서버를 운영하는 방법은 여러 가지 형태로 존재하며 크게 집약형과 분할형으로 구분된다. 몇 가지 대표적인 예시를 알아보자.

◆ **집약형 아키텍처**

집약형 아키텍처는 흔히 옛날 영화에 나오는 방 하나에 기계와 거대한 장치들이 있는 장면을 떠올리면 된다. 집약형은 대형 컴퓨터를 이용해 모든 업무를 처리하는 형태이다. 하나의 컴퓨터로 모든 처리를 하므로 집약형이라고 부른다. 그러나 한 대의 컴퓨터로 모든 처리를 감당해야 하므로 대부분은 컴퓨터를 구성하는 주요 부품들이 모두 다중화 되어 있다.

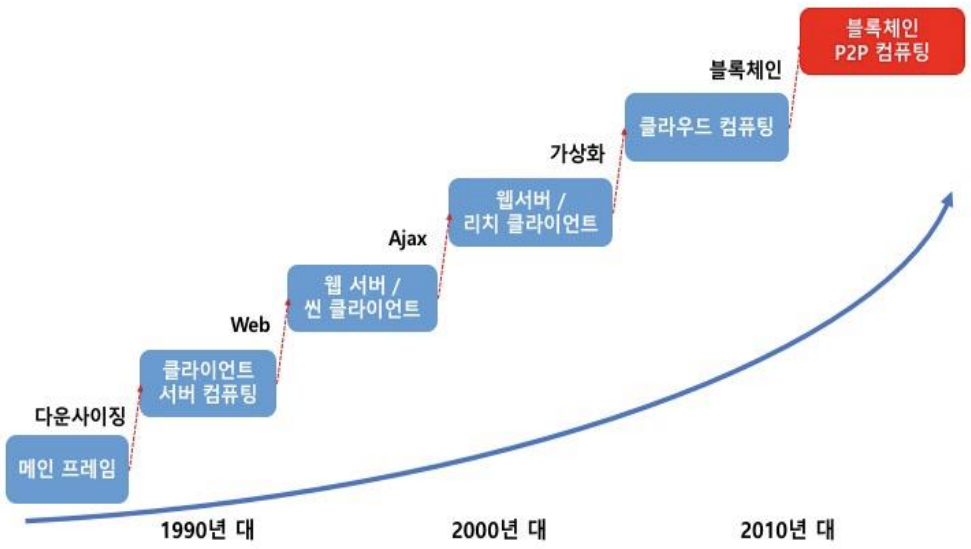
◆ **클라이언트-서버형**

클라이언트-서버형은 수직 분할형의 하나로 업무 애플리케이션, 미들웨어, 데이터 베이스 등의 소프트웨어를 물리 서버상에서 운영하고, 이들의 소프트웨어를 클라이언트 혹은 단말이라고 불리는 소형 컴퓨터가 접속해 이용하는 형태이다. 클라이언트-서버는 클라이언트(client)가 전용 소프트웨어를 설치해야 하며, 업무 애플리케이션이 갱신이 될 때마다 클라이언트 소프트웨어가 업데이트되야 한다. 또한, 서버의 처리가 집중되면 확장성의 한계가 발생할 수 있다. 이를 발전 시킨 것이 3 계층형 아키텍처이다.

◆ **3 계층형 아키텍처**

3 계층형은 클라이언트-서버형을 발전시킨 형태로 위에서 프리젠테이션 계층, 애플리케이션 계층, 데이터 계층의 3 층 구조로 분할 되어 있다. 이는 서버 부하의 집중을 개선한 것으로 클라이언트는 정기적인 업데이트를 하지 않아도 된다는 장점이 있다. 그러나 클라이언트-서버보다 복잡하다는 단점이 존재한다.

이 외에도 수평 분할형 아키텍처나 지리 분할형 아키텍처 등 다양한 아키텍처가 존재한다. 2000 년대부터는 웹이 발전함에 따라 경량 웹(Thin Client) 처리방식이 제공되기 시작했다. 이는 HTML 을 렌더링한 결과만을 제공하기 때문에 가볍다는 장점이 있다. 이후 경량 웹 방식은 단일 웹 페이지 애플리케이션 (SPA, Single Page Application)으로 발전하였다. Ajax(Asynchronous Javascript + XML) 등을 이용한 다양한 웹서비스가 증가하다 보니 비동기식 처리가 많아지게 되었다. 동기(Synchrony)는 누군가에게 일을 부탁하고 그 일이 끝나기까지 기다리는 것이고, 비동기(Asynchrony)는 '끝나면 말해'라고 하고 다른 일을 하는 것을 의미한다. 비동기의 경우 다른 작업을 하면서 호출을 한 처리가 되었는지 계속해서 체크를 한다. 이후 가상화 기술의 발전으로 스토리지부터, 서버 컴퓨팅, 네트워크, 운영체제에 이르기까지 모든 자원을 추상화할 수 있게 되었다. 이렇게 발전하게 된 클라우드 컴퓨팅으로 인해 더 인프라를 직접 구축하지 않아도 적은 비용으로 서비스를 개발할 수 있는 환경이 조성되었다.



〈그림 1.2. 컴퓨터의 발전 〉

◆ 블록체인 P2P 컴퓨팅

P2P 컴퓨팅이란 네트워크에 참여한 모든 컴퓨터가 동일한 역할과 기능을 수행하는 방식을 의미한다. 이는 클라이언트가 서버의 역할을 동시에 수행한다는 의미이다. P2P 네트워크는 중앙집중형, 링형, 계층형, 완전 분산형 등 다양한 방법으로 연결이 될 수 있다. 이더리움(Ethereum) 플랫폼 [7]의 경우, P2P 기반에서 완전 분산형 방식으로 컴퓨팅 프로세스, 파일, 메시지 공유 기능을 활용하여 다양한 응용 서비스를 지원하고자 한다. 이처럼 다양한 아키텍쳐가 상황에 맞게 발전하며, 오늘날 다양한 블록체인 시스템 발전의 촉매제 역할을 하고 있다.

2. FLP Impossibility

분산 시스템 이론 중 빼놓을 수 없는 이론은 바로 FLP Impossibility 이다. FLP Impossibility 는 1985 년 피셔(Fischer), 린치(Lynch), 패터슨(Paterson)이 공동으로 저술한 'Impossibility of Distributed Consensus with One Faulty Process' [8] 논문에서 나온 이론이다. 이는 비동기식 네트워크 환경에서 실패할 가능성이 있는 노드가 단 하나라도 있을 경우 'Safety'와 'Liveness' 이 두 가지 속성을 모두 만족하는 합의 알고리즘은 존재할 수 없음을 증명하였다. **비동기 네트워크(asynchronous network)** 특성상 통신 지연시간을 예측할 수 없어 특정 문제로 인해 노드가 작동에 실패한 것인지 아니면 단순히 네트워크 지연으로 인해 응답 시간이 오래 걸린 것인지 알 수 없다. FLP Impossibility 를 이야기하기 이전에 FLP 이론에 등장하는 단어에 대한 정의를 하고자 한다.

◆ **동기성(Synchronous)**

동기성이란 시스템에 콜(요청)을 하였을 때 콜이 끝날 때까지 노드가 대기하고 완료가 되면 결과값을 반환(return)하는 네트워크를 의미한다. 소프트웨어의 관점에서는 하나의 과정(routine)이 완벽하게 끝나야 반환을 하는 형태를 의미한다.

◆ **비동기성(Asynchronous)**

비동기성이란 하나의 콜을 하고, 완료의 유무와 상관없이 응답(respond)를 하는 것을 의미한다. 콜 이후에 노드는 다른 작업을 병행할 수 있으며, 콜이 완벽하게 반환이 되면 결과물을 다시 가져오는 형태이다.

다음 정의를 한 특징을 바탕으로 분산 네트워크에서는 네트워크 간의 통신을 동기성과 비동기성으로 구분한다.

동기성 네트워크(Synchronous Network)의 경우 메시지를 전송하고, 이를 전파하는 과정(Message Propagation)에서 네트워크의 정보 전파 시간을 완전히 조절할 수 있는 것을 의미한다. 또한, 모든 노드는 자신이 보낸 메시지에 도달하는 시간을 조절 혹은 예측을

할 수 있다. 이때, 정확하게 도달 시간을 예상할 수 있어야 하므로 이를 방해하는 어떤 경우의 지연(Delay)도 일어날 수 없다.

이와 달리 **비동기성 네트워크(Asynchronous Network)**의 경우 메시지를 전송하고, 이를 전파하는 과정에서 네트워크의 정보 전파 시간을 예측할 수 없는 경우를 의미한다. 네트워크 상에 존재하는 수많은 방해노드(Botnet)과 DDoS 공격 등으로 인해 노드 간의 메시지가 완전하게 유실되거나 사라지는 가능성이 있음을 전제로 한다.

다음의 동기성과 비동기성을 중첩한 경우는 **부분 동기성 네트워크(Partial Synchronous)**라고 하며, 네트워크 통신 상태를 사용자가 어느 정도 조절할 수 있는 경우를 의미한다. 네트워크 상에서의 방해가 있더라도 일정 시간 하(Upper Bound)에는 반드시 메시지를 응답 받을 수 있다. 이는 Chapter 4 의 PBFT 에서 더 알아보자.

FLP 이론은 앞서 언급한 바와 같이 비동기성 네트워크(부분 동기성 네트워크도 포함) 환경에서 실패 가능성이 단 하나라도 있을 경우 'Safety'와 'Liveness' 모두를 만족하기 힘들다는 이론이다. 여기서는 다루지 않지만, Paxos 와 같이 궁극적으로 최종 합의가 되는 경우(Eventual Liveness)는 가능하지만, 시간의 범위(Time Bounded)가 정해진 경우는 불가능하다.

FLP Impossibility 에 따르면, 분산 네트워크는 다음 조건들을 만족한다고 가정하였다.

1) Termination: 모든 동작하는 노드들은 언젠가는 어떤 값을 결정한다.
2) Agreement: 노드들은 모두 동일한 값을 결정한다.
3) Validity: 결정된 값은 어떤 과정을 통해 결정되어야 한다. 가령, 노드가 어떤 메시지를 받던 항상 0 의 값을 출력하는 알고리즘을 가지고 있다면 이를 만족시키지 못한다.

FLP Impossibility 는 다음의 가정을 바탕으로 크게 두 가지로 정리하여 증명 [9]한다.

1) 비동기성 네트워크에서 노드의 합의는 결정성(Deterministic)을 가지지 않고, 노드의 실패 여부 등으로 인해 비결정성(Non-deterministic)을 가지게 된다.

2) 1)의 정리를 바탕으로 아직 합의가 이루어지지 않은 상황은 무한히 합의가 이루어지지 않는 상황으로 이어지는 것이 가능하다.

이는 합의가 발생하지 않을 수 있다는 의미이며, 비동기성 네트워크에서는 합의가 보장되는 합의 알고리즘은 불가능하다는 것이다. 다음의 증명 정리를 바탕으로 FLP 이론을 3 가지의 경우로 간단하게 정리할 수 있다.

1) 증명에서 가정되는 합의는 결정성(Deterministic)을 가진다.
2) 노드가 실패할 수 있는 환경은 결정성(Deterministic)을 가지지 않는 환경이다.
3) 결정성(Deterministic)을 가지지 않는 환경에서는 어떠한 경우에도 결정성을 가질 수 없다.

FLP Impossibility 에서는 다음과 같은 증명을 통해 비동기식 네트워크에서는 'Safety'와 'Liveness'를 모두 만족하며 합의하는 것은 불가능하다고 이야기한다. 블록체인에서는 이 FLP Impossibility 를 우회하기 위해 다음과 같은 방법을 이용한다.

1) 합의를 재정의한다. (Liveness over Safety)
2) 비동기성을 포기한다. (Safety over Liveness)

블록체인에서 이 두 가지 방법으로 FLP Impossibility 를 우회하는 방법을 차례대로 알아보자.

3. Byzantine Fault Tolerance (BFT)

현재 분산 네트워크 시스템에서는 BFT 기반의 합의 알고리즘이 많이 사용되고 있다. BFT (Byzantine Fault Tolerance)는 분산 네트워크 안에서 악의의 노드 (Byzantine)가 네트워크에 존재하는 경우에도 선의의 노드들이 안전하게 네트워크를 사용할 수 있는 합의 방법은 무엇인지 연구하는 분야이다.

현재 BFT 가 어떻게 블록체인 기술에서 활용되고 있는지 알아보기 이전에, 1982 년 레슬리 램포트 (Leslie Lamport) [10] 외 2 명이 발간한 논문을 중심으로 BGP (Byzantine Generals' Problem)를 살펴보면서 초기의 분산 네트워크 합의 알고리즘 연구 방향과 그 내용을 살펴본다.

3.1 Two Generals' Problem

두 장군 문제는 신뢰할 수 없는 네트워크에서의 노드 간 메시지가 올바르게 전송되기 위한 조건을 논의하면서 처음 나온 개념이다. 1957 년에 출판된 논문" *SOME CONSTRAINTS AND TRADEOFFS IN THE DESIGN OF NETWORK COMMUNICATIONS*" [11]에서 처음 논의를 시작했고, 두 장군 문제라는 용어는 1978 년에 만들어졌다. 이는 네트워크를 구성하는 노드를 장군으로 가정하고, 두 장군이 공동의 적을 공격하는 시나리오이다. 두 장군은 적진을 사이에 두고 공격 시각을 합의해야 하는 상황이다. 다음은 합의를 위해 두 장군이 고려해야 하는 사항들이다.

1) **반드시 두 장군이 동시에 공격해야만 이길 수 있다.**
2) **공격 시각을 합의하기 위해 각자의 메신저를 보낸다.**
3) **메신저는 메시지 전달을 위해 반드시 적진을 통과해야 한다.**

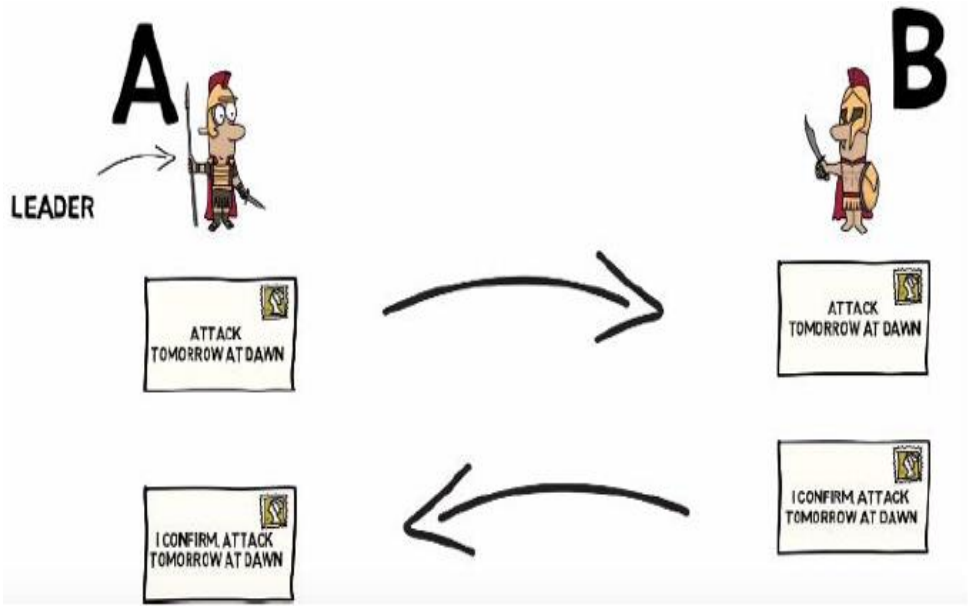
이때 장군들은 공격 시간을 결정하기 위해 다음과 같은 생각을 할 수 있다.

1) 장군 A 가 메신저를 통해 장군 B 에게 공격 시간을 전달

2) 장군 B 가 장군 A 의 메시지를 확인 후 회신

3) 장군 A 가 장군 B 의 회신 메시지를 확인

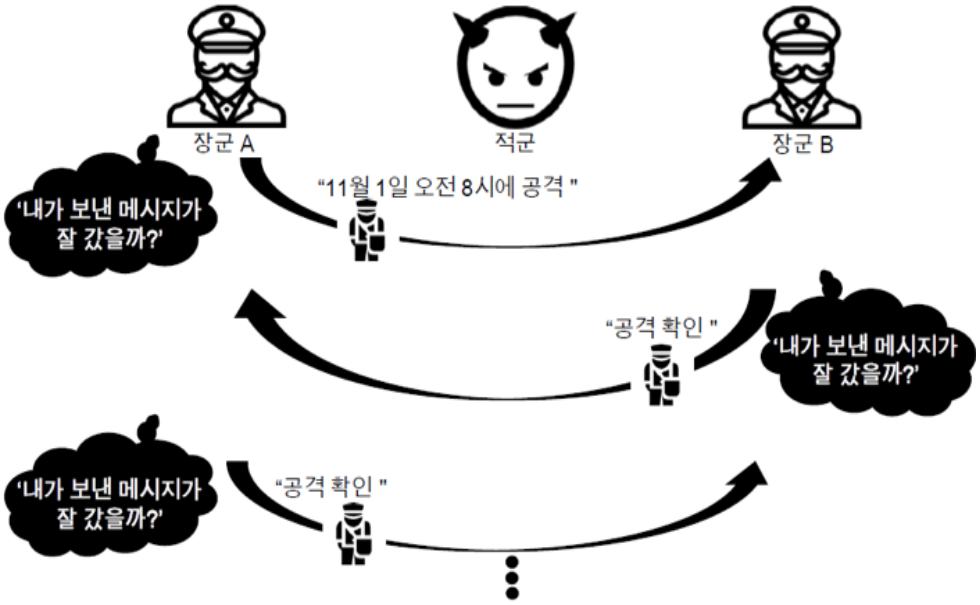
4) 결정된 공격 시각에 일제히 공격!



〈그림 3.1. 공격 시각 결정 방법 [12]〉

다음과 같은 공격 시간이 추가되었을 때 장군들은 고민에 빠지게 된다. 가령 장군 A 가 "11 월 1 일 오전 8 시에 공격"이라는 메시지를 작성해서 장군 B 에게 보낸다. 장군 B 가 장군 A 의 메시지를 받고 "공격 확인"이라는 응답 메시지를 장군 A 에게 보낸다. 장군 A 가 메시지를 받으면 합의는 성공적으로 이루어진다.

그러나, 장군 A 가 보낸 메시지를 가지고 떠난 메신저가 도중에 적군에게 잡힐 수도 있다. 장군 A 의 메신저가 적군의 경계를 무사히 뚫고 장군 B 에게 메시지를 전달했다고 해도 장군 B 의 메신저가 적군에게 잡힐 수도 있다. 이 경우 메시지가 잘 전달될 확률이 항상 100 보다 낮기 때문에 장군들은 합의에 도달할 수 없다. 이는 누구도 메시지가 잘 도착했는지 알 수 있는 방법이 없다는 것을 의미한다.



〈그림 3.2. 두 장군 문제〉

3.2 Byzantine Generals' Problem (BGP)

BGP (Byzantine Generals' Problem)는 앞서 설명한 두 장군 문제를 일반화한 주제로 셋 이상이 네트워크에 참여할 때 서로 합의를 도출하기 위한 방법을 논의한다. 두 장군 문제와 같이 공동의 적을 상대로 승리하고 싶은 장군들을 예시로 제시하고 더불어 메시지 위조가 가능하며 담합도 할 수 있는 배신자를 등장시킨다. 여기서의 논의 주제는 배신자가 존재할 경우에도 합의에 성공하려면 배신자를 포함하여 몇 명의 장군이 있어야 할지를 정하는 것이다.

BGP 에서 기본적으로 제시하는 가정은 다음과 같다.

1) 적은 어느 정도의 병력이 모여서 한 번에 공격을 해야 이길 수 있는 상대이다. 이를 위해 여러 부대가 적진을 둘러싸고 있으며 각 장군들은 합의를 통해 공격을 해야할지 말아야 할지를 결정해야 하는 상황이다.

2) 각 부대는 하나의 장군이 이끌고 있으며 메신저가 전달하는 메시지를 통해서만 서로 소통이 가능하다.

3) 메신저는 반드시 적진을 통과해야지만 다른 부대로 갈 수 있다.

4) 장군들 중 한 명의 사령관이 "공격" 혹은 "후퇴" 를 결정하여 메시지를 모든 장군에게 전달한다.

5) 메시지를 받은 장군들은 나머지 장군들에게 사령관으로부터 받은 메시지를 전달한다. (장군들은 서로 연결되어 있다.)

6) 일정 시간 이후 각 장군들은 자신이 받은 메시지를 종합하여 "공격" 혹은 "후퇴"를 결정하여 명령을 이행한다. (동기화)

메신저(Messenger)가 메시지를 전달하는 도중에 실패할 확률이 있다. 적군에게 포획당하거나 살해당하는 등 실패를 할 수 있다. 혹은 장군들 중에 비잔틴 즉, 악의적인 배신자가 섞여 있을 수 있다. 배신자는 두 가지 경우로 생각할 수 있다.

1. 배신자는 메시지 내용을 위조할 수 있다.
2. 사령관도 배신자가 될 수 있다.

만약 사령관이 "공격" 명령을 하달했을 때 메시지를 받은 장군들이 모두 "공격"을 이행해야 전쟁에서 승리할 수 있다. 그렇기 때문에 배신자의 방해가 있더라도 모두 같은 결정을 할 수 있어야 한다. 이에 램포트(Lamport)는 추가적으로 아래의 두 가지 조건을 제시한다. (IC, Interactive Consistency)

- IC1: 배신자가 아닌 모든 장군들은 같은 명령을 이행한다.
- IC2: 만약 사령관이 배신자가 아니라면, 배신자를 제외한 모든 장군들은 사령관의 명령을 이행한다.

BGP 가 생길 수 있는 최소 조건은 아래와 같이 합의 대상이 3 명일 때이다.

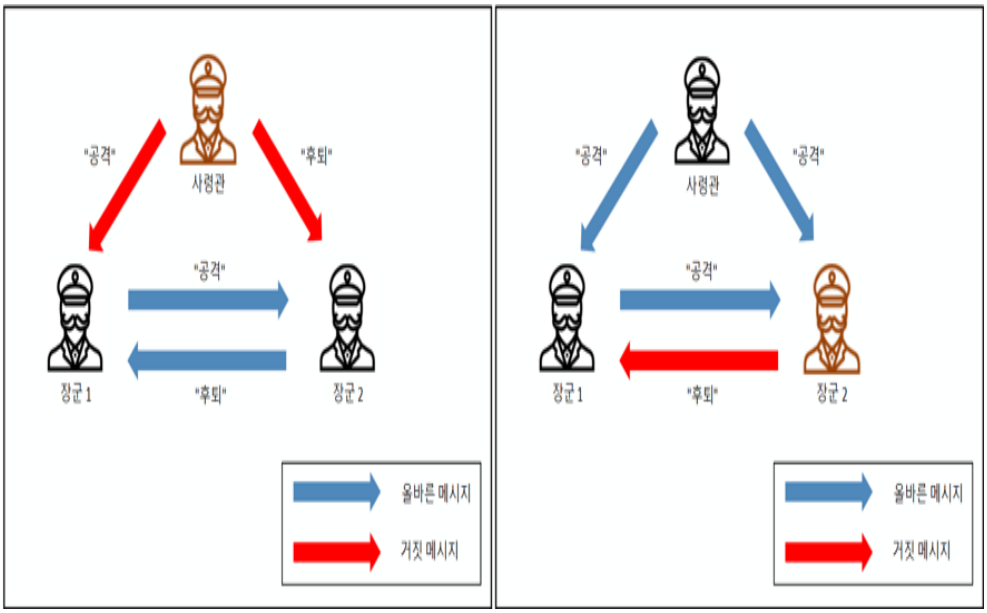
$V(i)$가 i번째 장군이 받은 메시지의 집합이라고 할 때, 일정 시간 이후 배신자가 아닌 장군들은 자신이 받은 메시지 내용으로 "공격"과 "후퇴" 중 하나를 선택하여 이행한다. (과반수 우선 선택 기준)

1) 사령관이 배신자일 경우

- $V(1)$ = {"***공격***", "*공격*"} -〉 장군 1 은 "공격" 으로 판단
- $V(2)$ = {"***후퇴***", "*공격*"} -〉 장군 2 는 "후퇴"로 판단

2) 장군 2 가 배신자일 경우

- $V(1)$ = {"***공격***", "*후퇴*"} -〉 장군 1 은 "공격"으로 판단



〈그림 3.3. 사령관이 배신자일 경우 (왼쪽), 장군 2 가 배신자일 경우 (오른쪽)

$\boldsymbol{m}$의 수를 늘려서 ($\boldsymbol{m} = \mathbf{2}, \mathbf{3}, \ldots$)장군의 수가 총 $\mathbf{3}\boldsymbol{m}$일 때를 생각해보면 배신자 집단에 사령관이 포함될 경우 정직한 장군들이 잘못된 결정을 하도록 조작할 수 있음을 알 수 있다.

안정적인 합의를 위해서는 장군들은 모두 같은 알고리즘을 통해 메시지를 전달하고, 전달받은 메시지에서 "공격"이나 "후퇴" 중 하나의 의미를 도출할 수 있어야 한다. 이를 위해 램포트(Lamport)는 두 가지 알고리즘을 제시한다. 첫 번째는 구두 메시지(Oral Message, OM)이고 두 번째는 서명된 메시지(Signed Message, SM)이다.

3.3 Solution 1: Oral Message (OM)

알고리즘 OM 에서의 기본 가정은 아래와 같다.

1) 장군들이 보낸 모든 메시지는 올바르게 전달된다.
2) 메시지를 받은 장군들은 누가 보낸 메시지인지 알 수 있다.
3) 더 이상 메시지가 도착하지 않음을 인지할 수 있다.

Algorithm OM(m)

BGP 를 해결하기 위해 램포트가 제안하는 OM 알고리즘은 아래와 같다.

- $OM(0)$

1) 사령관은 모든 장군들에게 메시지 ("공격" 과 "후퇴" 중 하나)를 보낸다.
2) 모든 장군들은 사령관이 보낸 메시지를 수신하여 "공격" 과 "후퇴" 중 하나를 결정한다.

- $OM(m), m$은 배신자의 수

1) 사령관은 모든 장군들에게 메시지 ("공격" or "후퇴" 중 하나)를 보낸다.
2) 각 i에 대해 사령관에게 장군 i가 받은 메시지를 $v(i)$라고 한다. 만약 장군 i 가 아무런 메시지를 받지 못했다면 그 장군은 "후퇴"로 결정한다. 장군 i는 $OM(m-1)$의 사령관 역할로 $v(i)$를 다른 (n-2)명의 장군들에게 보낸다.
3) 장군들은 메시지를 받고 $majority(v(1), v(2), \ldots, v((n-1)))$을 통해 "공격"과 "후퇴" 중 하나를 결정한다.
4) $majority(v(1), v(2), \ldots, v((n-1)))$을 통해 "공격"과 "후퇴" 중 하나를 결정한다.

- ◆ $majority()$**가 값을 결정하는 기준은 다음과 같다.**

A. 장군이 받은 메시지에서 "공격"과 "후퇴" 중 많은 쪽을 택한다. 만약 아무런 메시지를 받지 못했다면 "후퇴"한다.

B. 만약 1 번에 과반수를 선택할 수 없는 경우, 중앙값을 선택한다.

(메시지가 순서 집합의 형태일 때)

앞에서 장군의 수가 $3m$ 이하일 경우에는 BGP 를 해결할 수 없음을 얘기했기 때문에, 여기서는 $3m + 1$일 경우를 가정하여 알고리즘 OM 을 설명한다. 배신자가 1 명일 경우 합의에 참여하는 장군의 수는 사령관 포함 총 4 명이다.

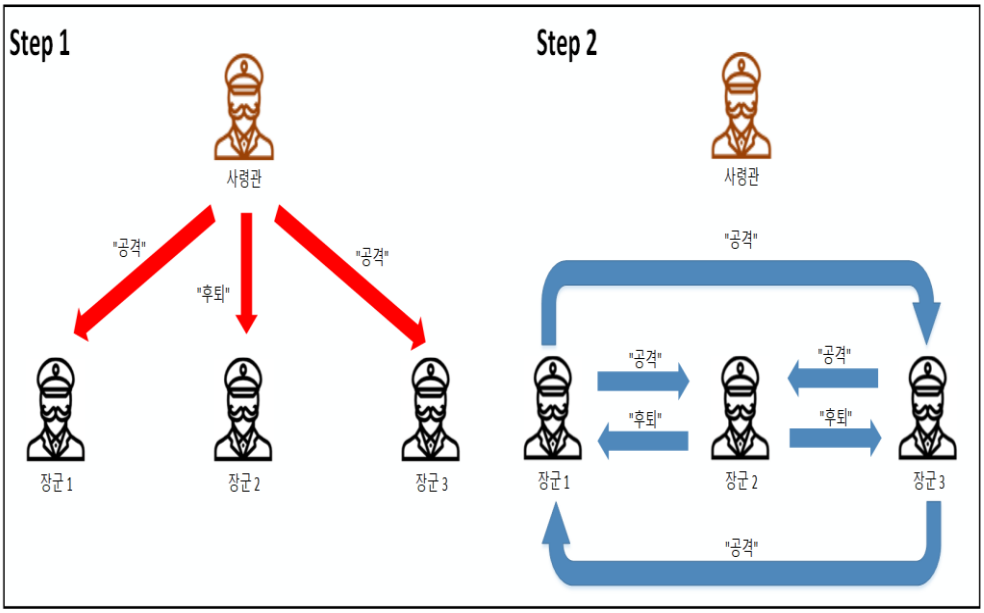
1) 사령관이 배신자일 경우

- ◆ $V(1)$ = {"***공격***","*후퇴*","*공격*"}-〉 장군 1 은 "공격" 으로 결정
- ◆ $V(2)$ = {"***후퇴***","*공격*","*공격*"}-〉 장군 2 는 "공격" 으로 결정
- ◆ $V(3)$ = {"***공격***","*후퇴*","*공격*"}-〉 장군 3 은 "공격" 으로 결정

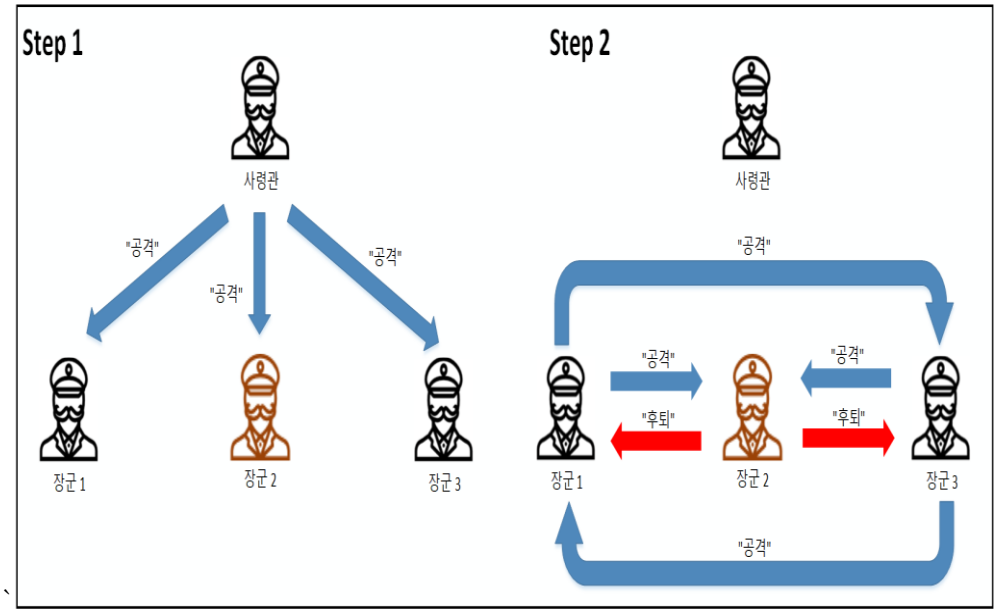
2) 장군 2 가 배신자일 경우

- ◆ $V(1)$ = {"***공격***","*공격*","*후퇴*"}-〉 장군 1 은 "공격" 으로 결정
- ◆ $V(3)$ = {"***공격***","*공격*","*후퇴*"}-〉 장군 3 은 "공격" 으로 결정

두 가지 경우 모두 배신자가 아닌 장군들이 모두 "공격"으로 결정하였음으로 합의에 성공하여 전쟁에 승리할 것이라 예상할 수 있다.



〈그림 3.4. OM(1), 사령관이 배신자일 경우〉



〈그림 3.5. OM(1), 장군 2 가 배신자일 경우〉

알고리즘 OM 은 $n \geq 3m+1$일 때 안전하게 사용할 수 있지만, 재귀적으로 동작하기 때문에 배신자의 수가 늘어날수록 메시지의 전달 과정이 지수적으로 (기하급수적으로) 늘어난다. 예를 들어, 배신자가 3 명이라면 $OM(3) <- OM(2) <- OM(1) <- OM(0)$ 과정을 거친다. 즉, OM 알고리즘을 $(m+1)$번 거쳐야 합의에 도달할 수 있다는 의미이다. 이에 따른 복잡도는 아래 표와 같이 나타낼 수 있다.

배신자의 수 (m)	복잡도 (n 은 장군의 수)
0	O(n)
1	O(n^2)
2	O(n^3)
3	O(n^4)
…	…

〈표 3.1. 메시지 교환 횟수에 따른 OM 의 복잡도〉

3.4 Solution 2: Signed Message (SM)

램포트(Lamport)의 두 번째 해결 방안은 메시지에 서명을 붙임으로써 누가 보낸 메시지인지 인지할 수 있도록 하는 방법이다. 기존 OM 의 가정에 다음의 가정이 추가된다.

1) 배신자가 아닌 장군의 서명은 위조될 수 없으며, 메시지를 수신한 장군은 서명된 메시지의 내용이 변경되었을 경우 인지할 수 있다.
2) 작성된 서명은 누구나 검증할 수 있다.

서명한 메시지는 다음과 같이 정의한다. 사령관을 포함한 장군들은 각자의 식별자 (ID)를 가진다.

- **v: 0 –〉 사령관이 서명한 메시지 v**
  - v (value: "공격" or "후퇴" 둘 중 하나의 값을 가짐): 메시지 내용
  - 0: 사령관의 ID

- **$v:i:j$ –〉 장군 i와 장군 j의 서명이 담긴 메시지 v**
  - j: 메시지 v에 서명한 장군 j
  - i: 메시지 $v:j$에 서명한 장군 i

SM 알고리즘의 동작 과정은 아래와 같다.

- 사령관은 모든 장군들에게 서명하고 $v:0$을 보낸다.
- 각 i에 대해,

a. 장군 i 가 처음 사령관의 메시지 $v:0$를 받았다면,

- $V(i) = \{v$
- $v:0$에 자신의 서명을 붙여서 $v:0:i$를 다른 장군들에게 전달한다.

b. 장군 i가 메시지 $v:0:j_1:j_2:\ldots:j_k$를 받았는데, $V(i)$에 v가 없다면 (i.e., 계속 "공격" 을 받았는데, 방금 온 메시지는 "퇴각" 이라면)

- $V(i)$에 v를 추가한다.
- $j(1)$~$j(k)$를 제외한 다른 장군들에게 메시지 $v:0:j_1:j_2:\ldots:j_k:i$를 전달한다.

더 이상 메시지가 오지 않을 때, $choice(V(i))$의 값을 실행하여 공격할지 퇴각할 지 결정한다.

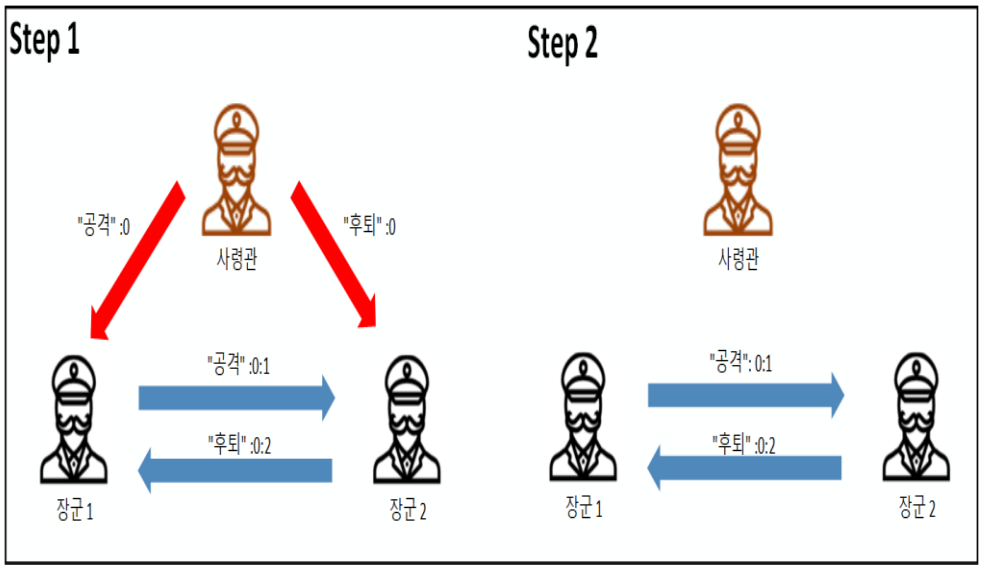
- if a set V has one single element v, then $choice(V) = v$
- choice(Φ) = R, where Φ is the empty set
  - RETREAT is default
- choice(A,R) = R
  - RETREAT is default
- set V is not a multiset (recall definition of a multiset)
- thus set V can have at most 2 elements, e.g. V = {A,R}.

〈그림 3.6. 선택을 위해 사용하는 함수 choice〉

앞에서 배신자가 1 명일 때 장군이 4 명 이상이라면 합의가 가능함을 보였으니 ($n \geq 3m + 1$) 3 명의 장군이 합의하는 경우를 생각해보자. 사령관이 합의를 방해하기 위해 한 명에게는 "공격" 이라고 전달하고, 다른 한 명에게는 "퇴각"이라고 지시한다. 장군들은 아직까지 사령관이 배신자인지 모르기 때문에 다른 장군에게 받은 메시지를 그대로 전달한다. 배신자가 아닌 장군 1 과 2 가 받은 메시지는 아래와 같다.

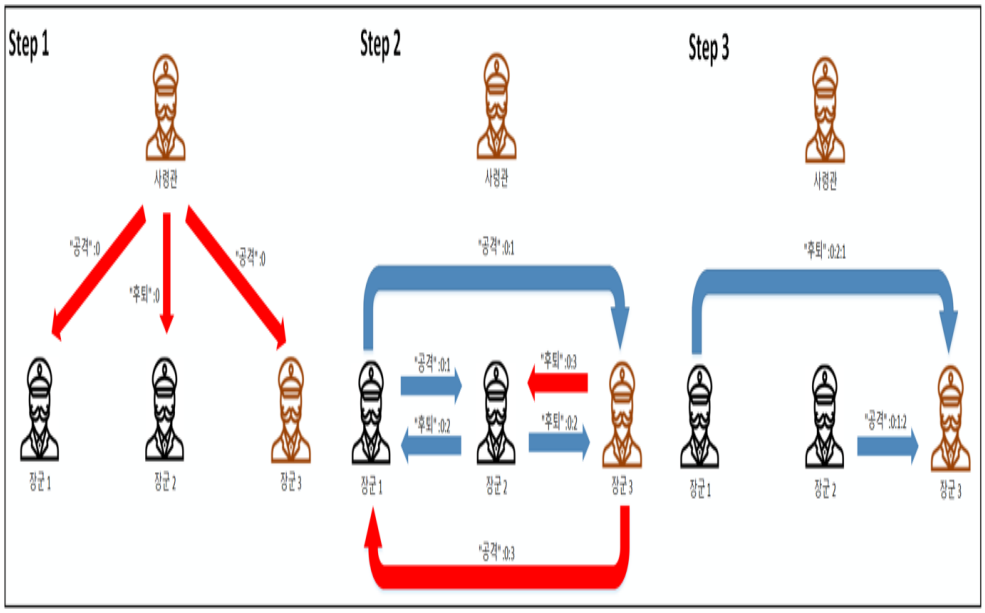
- $V(1) = \{"공격": 0, "후퇴": 0{:}2\}$ -〉 장군 1 은 "후퇴"로 결정
- $V(2) = \{"후퇴": 0, "공격": 0{:}1\}$ -〉 장군 2 는 "후퇴"로 결정

장군 1 과 2 는 두 번째로 받은 메시지를 통해 사령관이 배신자임을 알고 "공격" 명령을 수행하지 않을 것이다. 배신자가 아닌 두 장군이 "후퇴"라는 합의를 얻었기 때문에 IC1 을 만족했다고 할 수 있다. ($choice(V(1)) = choice(V(2))$), IC2 는 사령관이 배신자가 아닌 경우에만 고려) 또한, 알고리즘 SM 에서의 첫 번째 가정에 의해 배신자가 아닌 장군의 서명은 위조할 수 없기 때문에 장군 1, 2 중 하나가 배신자라면 두 번째 메시지에 사령관의 서명이 없었을 것이다.



〈그림 3.7. SM(1), 사령관이 배신자일 경우〉

만약 앞의 케이스에 배신자 장군 3 을 추가해도 $SM(m)$으로 합의에 성공할 수 있을지 알아본다. 메시지 교환 과정은 SM(1)과 거의 비슷하나, 장군 1 은 장군 2 에게 메시지 {"*후퇴*" : 0: 2}를 받고 장군 3 에게 메시지 {"*후퇴*" : 0: 2: 1}을 보내는 과정과 장군 2 가 장군 1 에게 메시지 {"*공격*": 0: 1}을 받고 장군 3 에게 메시지 {"*공격*": 0: 1: 2}를 보내는 두 과정이 추가된다.



〈그림 3.8. SM(2), 사령관과 장군 3 이 배신자일 경우〉

장군 1, 2 가 받은 메시지와 최종 결정은 아래와 같다.

- $V(1) = \{"공격": 0, "후퇴": 0: 2\}$ -〉 장군 1 은 "후퇴"로 결정
- $V(2) = \{"후퇴": 0, "공격": 0: 1\}$ -〉 장군 2 는 "후퇴"로 결정

생각할 수 있는 최소 경우인 장군의 수 n이 $m + 2$일 때 합의가 가능함을 보였으므로, $n \geq m + 2$일 때 알고리즘 SM 을 이용하면 일치된 합의를 통해 전쟁을 승리로 이끌 수 있다. 이상으로 Byzantine Generals' Problem 에 대해 알아보았다. 다음의 논문을 통해 $3m$일

경우 합의가 불가능함을 알 수 있었다. 또한, 램포트의 솔루션을 통해 다음과 같은 경우 합의가 가능함을 알 수 있게 되었다.

1. 알고리즘 OM 을 사용하면 $n \geq 3m + 1$일 때 합의 가능
2. 알고리즘 SM 을 사용하면 $n \geq m + 2$일 때 합의 가능

그러나 램포트의 논문에 따르면 많은 제약조건을 가지게 되고 실제 사용하는 네트워크 안에서 노드 간 합의를 하는 과정은 이보다 어렵다는 것을 예상할 수 있다.

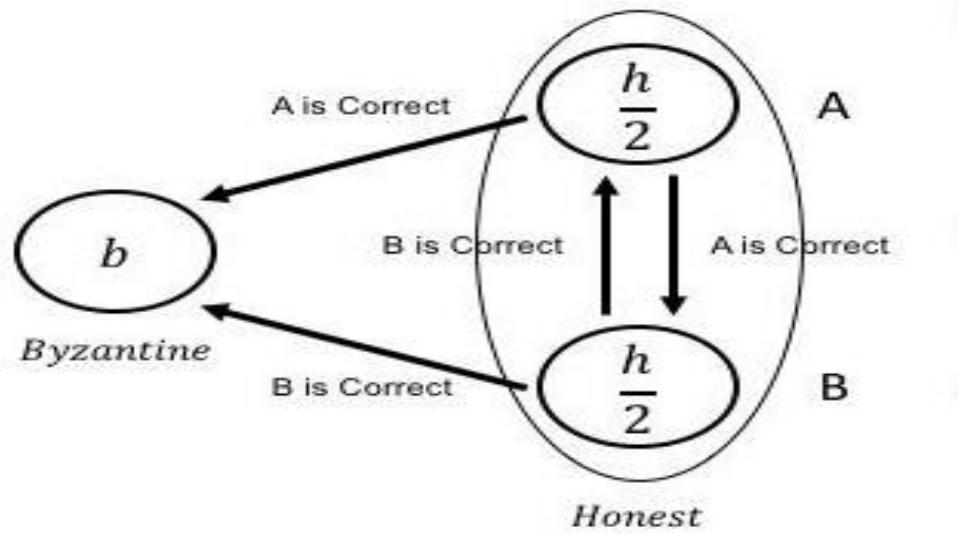
3.5 Generalized BFT

지금까지 BGP 를 통해 $3m + 1$ 이상의 노드가 존재할 때 m 명의 악의적인 노드가 있더라도 합의가 될 수 있음을 알아보았다. 그러나 논문의 증명을 통해 알 수 있듯이 램포트는 $3m + 1$보다 작은 노드의 상황에서 합의(Consensus)가 되지 않는다는 방식으로 증명을 하였다. 본 단락에서는 이를 일반화한 증명을 통해 왜 전체 3m+1 의 노드(m 명의 악의적인 노드)가 필요한지 알아보고자 한다.

여기 전체 N 개의 노드로 구성이 된 네트워크가 있다고 가정한다. 이 중 악의적인 비잔틴 노드가 b 개, 정직한 노드가 h 개 있다. 이 노드들은 전체 N 개의 노드들 중 t 개 이상이 동의한 다음 상태(state)가 있으면 해당 상태로 이동한다. 이 네트워크 안에서는 투표가 비동기적(asynchronous)으로 일어난다. 여기서 일어나는 합의가 Safety 와 Liveness 모두를 만족할 때까지 가능한 t 의 범위를 계산해본다.

$$N = h + b$$

여기 새로운 상태 A 와 B 가 제안 되었다. 이 때 각각의 상태에 대한 투표가 진행 된다. 정직한 노드들은 각각 절반씩 상태 A 와 B 를 지지를 하게 된다.(상태 A 와 B 이렇게 두 가지 선택이 있기 때문에 확률적으로 1/2 이다). 이렇게 투표를 한 정직한 노드들은 자신의 상태에 대한 투표를 다른 노드들에게 전달 한다.



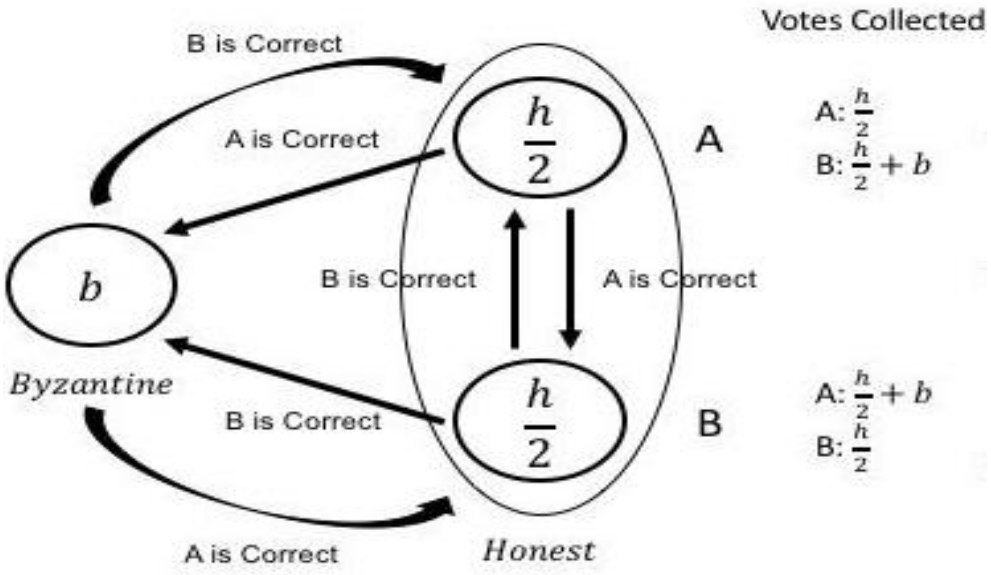
〈그림 3.9. 정직한 노드들의 투표와 전파〉

정직한 노드들이 상태에 투표를 전파하였다. 남은 노드들은 비잔틴, 즉 악의적인 노드들이다. 비잔틴한 노드들은 네트워크를 원할 하지 못하게 하는 것이 목적이기 때문에 정직한 노드들의 전파를 모두 받고 선택을 하게 된다. 이 때 비잔틴한 노드들은 거짓된 투표정보를 전파한다. 상태 A 에는 B 가 맞다는 투표를, 상태 B 에는 A 가 맞다는 정보를 동시에 보낸다. 이럴 경우 각 상태 A 와 B 는 투표를 다음과 같이 얻게 된다.

상태(State)들이 득표한 투표수		
	상태 A	상태 B
상태 A 로의 투표 수	$\frac{h}{2}$	$\frac{h}{2} + b$
상태 B 로의 투표 수	$\frac{h}{2} + b$	$\frac{h}{2}$

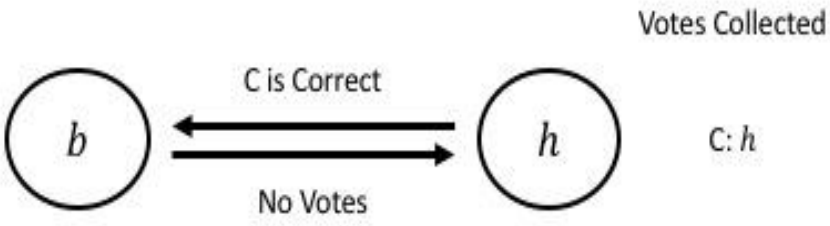
〈표 3.2. 상태(State)가 득표한 투표수〉

다음 표를 보면 알 수 있듯이 만약 $t < h/2 + b$일 경우 노드가 다음 상태로 진행되기 위한 최저 투표수 이상의 표를 받은 상태가 존재하게 된다. 따라서 네트워크의 일부는 A, 나머지는 B 로 진행되어 분할이 된다. 이는 블록체인 네트워크에서 자주 듣게 되는 네트워크 포크(Fork) 상태가 된다. 이를 방지하기 위해서는 $t > \frac{h}{2} + b$가 되야 한다.



〈그림 3.10. 비잔틴 노드의 투표와 최종 상태 투표 수〉

방금 비잔틴 노드들의 악의적인 투표를 막기 위해서는 $t > \frac{h}{2} + b$ 가 되야 한다고 이야기했다. 이는 네트워크 특성 중 Safety 를 만족하기 위함이다. 그렇다면 비잔틴한 노드들은 Liveness 의 방해를 아예 합의에 참여 안함으로 방해할 수 있다. 이 경우는 다음 그림을 통해 알아보자.



〈그림 3.11. 새로운 상태 C 에 대한 투표〉

새로운 상태인 C 가 제안이 되었다고 가정한다. 여기에 정직한 노드들은 h 개 만큼의 지지 투표를 상태 C 에게 보낼 것이다. 비잔틴한 노드들은 네트워크의 Liveness 를 방해하기 위해 투표를 하지 않았다. 이 경우 합의의 대상인 C 에 문제가 없다면 다음 상태인 C 로 네트워크는 진행이 되어야한다. 만약 t›h 인 경우, 최종 투표 수가 합의를 위한 최소 표의 수보다 크기 때문에 네트워크가 상태 C 로 진행되지 못한다. 따라서 Liveness 가 보장이 되기 위해서는 $t <= h$ 여야만 보장이 될 수 있다.

지금까지 이야기한 것을 정리하면 다음과 같다.

$$N = h + b$$

$$t > \frac{h}{2} + b$$

$$t <= h$$

이를 정리하면 다음과 같은 결론이 도출된다.

$$h > \frac{h}{2} + b, \qquad h = N - b$$

$$N - b > 2b$$

$$N > 3b$$

$$b < \frac{1}{3}N$$

따라서 b<1/3N 일 때 Safety 와 Liveness 모두를 만족시킬 수 있다는 결론이 나오게 된다. 이는 BFT 에서 이야기하는 3m+1 의 노드가 있을 때 m 개의 악의적인 노드가 있더라도 합의가 가능하다는 결론과 동일하다. 지금까지 일반화 증명을 통해 BFT 를 해결법을 알아보았다. 다음에서는 블록체인에서 어떤 방식으로 BFT 문제를 해결했는지 알아본다.

3.6 블록체인에서의 BFT 해결 [13,14, 23]

블록체인 중 가장 처음 제시된 비트코인의 합의 알고리즘 중 작업증명방식(Proof of Work, POW)는 타임스탬프(Timestamp)와 서명(Sign, 블록체인에서는 Key)을 통해 이 문제를 해결한다.

1) 장군은 메시지를 보내기 위해 10 분의 시간을 가진다.
(nonce 를 찾기 위한 컴퓨팅 작업)

2) 메시지는 모든 장군(이전 포함)의 메시지와 메시지를 보내기 위해 10 분을 들였다는 증거를 포함한다. 다음의 규칙이 추가 됨에 따라 중간의 비잔틴 장군이 존재하더라도 다른 장군들은 이 장군이 거짓임을 밝혀낼 수 있다. 혹은 초기에 메시지를 받은 장군이 존재하더라도 앞선 시나리오처럼 정직한 장군들이 성을 함락할 수 있게 된다.

지금까지 비트코인의 합의 알고리즘 (Proof-of-Work)을 통해 BFT 의 해결과정 또한 알아보았다. 비트코인의 합의 알고리즘은 사토시 나카모토의 이름을 딴 나카모토 컨센서스(Nakamoto Consensus)로도 불리며, 작업증명 방식에 따라 언제나 어려운 문제에 대한 답(Nonce)을 마이너(Miner)가 찾으면 블록이 생성되어 체인을 이어나가게 된다. 이렇게 만들어진 가장 긴 체인은 유효(valid)한 체인이며, 정격 체인(canonical chain)이라 부른다. 이 방식은 FLP Impossibility 를 우회하는 방법 중 Liveness over Safety 의 방법을 택한 것이며, 비트코인의 경우 Liveness 를 먼저 확보하고 6 컨펌(약 60 분)을 통해 완결성(Finality)을 가져 Safety 를 보완한다.

FLP Impossibility 를 우회하는 두 번째 방법인 ‘Safety over Liveness’의 방식으로는 BFT 계열의 합의 알고리즘이 있다. 부분 동기성 네트워크 모델인 PBFT 와 이를 응용하는 블록체인을 통해 어떻게 문제를 해결하는 지 알아보자.

4. Practical Byzantine Fault Tolerance (PBFT)

4.1. Introduction

PBFT 합의 알고리즘은 비잔틴 노드가 존재할 수 있는 비동기 분산 시스템 상황에서도 참여한 노드가 성공적으로 합의를 달성할 수 있도록 고안된 방안이다. 미구엘 카스트로(Miguel Castro)와 바바라 리스코프(Barbara Liskov) [15]가 1999 년에 제안한 이 BFT 계열 합의 알고리즘은 일정 수준의 배신자 노드가 존재하더라도 비동기 네트워크에서 합의에 도달함을 보여준다. 현재 PBFT 모델은 다양한 비허가형(permissioned) 및 허가형(permissionless) 블록체인 시스템에서 적용되고 있으며, 특히 분산 환경에서의 노드 간의 결함 감내를 보장하기 위해 사용된다.

4.1.1. Service Properties

PBFT 알고리즘은 분산 시스템에서 다른 노드들 간의 복제(Replicate)가 가능한 상태 머신(state machine)을 모델로 하며, 각 노드는 서비스 상태(service state)[1]를 유지하면서 서비스 운영(service operation)[2]을 실행한다.

기존의 상태 머신 복제(state machine replication)시스템과 마찬가지로, 노드(replica)는 두 가지 요구 사항을 따른다.

1. 노드는 결정론적(deterministic)이다. 결정론적 시스템에서는 특정 입력 값이 들어오면 항상 똑같은 과정을 거쳐 동일한 결과를 만들어낸다.
2. 노드는 동일한 상태(state)에서 시작한다.

따라서, 해당 조건에서는 모든 정직한 노드는 요청 실행 순서를 합의하기 때문에 특정 실패가 발생해도 safety 를 보장한다.

[1] *Service state: 하나의 서비스가 작업을 수행하기 위해 읽고 쓰는 데이터를 나타낸다.*

[2] *Service operation: 하나의 서비스가 어떻게 잘 운영할지 정리하는 서비스 운영 단계이다.*

4.1.2. System Model Assumption

- 네트워크에 노드가 연결되어 있는 비동기성(asynchronous) 분산 시스템을 가정하기 때문에 메시지 전달이 실패, 지연, 중복, 또는 무작위 순서가 될 수 있다.

- 비잔틴 노드가 임의적으로 행동할 수 있는 Byzantine failure 모델을 가정한다. 또한, 독립적인 노드의 실패 가능성을 열어 두어, 특정 노드의 실패가 개별 노드의 직접적인 영향을 주지 않도록 한다. 즉, 한 노드가 악의적인 공격으로 입은 피해가 다른 노드의 영향을 주어서는 안되기 때문에 서로 다른 서비스 코드(service code)와 운영 시스템(operating system)을 실행시켜야 하며, 루트 패스워드(root password)와 관리자 계정 또한 모두 서로 달라야 한다.

- Replicated service 에 심각한 장애를 일으키기 위해 아주 강력한 비잔틴 공격자의 상황을 가정하여 정직한 노드의 시스템을 지연시킨다. 단, 정직한 노드를 무기한 지연 시키는 것은 불가능하며 정직한 노드의 유효한 서명을 위조할 수는 없다.

4.1.3 Cryptography

암호기법을 통해서 spoofing 및 replay 공격을 방지하고 악의적인 메시지를 구별한다. 메시지에는 퍼블릭키 서명(Public-key signatures), 메시지 인증 코드(MAC), 충돌 회피 해시 함수(CRHF)에서 나온 메시지 다이제스트(message digest)[3]를 포함한다. 전체 노드는 퍼블릭키 서명을 통해 해당 메시지를 인증한다. 노드 i가 서명한 메시지 m은 $< m >_{\sigma_i}$으로, 메시지 다이제스트는 $\mathrm{D}(m)$으로 표기한다.

---

[3] *메시지 다이제스트(Message Digest)*

메시지 다이제스트는 고정된 크기의 메시지로, 해쉬함수를 이용해서 생성된다. 해쉬함수가 안전성이 보장된다는 가정하에서, 메시지 다이제스트 서명은 메시지 자체에 서명을 하는 것과 동일한 보안 서비스를 제공한다.
(http://ko.security.wikidok.net/wp-d/575aa262d16e73173de7a3ed/View)

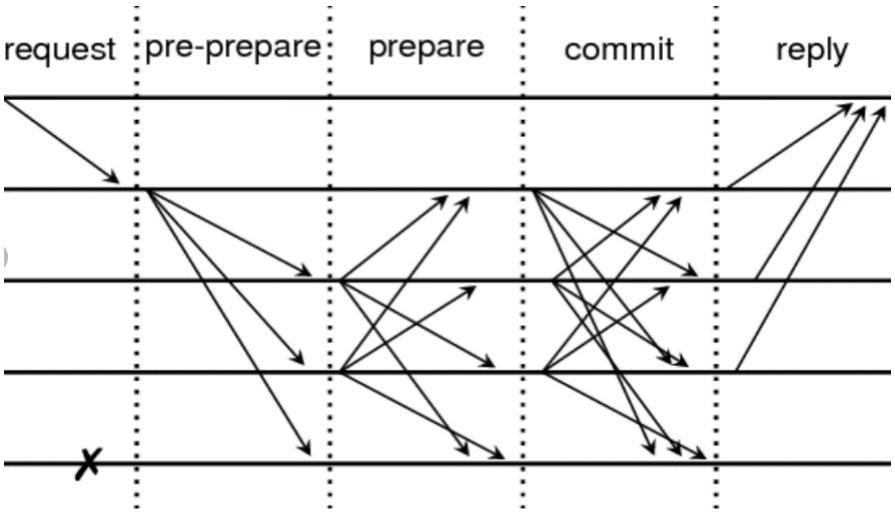
4.2 Algorithm Properties

Safety: 모든 정직한 노드는 실패가 발생하더라도 메시지 요청 실행 순서를 합의한다.

Liveness: 리더 primary 노드의 문제가 생기더라도 시스템은 멈추지 않는다.

PBFT 메시지 합의 과정은 크게 Pre-Prepare, Prepare, Commit [17]순으로 이어진다. 클라이언트의 요청이 성공적으로 실행이 되면 답변(reply)을 받게 된다. PBFT 알고리즘의 대략적인 진행 방향은 다음과 같다.

1) 클라이언트가 서비스 운영(service operation)을 호출하기 위해 primary 노드에게 요청을 전송한다.
2) Primary 노드는 해당 클라이언트의 메시지 요청을 수집하고 정렬하여 나머지 노드(backups)에게 전송한다.
3) 각 노드는 해당 요청사항을 실행하여 검증 절차를 거친다. 모든 검증 작업이 끝나면 해당 메시지에 대한 답변을 클라이언트에게 전송한다
4) 클라이언트는 f+1 개 이상의 동일한 결과를 서로 다른 노드로부터 전달 받으면 해당 요청사항이 문제없이 처리된 것을 확인한다.



〈그림 4.1. PBFT Consensus Algorithm [16]〉

본격적으로 합의 알고리즘의 진행 방향을 알아보기 전에 몇 가지 개념을 다시 한 번 정의하고자 한다.

v	뷰 번호(view number)
n	순서 번호(sequence number)
σ	서명(signature)
m	요청 메시지(requested message)
p	Primary 리더 노드 (primary replica)
$< m > \sigma_i$	노드 i 가 서명한 메시지
D(m)	메시지 다이제스트 m

〈표 4.1. PBFT 용어 설명〉

4.2.1. Request

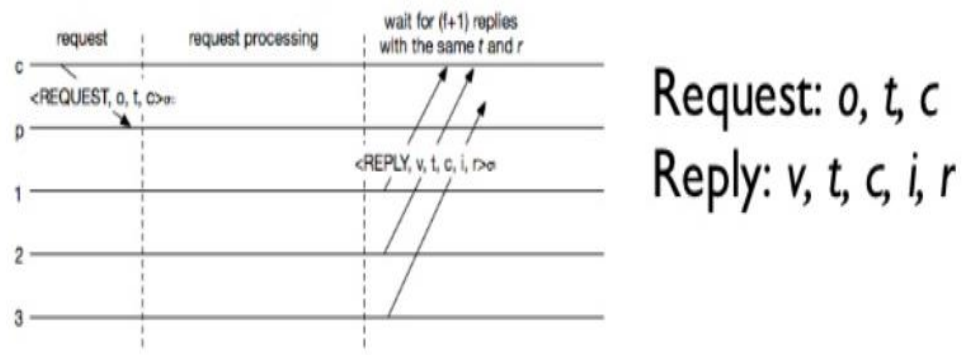
클라이언트는 $< \text{REQUEST}, o, t, c > \sigma_c$ 형태로 primary p 에게 메시지를 전송한다. Primary 가 클라이언트 요청 m을 수신하면, 이를 정렬하여 나머지 백업 노드들에게 전송한다. 그러나 프로토콜에서 처리되는 메시지의 수가 기준치를 초과하면 primary 노드는 프로토콜을 즉각적으로 실행하지 못한다. 따라서, 이 경우에는, 메시지 요청을 버퍼(buffer)시켜서 뒤이어서 전송한다. 이를 통해 메시지 트래픽 및 CPU 오버헤드를 줄일 수 있다.

4.2.2. The Client

메시지 요청이 시작되는 클라이언트(client)의 관점에서 일련의 합의 과정을 알아보자. 먼저, 클라이언트 c 는 $< \text{REQUEST}, o, t, c > \sigma_c$ 메시지를 primary 에게 전송하여 state machine operation 의 실행을 요청한다. 요청을 받은 primary 는 이를 정리해서 노드(backup)에게 전송한다. 각 노드는 전달 받은 요청(request)에 대한 답변을

클라이언트에게 직접 전달한다. 답변의 형태는 $< \text{REPLY}, v, t, c, i, r > \sigma_i$ 이며, 의미는 다음과 같다.

- v는 현재 뷰 번호(view number)이다. 클라이언트는 뷰(view)를 확인하여 현재 primary 가 누구인지 확인한다. 뷰(view)는 쉽게 말해, 리더 노드인 primary 와 나머지 backup 노드가 한 번 활동하는 기간이다. 뷰 번호(View number)를 확인하여 클라이언트는 현재 primary 에게 요청(request) 메시지를 보내게 되는데, 이때 전송 방식은 점대점(point-to-point)[4]을 사용한다.
- t 는 요청에 대한 타임스탬프, i 는 노드의 숫자를 나타내며, r 은 요청한 operation 에 대한 실행 결과이다.



〈그림 4.2. Client Request & Reply [16]〉

클라이언트는 유효한 서명 및 동일한 t와 r이 담긴 메시지 답변을 각각의 노드에게서 f+1 개 일 때까지 기다린다. 그 전까지는 결과값 r을 수락해서는 안된다.

만약 클라이언트가 답변을 제때 받지 못했다면, 이번엔 해당 요청을 모든 노드에게 전송한다. 요청은 이미 처리가 된 상황이고 단순히 네트워크 문제로 메시지 전달이 지연된

[4] *Point to point: 한 명의 송신자가 한 명의 수신자에게 메시지를 보내는 방식으로 계속해서 대화를 나누며 빠른 응답을 필요로 하는 경우에 주로 사용된다.*

경우라면 노드는 해당 요청에 대한 답변을 재전송 해야한다. 따라서, 모든 노드는 각각의 클라이언트에게 보낸 마지막 답변 메시지(reply message)를 항상 기억해야 한다.

반대로, primary 가 요청(request)을 백업 노드에게 전송하지 않아서, 메시지가 처리되지 않았을 수 있다. 이때, 다수의 노드들은 현재 primary 가 악의적인 노드라 판단하고 view change 를 발생시킨다. View change 는 리더인 primary 를 교체하는 작업으로 〈4.5. view change〉에서 더 자세히 알아본다.

4.3.Normal Case Operation

Pre-prepare 및 prepare 는 요청 순서를 제안한 primary 가 악의적인 노드일지라도 동일한 뷰(view)에서 보내진 요청을 정렬화 시키기 위한 단계이다. Prepare 및 commit 은 커밋된 요청을 뷰(view) 전반에 걸쳐서 순서를 정돈하는 단계이다.

4.3.1. Pre-prepare 단계

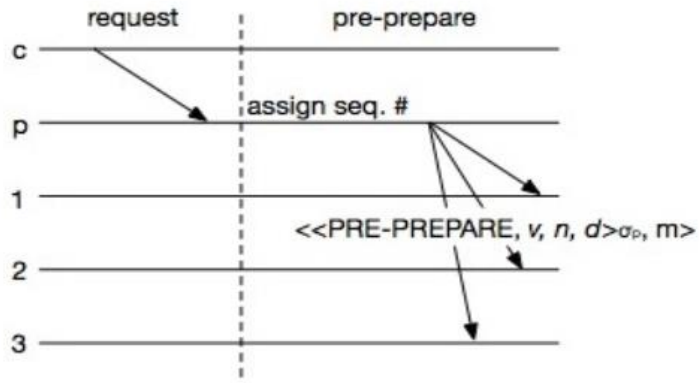
클라이언트로부터 요청이 도착하면 가장 먼저 primary 는 유효한 요청에 sequence number n을 부여하는데, 이는 요청 순서를 의미한다. Primary 는 pre-prepare 메시지를 나머지 노드에게 전송하고 자신의 log 에 해당 메시지를 저장한다. 이때, 메시지 m은 piggyback[5](같이 태워서 보냄) 형식으로 보내게 되어 데이터 전송 효율성을 높인다.

Pre-prepare 메시지는 $\ll \text{PRE} - \text{PREPARE}, v, n, d > \sigma_p, m >$ 형태로 전송된다. 해당 메시지의 의미는 다음과 같다.

- v: 메시지가 전송되는 view
- n: 메시지 요청 순서(sequence number)

[5] *Piggyback 이란?*
수신자는 수신 받은 데이터에 대한 확인(Acknowledgement)를 즉각적으로 보내지 않고, 우선 기다리고 있다가 전송할 데이터가 있는 경우에 함께 보낸다. 이렇게 하는 이유는 수식자 측에서 데이터를 잘 받았다는 확인서(Ack)를 보낼 때 일정량의 대역폭을 사용하게 된다. 수신자는 송신자에게 데이터를 전송할 때, 나머지 데이터도 함께 태워서 보내면 이런 비효율성을 줄일 수 있다.

- d: 메시지 m의 다이제스트(digest)
- σ_p: 메시지의 유효성을 증명하는 primary 의 서명
- m: 클라이언트가 요청한 메시지



〈그림 4.3. Pre-prepare Phase [16]〉

이제 노드(backup)들은 pre-prepare 메시지의 진위여부를 확인 후 수락하는 작업에 착수한다. 메시지의 참인 성립 조건은 다음과 같다.

- v, n, d, σ 모두 유효해야 한다.
- 다른 다이제스트가 포함된 view v와 sequence number n을 수신하지 않아야 한다.
- Pre-prepare 메시지의 sequence number 는 low water mark h와 high water mark H 사이에 존재해야 한다.

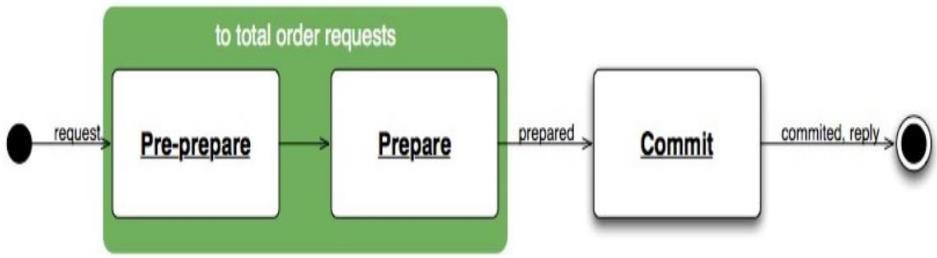
특히, 마지막 (h<n H) 조건은 악의적인 primary 노드가 가장 큰 수를 선택함으로써 sequence number 의 공간을 모두 차지하는 것을 방지한다. (4.4. Checkpoint 부분에서 (h<n H) 조건에 대해서 더 자세히 알아본다.

4.3.2. Prepare 단계

Backup 노드 i가 $\ll \text{PRE}-\text{PREPARE}, v, n, d > \sigma_p, m >$ 메시지를 수락하면 prepare 단계에 진입하게 된다. 이제 $< \text{PREPARE}, v, n, d, i > \sigma_i$ 메시지를 다른 모든 노드들에게 전송하고 자신의 log 에 저장한다. Primary 를 포함한 노드들은 다음 조건을 확인하면 해당 prepare 메시지를 수락하고 자신의 log 에 저장한다.

- 메시지의 서명이 유효하다.
- 메시지의 view number 와 노드의 현재 view 가 일치하다.
- Sequence number 는 h와 H사이다.

Pre-prepare 와 prepare 단계에서는 동일한 view 에서 보낸 메시지 순서를 정렬한다. 이 두 단계를 통해 정직한 노드들은 동일한 view 내에 메시지에 대한 전체 순서를 합의하게 된다. 노드들은 동일한 뷰(view), 순서 번호(sequence number), 다이제스트(digest)를 가지고 있는지 확인하면서 prepare 메시지와 pre-prepare 메시지가 일치하는지 검증한다. 서로 다른 노드로부터 pre-prepare 메시지와 일치하는 prepare 메시지 2f 개를 수집하면 검증 절차가 완료되고, 이를 **"prepared"**된 상태라 부른다.



〈그림 4.4. Pre-prepare & Prepare(Prepared)〉

만약 노드 i 입장에서 $prepared(m,v,n,i)$ 메시지가 참이라면 정직한 노드 j 에게는 $prepared(m`,v,n,j)$은 거짓이며, 따라서 D(m`)!=D(m) 조건은 성립된다. 이 조건이 참인 이유는 $prepared(m,v,n,i)$와 R=3f+1 을 통해 최소 f+1 개의 정직한 노드는 sequence number n 을 가진 view v내에 메시지 m에 해당하는 pre-prepare 또는 prepare 를 보냈기 때문이다.

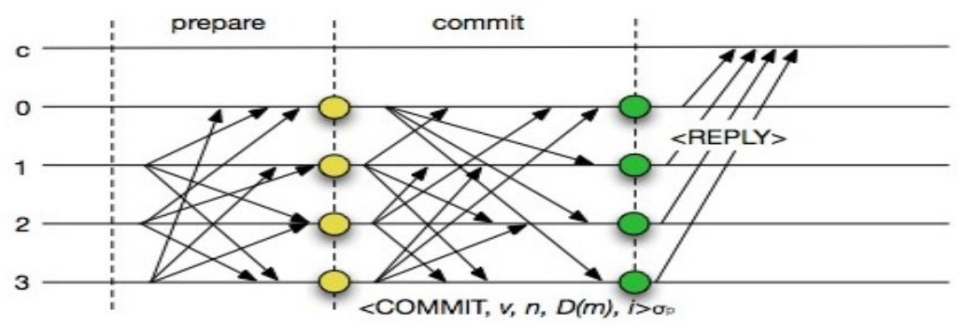
4.3.3. Commit Phase

노드 i는 $prepared(m,v,n,i)$메시지가 참이 되면 $< COMMIT, v, n, D(m), i > \sigma_i$ 메시지를 노드들에게 전송한다. 각 노드들은 다음 조건을 확인하면 해당 commit 메시지를 수락하고 자신의 log 에 저장한다.

- 메시지의 서명이 유효하다.
- 메시지의 view number 와 노드의 현재 view 가 일치하다.
- Sequence number 는 h와 H사이다.

Commit 단계에서는 committed 와 committed-local 이 존재한다.

- $\boldsymbol{committed(m,v,n)}$: f+1 개의 정직한 노드가 존재하는 상황에서 $prepared(m,v,n,i)$가 참인 경우
- $\boldsymbol{committed-local(m,v,n,i)}$: $prepared(m,v,n,i)$ 가 참이고 노드 i 가 자신의 메시지를 포함해서 다른 노드로부터 pre-prepare m 과 일치하는 2f+1 개의 commit 메시지를 받은 경우



〈그림. 4.5. Commit Phase〉

바로 이 부분에서 commit 단계가 필요한 이유다. 노드 i는 다른 노드로부터 pre-prepare m 과 일치하는 2f+1 개의 commit 메시지를 받는다고 했다. 그렇다면 왜 prepare 메시지가 아닌 pre-prepare 메시지와 대조 작업을 하는 것일까? 아래 예를 통해 확인해보자.

2f+1 개의 prepare 메시지를 받지 못한 (정직한) 노드가 있다고 가정해보자. 네트워크 연결이 끊어 졌거나 잠시 동안 오프라인 상태여서 메시지를 못 받았을 것이다. 따라서, 이들은 당연히 prepared 단계를 시작하지 못하였다. 그러나 약간의 시간이 지난 후, 2f+1 개의 노드가 commit 메시지를 전파 했다는 사실을 듣는다면, 그때서야 (m, v, n, i)을 커밋해도 된다는 확신을 하게 된다. 즉, 정직한 노드라면 내부적으로 커밋한 요청 메시지의 순서 번호(sequence number)를 합의하게 된다.

결론적으로, 메시지 전달은 순서대로 진행되지 않기 때문에 노드는 요청 메시지를 뒤죽박죽(out-of-control)으로 커밋(commit)할 수 있다. 그럼에도 메시지 처리 문제가 발생하지 않는 이유는 해당 요청 메시지가 실행 되기 전까지, 각 노드는 pre-prepare, prepare 그리고 commit 단계마다 메시지를 자신의 log 에 저장 해두기 때문이다.

4.4 Checkpoint (Garbage Collection)

PBFT 알고리즘은 safety 를 보장하기 위해 노드가 메시지를 자신의 log 에 저장하도록 한다. 최소 f+1 개의 정직한 노드가 요청을 실행했다는 사실을 알기 전 까지는 메시지를 보관해야 한다. 하지만, 지속적으로 쌓여가는 메시지를 적절히 관리하지 않으면 노드의 log 는 비대해지고 결국 비율효적인 상태가 된다. 이런 문제를 해결하기 위해서 일정 기간을 두고 **안정된 체크포인트**(stable checkpoint)를 만들어 낸다. **체크포인트**(Checkpoint)**란** 노드의 log 에 저장해 두었던 메시지를 정리하여 폐기하는 단계로써, 검증이 안정적으로 완료되면 stable checkpoint 가 만들어진다.

그럼 이제 구체적으로 체크포인트가 어떻게 진행되는지 알아보자. 먼저, 노드 i는 체크포인트를 만들고 〈$\text{CHECKPOINT}, n, d, i > \sigma_i$ 메시지를 나머지 노드에게 전송한다. 해당 메시지의 의미는 다음과 같다.

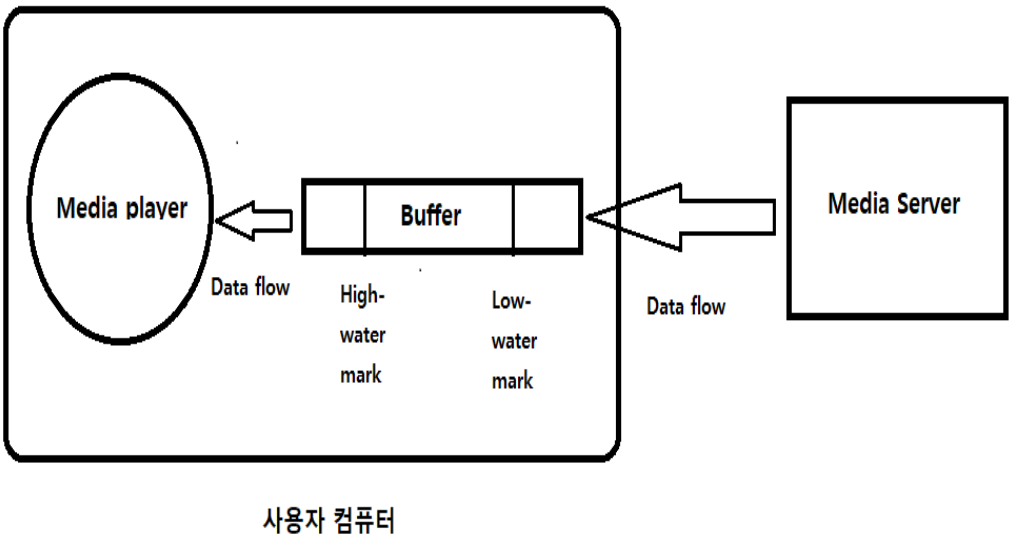
- n은 상태(state)에서 실행 된 마지막 메시지의 순서 번호(sequence number)이며
- d는 상태(state)의 다이제스트이다.

다른 노드가 서명한 동일한 다이제스트 d를 가진 sequence number n에 해당하는 2f+1 개의 체크포인트 메시지를 모으기 전까지는 자신의 log 에 해당 메시지를 모아둔다. 유효한 체크포인트 메시지 2f+1 개가 모이면, 이 체크포인트는 안전하다고 증명(proof)된다. 안전성이 증명된 체크포인트(stable checkpoint)에서는 노드는 자신의 log 의 sequence number 보다 작거나 같은 pre-prepare, prepare, commit 메시지를 모두 버린다.

또한, 체크포인트에는 메시지를 얼마나 받아들이고 제한할지를 결정하는 low water mark h 와 high water mark H 가 존재한다. 매번 operation 을 실행한 후 메시지 처리 과정을 진행하는 것은 비용적으로 비효율적이다. 따라서, 체크포인트(state snapshot 상태)는 일정 주기를 두고 생성된다. Low water mark 는 마지막 stable checkpoint 의 sequence number 에 해당하여, 메시지의 sequence number 가 H 에 도달하게 되면,

체크포인트 작업을 시작한다. High-water mark와 low-water mark의 직관적인 이해를 돕기 위해 데이터 처리가 어떻게 이루어지는지 알아본다.

<데이터 전송에서 High-water mark 와 Low-water mark 에 대한 이해>[18]
High-water mark 와 low-water mark 는 데이터 처리를 원활하게 하기 위한 장치이다. Buffer 는 일부의 데이터를 잠깐 저장하는 제한된 공간이다. 아래의 그림에서 media server 는 buffer 의 water-mark 를 통해 얼마만큼의 데이터를 전송할지 결정한다. Buffer 가 low-water mark 보다 데이터를 덜 갖고 있으면, media player 는 더 많은 데이터를 전송해달라는 신호를 보내고, 반대로 데이터가 high-water mark 를 넘으면, media player 는 데이터를 그만 보내라는 신호를 보내게 된다.



〈그림. 4.6. Low-water mark & High-water mark〉

4.5. View Change

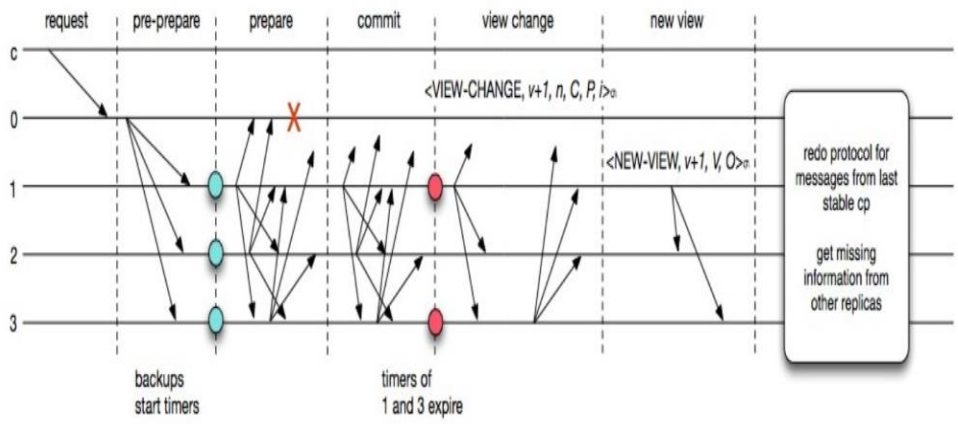
리더 노드가 요청 메시지를 멀티캐스트하지 않고 일정 시간이 지나게 되면 리더 변경 절차인 view change 가 발생한다. View change 는 liveness 를 확보하는 단계로써, primary 노드가 비잔틴일 경우에도 시스템이 계속해서 작동하도록 하는 단계이다. 즉, primary 의 메시지를 기다리는 backup 노드들은 무기한 기다리지 않으며, 타임아웃되면 리더를 변경하는 〈VIEW-CHANGE〉를 발생시킨다.

Backup 노드의 타이머 조건은 다음과 같다.

- 타이머(Timer)가 시작되는 조건
  -이미 다른 타이머가 작동하지 않고 메시지 요청을 수신했을 때
- 타이머(Timer)를 정지하는 조건
  - 메시지 요청 실행을 더 이상 기다리지 않을 때
- 타이머(Timer)를 재시작하는 조건
  - 특정 시점을 기다리면서 다른 요청을 실행할 때

특정 view 내의 backup 노드 i의 타이머가 만료되면, view *v+1* 로 이동하기 위한 view change 가 시작된다. View change 가 시작되면, 노드는 checkpoint, view-change, new-view message 를 제외한 다른 메시지를 일절 받지 않으며, $< \mathrm{VIEW-CHANGE}, v+1, n, C, P, i > \sigma_i$ 메시지를 모든 노드에게 전송한다. 해당 메시지의 의미는 다음과 같다.

- $v+1$: 새로운 view 번호
- n: 노드 i가 알고 있는 가장 최신의 stable checkpoint s의 순서 번호(sequence number)
- C: s가 올바른 checkpoint 인지 증명하는 2f+1 개의 유효한 체크포인트 메시지
- P: 노드 i의 prepared 메시지 중에서 n 보다 큰 sequence number 를 가진 요청 메시지 집합



〈그림 4.7. View Change & New-View〉

v+1 에서 primary p 가 다른 노드로부터 2f 개의 유효한 view-change 메시지를 전달받으면, $< \text{NEW} - \text{VIEW}, v+1, V, O >_{\sigma_P}$ 메시지를 다른 모든 노드에게 전송한다. 해당 메시지의 의미는 다음과 같다.

- $v+1$: 새로운 view 번호
- V : primary 가 받은 유효한 view-change 메시지 + primary 가 보낸 view-change 메시지(v+1)
- O: pre-prepare 메시지 집합 (피기백(piggyback) 요청은 제외)

앞서서 보았듯이, 2f+1 개 이상의 노드가 메시지를 검증하면 나머지 노드를 기다리지 않기 때문에 일부 노드는 요청 메시지 m 또는 stable checkpoint 가 다른 노드와 정확히 일치하지 않을 수 있다. 따라서, view-change 가 일어나서 새롭게 교체된 primary 가 유효한 작업을 시행하려면 backup 노드들과 메시지 반영 상태를 sync 할 필요가 있다. 이때 필요한 작업이 유효한 집합 O 의 계산이다. Pre-prepare 메시지 집합인 O 가 만들어지는 과정은 다음과 같다.

먼저, V 의 메시지들 중 가장 최근의 stable checkpoint 의 순서 번호(sequence number)를 min-s 라 하고, V의 prepare 메시지들 중 가장 높은 순서 번호(sequence

number)를 max-s 라 한다. 모든 sequence number n 이 min-s< n < max-s 에 대해, primary 는 새로운 pre-prepare 메시지를 만든다. 이 경우 두 가지의 경우를 생각해 볼 수 있다.

- Case 1: V내의 P가 sequence number n 인 메시지 집합이 적어도 하나인 경우, $< \text{PRE} - \text{PREPARE}, v+1, n, d > \sigma_p$ 메시지를 생성한다.
- Case 2: 메시지 집합이 하나도 없는 경우, $< \text{PRE} - \text{PREPARE}, \text{v} + 1, \text{n}, \text{d}^{null} > \sigma_p$ 메시지를 생성한다.

이제 Primary는 O의 pre-prepare 메시지를 자신의 log에 붙이게 되면, v+1 로 진입하게 되어 메시지를 다시 수신 받게 된다. Backup 노드들이 v+1 의 NEW-VIEW 메시지를 받아들이는 조건은 다음과 같다.

- 서명이 유효하다.
- 포함되어 있는 view-change 메시지가 유효하다.
- 집합 O의 계산이 유효하다.

검증을 완료한 노드들은 NEW-VIEW 를 받아 들이고 normal-case operation 대로 다시 진행 하게 된다.

5. Use Case: Tendermint

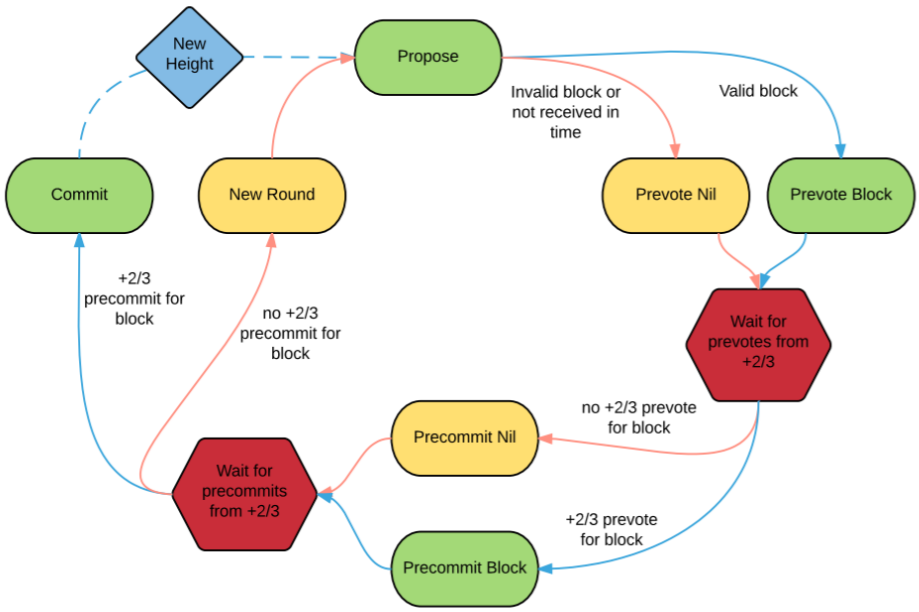
PBFT 가 어떤 방식으로 합의에 도달하는지 증명을 통해 알아보았다. 블록체인에서는 PBFT 를 어떻게 응용하여 사용하는지 대표적인 코스모스(Cosmos)프로젝트의 텐더민트(Tendermint) [19,20]를 통해 알아보자. 텐더민트는 코스모스 네트워크(Cosmos Network)에서 사용하는 합의 엔진으로써, 전통적인 PBFT 합의 프로세스를 블록체인에 맞게 적용한 대표적인 사례이다. PBFT 와 Tendermint 모두 최대 1/3 개의 비잔틴 노드가 존재하는 상황에서도 성공적으로 합의에 도달한다.

텐더민트의 전체 합의 프로세스는 PBFT 와 유사하지만 약간의 용어 차이가 존재함으로 알아보자.

PBFT	Tendermint
Primary	Proposer
Replica	Validator
Pre-prepare phase	Propose step
Prepare phase	Prevote step
Commit phase	Precommit step
View change	Round change

〈표 5.1. Terminology of PBFT & Tendermint〉

텐더민트 투표 프로세스(Voting Process)는 〈그림 5.1〉과 같은 방식으로 이루어진다.



〈그림 5.1.Tendermint voting process〉

텐더민트는 비잔틴 결함을 감내하기 위해 3f+1 규칙에 따라 최소 4 명의 검증인이 필요하다. 각 검증인은 퍼블릭키를 통해 신원을 확인하며, 모든 제안과 투표는 각 검증인의 프라이빗키를 통해 서명이 이루어지는데, 이때 디지털 서명은 비대칭 암호키쌍(asymmetric cryptographic key-pair)을 사용한다. 이를 통해서, 제안과 투표의 유효성을 검증 할 수 있다.

합의(consensus)는 라운드 0 에서 시작하며 검증인 리스트 L에서의 첫 검증인이 첫 번째 블록 제안자가 된다. 라운드 별 결과는 항상 commit 을 하거나 다음 라운드로 이동하는 결정(decision) 두 경우로 나뉜다. 여러 번의 라운드를 통해 네트워크가 비동기이거나 검증인에게 문제가 생기는 경우에도 합의에 도달할 수 있는 기회가 제공한다. 여러 번의 라운드가 발생할 수 있는 상황은 다음과 같다.

- 네트워크가 지연될 때
- 지정된 블록 제안자가 온라인 상태가 아닐 때

- 지정된 블록 제안자가 생성한 블록이 유효하지 않을 때
- 지정된 블록 제안자가 생성한 블록을 제 때 전송하지 않을 때
- 제안된 블록은 유효하나 검증인들이 pre-commit 단계에 진입할 때 2/3 이상의 prevote 를 제 때 수신하지 못할 때
- 제안된 블록도 유효하고 충분한 숫자의 검증인이 2/3 이상의 prevote 하였지만 2/3 이상의 pre-commit 을 수신하지 못했을 때

이런 문제를 극복하기 위해 텐더민트는 매 라운드마다 새로운 블록 제안자가 선택된다. 블록 제안자 선정 방식은 〈5.1.4 Leader Selection Procedure〉에서 좀 더 자세히 알아본다

5.1. Consensus

1. 제안(Proposals): 각 라운드마다 정직한 제안자는 새 블록을 생성한다. 제안이 제때 도착하지 못하면, 해당 제안자의 순번은 건너 뛴다.

2. 투표(Votes): 최적의 비잔틴 결함 내성을 보장하기 위해 투표는 *pre-vote* 와 *pre-commit* 두 단계를 거친다. 동일한 라운드(round)에서 2/3 이상의 지분을 가진 검증인(validator)이 pre-commit 하면 블록은 커밋(commit)된다.

3. 잠금(Locks): 텐더민트는 두 명의 검증인이 동일한 높이에서 서로 다른 블록을 커밋하지 못하도록 잠금 매커니즘(Locking mechanism)을 활용한다. 잠금 매커니즘은 동일한 높이에서 이전 pre-votes 와 pre-commits 에 따라서, 다음 라운드에서 검증인이 pre-vote 또는 pre-commit 할지 결정하는 방식이다.

5.1.1 Proposal

한 라운드는 Propose〉Prevote〉Precommit〉순서로 진행되며, 동일한 높이의 블록 제안(proposal)이 매 라운드의 시작이다. 라운드 별 제안자는 Mempool(블록이 포함되기 전의 트랜잭션 저장소)에서 최근 수신 된 트랜잭션을 가져와서, 블록을 생성하고, 서명된

제안자 메시지(ProposalMsg)를 전송한다. (만약 블록 제안자가 비잔틴이라면, 다른 검증인에게 다른 제안을 전송하게 된다.)

결정론적인 라운드 로빈(round robin)방식을 통해 블록 제안자 순서가 정해지기 때문에 각 라운드 별 단 한 명의 검증인만이 유효하다. 만약 제안이 낮은 숫자의 라운드에서 수신되거나 비잔틴 제안자에서 온다면, 해당 제안은 거부된다. 텐더민트는 투표와 잠금 매커니즘을 통해 safety 를 확보하며, 제안자 순환을 통해 liveness 를 유지한다. 따라서, 한 명의 제안자가 트랜잭션을 제 때 처리하지 않으면 다른 검증인이 뒤이어서 처리한다.

5.1.2 Votes

투표는 두 단계(Pre-vote, Pre-commit)를 거쳐서 진행되며, 성공적으로 2/3 이상의 검증인(stake, 지분)이 동일 라운드의 같은 높이의 블록을 pre-commit 하면 해당 블록은 성공적으로 commit 된다. 비잔틴 검증인이 존재할 수 있는 비동기 환경에서는 단일 투표 단계는 safety 를 충분히 보장하지 못한다. 단일 투표 단계에서는 검증인들은 제안이 무엇인지 서로 알 수 있다. 그러나 비잔틴 결함을 극복하기 위해서는, 검증인은 다른 검증인들이 해당 제안에 대해 알고 있는 사실들을 서로에게 알려줘야 한다. 쉽게 말해서, 두 번째 투표 단계를 통해서 검증인들은 첫 번째 투표 결과를 충분히 알 수 있어 악의적인 검증인의 행동을 알아차릴 수 있다.

Pre-vote 에서의 블록 투표는 블록을 커밋하기 위한 준비 단계이다. Pre-vote 에서 nil 에 투표는 다음 라운드의 이동을 의미한다. 즉, 2/3 이상의 검증인이 제안을 pre-vote 하게 되면 올바른 방향으로 진행됨을 알 수 있다. 한 라운드에서 하나의 블록이 2/3 이상의 pre-vote 가 이루어졌으면 이를 *polka* 라 부르며, 반대로 nil 에 투표가 진행되었다면 *nil-polka* 라 한다. (현재는 자주 언급되는 개념은 아니지만 해당 글에서는 polka 의미를 사용한다.) 만약 검증인이 ***ProposalTimeout***내에 유효한 제안을 수신 받지 못하면, pre-vote 에는 ***nil***을 선택하게 된다.

성공적으로 검증인이 polka 를 받으면, 해당 블록의 pre-commit 투표를 서명하고 전파하는 단계에 돌입한다. 그러나 종종 네트워크 비동기성 때문에, 특정 검증인은 polka 를 받지 못하는 상황이 있을 수 있다. 이 경우 검증인은 해당 블록을 pre-

commit 으로 서명할 수 없기에 pre-commit 투표는 nil 에 서명한다. 바꿔 말하면, polka 를 받지 못한 검증인이 pre-commit 을 서명하면, 이는 비잔틴 행위이다.

이제 검증인이 한 개 블록의 2/3 이상의 pre-commit 을 받으면, 해당 블록은 커밋(commit)되고, 다음 높이에서 새로운 라운드 0 이 시작된다. 마찬가지로, 검증인이 2/3 이상의 pre-commit nil 을 수신하면 다시 블록을 만들기 위해 다음 라운드로 이동한다.

5.1.3. Locks

동일한 블록 높이에서에서 서로 다른 두 개의 라운드 및 블록이 만들어 지는 상황을 피하기 위해 텐더민트는 잠금 매커니즘(Locking mechanism)을 활용한다. 2/3 이상의 pre-vote 를 의미하는 polka 를 통해 pre-commit 이 정당화 된다. 잠금 규칙에는 Prevote-the-Lock 과 Unlock-on-Polka 총 2 가지가 있다.

- Prevote-the-Lock : 특정 블록에 잠긴 검증인들은 반드시 pre-vote 해야하며, 제안자인 경우 블록을 반드시 제안해야 한다. 이를 통해 검증인들이 첫 라운드에서 동일한 높이에서 한 개의 블록에 대해 pre-commit 하고 다음 라운드에서 다른 블록의 polka 를 외치는 것을 막기 위함이다. (Safety 와 연관된 사항)

- Unlock-on-Polka: 검증인들은 자신들이 잠금된 라운드보다 높은 라운드에서 polka 가 이루어 졌다면, 잠금을 해제하고(unlock) 새로운 블록에서 pre-commit 을 한다. 이를 통해 검증인들이 어디 한 곳에 갇혀 합의에 도달하지 못하는 상황을 막는다. 쉽게 말해, 시스템이 멈추지 않고 계속 동작하도록 한다. (Liveness 와 연관된 사항)

검증인이 매 블록 높이에서 round-1 에 nil 을 투표해서 잠금 됐을 경우, 새로운 블록 높이에서 polka 를 보기 전까지 pre-commit 할 수 없음을 Unlock-on-Polka 를 통해 알 수 있다. 잠금 매커니즘의 직관적인 이해를 돕기 위해 검증인 예시를 통해 알아보자.

A, B, C, D 총 4 명의 검증인이 있으며 특정 라운드 R 에서 블록 X 에 대한 제안을 가정한다. 블록 X의 polka 가 이루어졌지만 A 는 확인하지 못해 pre-commit nil 을 한다. (물론 나머지 B,C,D 검증인은 pre-commit 한 상태)

이제 비동기 네트워크로 단지 지연이 발생한 상황에서 모든 검증인의 pre-commit 을 확인한 사람은 D 이며, B 와 C 는 D 의 pre-commit 을 보지 못했다고 가정해보자. (B 와 C 는 서로 pre-commit 한 사실과 A 의 pre-commit nil 을 확인한 상태) D 는 2/3 가 블록을 pre-commit 한 사실을 확인했음으로 블록을 커밋(commit)하지만 나머지 B 와 C 는 합의에 필요한 정족수를 채우지 못했음으로 R+1 로 이동한다. 검증인 D 는 이미 블록 X에 커밋을 했기 때문에 R+1 라운드에는 새로운 제안자가 블록 Y를 제안하고 투표가 진행 될 경우, safety 가 훼손된다. (누구도 비잔틴이 아닌 상황)

검증인 D 의 경우 나머지 검증인 B 와 C 의 pre-commit 을 보고 블록 X를 커밋 했기 때문에, 잠금 매커니즘을 통해 이미 pre-commit 한 검증인 D 는 블록 X를 유지한다. 특정 라운드에 2/3 이상의 pre-commit 된 블록이 있을 경우, 해당 블록은 잠금된다는 사실을 잊지 말자. 즉, 더 높은 라운드에서 새로운 블록에 유효한 polka 를 만들어 낼 수 없다. 이를 Prevote-the-Lock 이라 한다.

그러나 liveness 를 희생하지 않기 위해서는 잠금을 해제하는 과정이 필요하다. 특정 라운드에 검증인 A 와 B 가 블록 X 에 pre-commit 했고 검증인 C 와 D 가 pre-commit nil 을 해서 투표가 양분된 상황이라고 가정해보자. 이 경우 모든 검증인이 다음 라운드로 이동하고 C 와 D 가 pre-vote 하기 위한 블록 Y 가 제안된다. (A 와 B 는 이미 블록 X 에 pre-commit 했음으로 잠금 된 상태) 이제 검증인 A 가 비잔틴 노드여서, 블록 Y (block X 에 잠금 되었지만)에 또 pre-vote 를 해서 polka 가 만들어 졌다고 가정해보자. 이제, 검증인 B 는 polka 를 보지 못한 상태여서 pre-commit nil 을 한 상황이지만 검증인 C 와 D 는 블록 Y 에 pre-commit 한 상황이다. (악의적인 검증인 A 는 off-line 으로 전환한 상태) 비잔틴 합의에 도달하지 못하였음으로 다음 라운드로 이동한다. 검증인 B 는 여전히

블록 X 에 잠금 되어 있고 , 검증인 C 와 D 는 블록 Y 에 잠금되어 있다. 비잔틴 검증인인 A 는 offline 상태이기 때문에 절대 합의(polka)에 도달하지 못한다. 이런 딜레마를 해결하기 위해 검증인 B(블록 X 에 잠금된 상태)가 블록 Y 의 polka 를 확인했다면, 잠금을 해제하고 블록 Y에 pre-commit 할 수 있다. 이것이 Unlock-on-Polka가 필요한 이유다.

5.1.4 Leader Selection Procedure 〈21〉

앞서서 소개한 블록 제안자 선택 절차에 대해서 알아본다. 텐더민트에서는 각 제안자는 결정론적 [6] 으로 선택이 이루어지며 선택의 횟수는 아톰의 총 본딩 물량(자가 물량+위임자로부터 받은 물량)에 좌우된다. 예를 들어, 모든 검증인의 아톰 본딩 총합이 100 아톰이라고 가정해보자. 이때 A 라는 검증인의 총 본딩 물량이 10 아톰 이라면 10% 확률로 다음 블록 제안자가 된다.

텐더민트 검증인은 라운드 로빈 방식으로 블록 제안자로 선택된다. 라운드 로빈 방식이란 리스트의 최상위 순번에서 최하위 순번으로 차례대로 순번이 돌아간다는 뜻이다. 마지막 순번이 끝나면 다시 제일 위 상위 순번에게 순서가 돌아간다. 텐더민트 제안자 선택에서 이 순번을 정하는 핵심 요소가 바로 투표력(Voting power)이다. 첫 블록 라운드가 시작되면 검증인의 총 투표력(자가 물량+위임자로부터 스테이킹 받은 물량)과 아쿰(Accum)이 0 인 상태서 시작된다. 제안자 선택에서는 투표력을 Accum 으로 표시하여 이해를 돕고 있다. 각 라운드가 진행 될 때마다 *IncrementAccum* 기능을 통해 검증인의 Accum 이 자신의 투표력만큼 상승한다. 이때, 각 라운드 별 투표력이 가장 큰 검증인이 제안자로 선택된다.

---

6 *결정론적 알고리즘(deterministic algorithm)은 예측한 그대로 동작하는 알고리즘이다. 어떤 특정한 입력이 들어오면 언제나 똑같은 과정을 거쳐서 언제나 똑같은 결과를 내놓는다.*

검증인 제안자 선택 방식은 다음 검증인 A,B,C,D 예시를 통해 좀 더 자세히 확인해보자. 먼저, 〈표 5.2〉처럼, 검증인 A,B,C,D 의 투표력이 각각 10,20,30,40 으로 총 투표력이 100 이라고 가정해보자.

텐더민트 제안자 선택 절차 – 1

CosmosTalk.io

검증인	Voting Power
검증인 A	10
검증인 B	20
검증인 C	30
검증인 D	40

〈표 5.2. Tendermint Proposer Selection Procedure〉

첫 라운드가 시작되고 각 검증인의 Accum 은 자신의 투표력에 비례하여 〈표 5.3〉과 같이 상승하였다.

텐더민트 제안자 선택 절차 – 2

CosmosTalk.io

1st round	Accum
검증인 A	10
검증인 B	20
검증인 C	30
검증인 D	**40**

〈표 5.3. Tendermint Proposer Selection Procedure〉

D 의 Accum 이 가장 높기 때문에 첫 라운드는 블록 제안자는 D 가 된다. 첫 라운드 제안자가 된 D 의 Accum 은 총 투표력(100)만큼 떨어져서 〈표 5.4〉과 같은 수치가 된다.

텐더민트 제안자 선택 절차 – 3

CosmosTalk.io

1st round	Accum
검증인 A	10
검증인 B	20
검증인 C	30
검증인 D	**-60 (40 – 100 = 60)**

〈표 5.4. Tendermint Proposer Selection Procedure〉

두 번째 라운드가 시작되고 각 검증인의 Accum 은 자신의 투표력만큼 상승한다.

텐더민트 제안자 선택 절차 – 4

CosmosTalk.io

2nd round	Accum
검증인 A	20 (10+10)
검증인 B	40 (20+20)
검증인 C	**60 (30+30)**
검증인 D	-20 (-60+40)

〈표 5.5. Tendermint Proposer Selection Procedure〉

두 번째 라운드 제안자는 Accum 이 가장 높은 C(60)가 된다. 제안자가 된 C 의 Accum 은 총 투표력(100)만큼 떨어져서 〈표 5.6〉과 같은 수치가 된다.

텐더민트 제안자 선택 절차 – 5

CosmosTalk.io

2nd round	Accum
검증인 A	20
검증인 B	40
검증인 C	**-40 (60 – 100 = -40)**
검증인 D	-20

〈표 5.6. Tendermint Proposer Selection Procedure〉

세 번째 라운드가 시작되고 각 검증인의 Accum 은 자신의 투표력만큼 다시 상승한다.

텐더민트 제안자 선택 절차 – 6

CosmosTalk.io

3rd round	Accum
검증인 A	30 (20+10)
검증인 B	**60 (40+20)**
검증인 C	-10 (-40+30)
검증인 D	20 (-20+40)

〈표 5.7. Tendermint Proposer Selection Procedure〉

세 번째 라운드의 B 가 가장 큰 60 Accum 을 가지게 되어 블록 제안자가 된다. 제안자가 된 B 의 Accum 은 총 투표력(100)만큼 떨어져서 〈표 5.8〉과 같은 수치가 된다.

텐더민트 제안자 선택 절차 – 7

CosmosTalk.io

3rd round	Accum
검증인 A	30
검증인 B	**-40 (60 - 100 = -40)**
검증인 C	-10
검증인 D	20

〈표 5.8. Tendermint Proposer Selection Procedure〉

네 번째 라운드가 시작되고 각 검증인의 Accum 은 자신의 투표력만큼 상승한다.

텐더민트 제안자 선택 절차 – 8

CosmosTalk.io

4th round	Accum
검증인 A	40
검증인 B	-20
검증인 C	20
검증인 D	**60**

〈표 5.9. Tendermint Proposer Selection Procedure〉

네 번째 라운드의 D 가 가장 큰 Accum 을 가지게 되어 블록 제안자가 된다. 제안자가 된 D 의 Accum 은 총 투표력(100)만큼 떨어져서 〈표 5.10.〉과 같은 수치가 된다.

텐더민트 제안자 선택 절차 – 9

CosmosTalk.io

4th round	Accum
검증인 A	40
검증인 B	-20
검증인 C	20
검증인 D	**-40 (60-100)**

〈표 5.10. Tendermint Proposer Selection Procedure〉

지금까지 텐더민트의 BFT+BPOS 컨센서스 알고리즘을 알아보았다. “Internet of Blockchains” 으로 디자인 된 퍼블릭 블록체인 코스모스 네트워크는 이 개선된 합의 알고리즘으로 점점 더 구체화되고 있으며, 다년간의 개발의 결실이 얼마 남지 않은 코스모스 메인넷으로 꽃피우게 될 것이다.

5.2. Use Case: Tendermint test with 4 nodes [22]

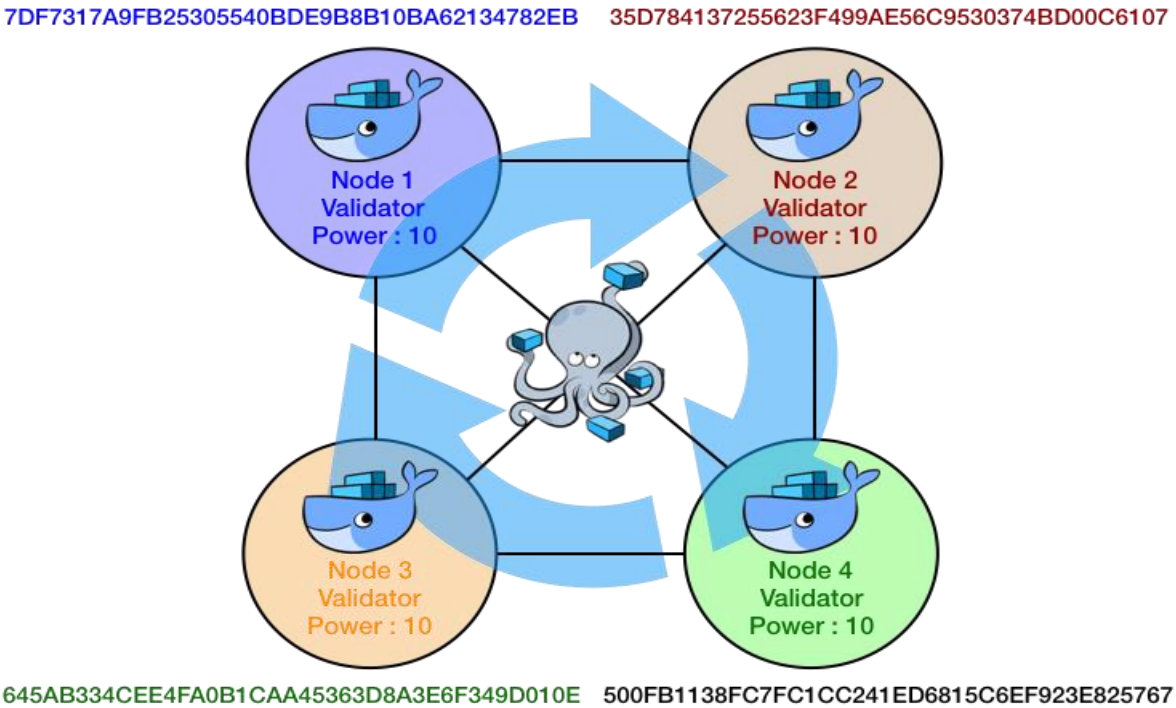
텐더민트의 합의 프로세스에 이어 이번에는 실제로 텐더민트가 어떻게 작동하는지 도커(Docker)를 통해 멀티노드를 구성하여 PBFT 합의의 작동 테스트를 진행해보았다.

테스트 순서

- docker-compose 를 이용하여 4 개 node 의 tendermint core 를 구동
- 각 tendermint core 에 구동될 ABCI 서비스 구동 (key & value 저장)
- 테스트 transaction 을 발생 및 Consensus (BPOS+PBFT)의 +2/3 합의 과정을 확인

docker-compose 는 개별 docker container 들을 하나의 네트워크로 묶어주고 구동, 정지 및 상태 모니터링등을 관리해주는 도구이다.

테스트 노드 구성도



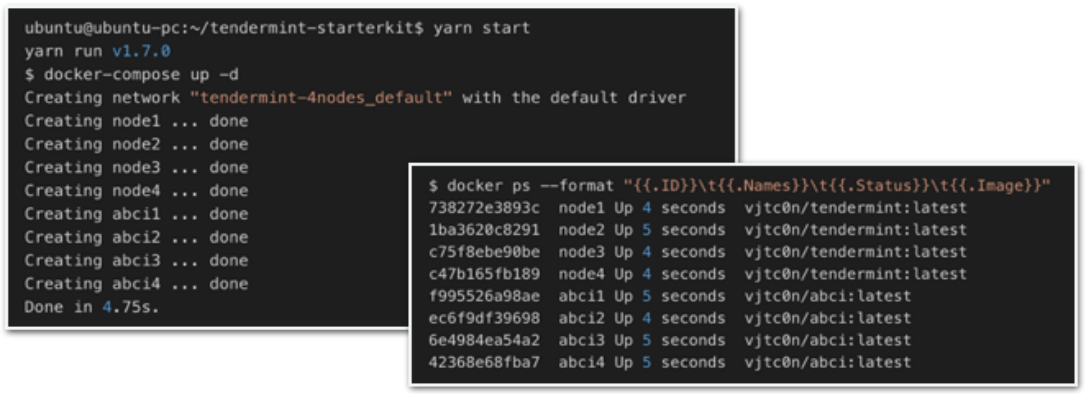
〈 그림 5.2. docker-compose 4-nodes 구성 〉

텐더민트 검증자 (Validator)는 라운드로빈 (Round Robin) 방식으로 블록 제안자 (Proposer)를 선택하며 이때, Voting Power 를 기준으로 순서가 정해진다는 사실을 앞서서 확인하였다.

(여기에서는 각 노드의 voting power 를 동일하게 10 으로 지정하였다)

구동 및 확인

1. docker-compose 로 노드 구동하기



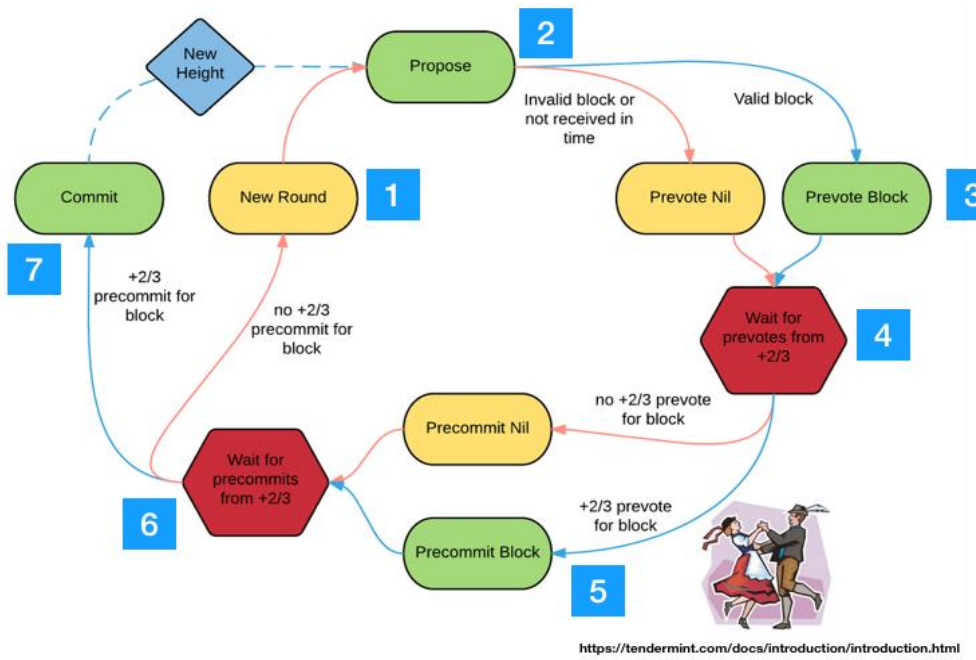
〈그림 5.3. Docker-Compose 로 노드 구성〉

2. docker-compose node 인스턴스 확인

```
$ docker-compose ps
Name              Command               State                    Ports
-------------------------------------------------------------------------------------------------
abci1   sh -c                           Up
        abci-cli kvstore  ...
abci2   sh -c                           Up
        abci-cli kvstore  ...
abci3   sh -c                           Up
        abci-cli kvstore  ...
abci4   sh -c                           Up
        abci-cli kvstore  ...
node1   sh -c tendermint node --ho ...   Up      0.0.0.0:30000->26660/tcp, 0.0.0.0:16657->46657/tcp
node2   sh -c tendermint node --ho ...   Up      0.0.0.0:26657->46657/tcp
node3   sh -c tendermint node --ho ...   Up      0.0.0.0:36657->46657/tcp
node4   sh -c tendermint node --ho ...   Up      0.0.0.0:46657->46657/tcp
```

〈그림 5.4. Docker-compose 로 노드 인스턴스 확인〉

텐더민트의 Voting Process 확인



〈 그림 5.5.텐더민트의 Voting Process 〉

트랜잭션 발생 및 블록생성 확인 순서

1. 트랜잭션 발생
2. New Round 의 Proposer 선택
3. Proposer 의 Validation 과정
4. 생성된 블록 확인

1. 트랜잭션 발생

key:value 값을 블록에 저장하는 kvstore abci 서비스로 트랜잭션을 발생시킨다.

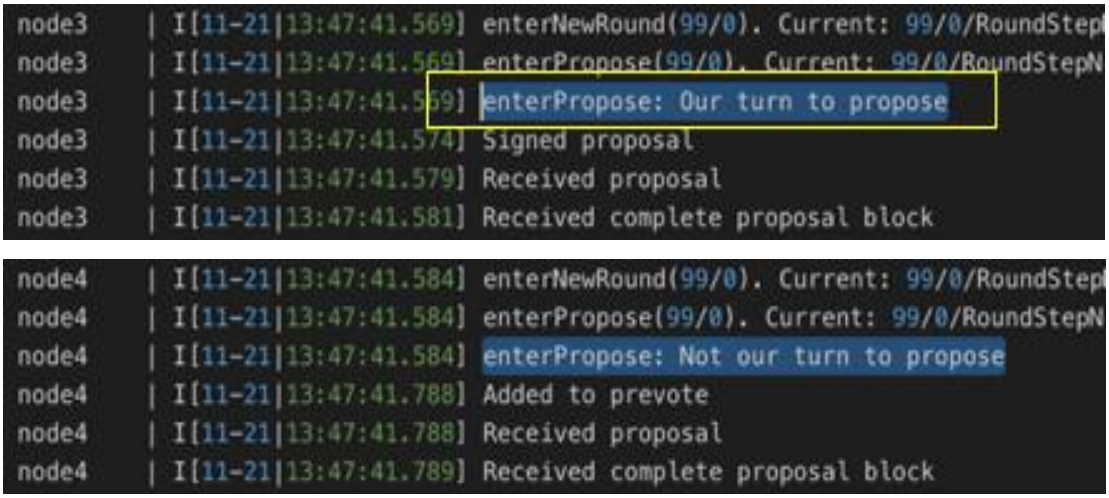
```
$ curl -s 'localhost:46657/broadcast_tx_commit?tx="abcd"'
```

2.New Round 의 Proposer 선택

텐더민트 노드에서 validation 을 위한 Proposer 선택은 Validation Power 를 기반으로하는 Round-Robin 방식으로 선택된다.

```
node1    | I[11-21|13:47:41.473] enterPropose(99/0). Current: 99/0/RoundStepN
node1    | I[11-21|13:47:41.473] enterPropose: Not our turn to propose
node1    | I[11-21|13:47:41.574] Ignoring inbound connection: error while add
node1    | I[11-21|13:47:41.678] Received proposal
node1    | I[11-21|13:47:41.678] Received complete proposal block
node1    | I[11-21|13:47:41.678] enterPrevote(99/0). Current: 99/0/RoundStepP
```

```
node2    | I[11-21|13:47:41.576] enterNewRound(99/0). Current: 99/0/RoundStep
node2    | I[11-21|13:47:41.576] enterPropose(99/0). Current: 99/0/RoundStepN
node2    | I[11-21|13:47:41.577] enterPropose: Not our turn to propose
node2    | I[11-21|13:47:41.793] Received proposal
node2    | I[11-21|13:47:41.794] Received complete proposal block
node2    | I[11-21|13:47:41.794] enterPrevote(99/0). Current: 99/0/RoundStepP
```



〈그림 5.6. New Round 에서 Proposer 선택〉

각 node 의 로그를 확인해보면 node3 이 현재 발생한 트랜잭션을 처리하기위한 블록 Proposer 로 선택되었다는 사실을 〈그림 5.6〉을 통해 확인 할 수 있다.

"enterPropose: Our turn to propose" 메시지 확인

3.Proposer 의 Validation 과정

다음으로, node3 의 Voting 과정을 로그를 통해서 확인해보자. 다음은 99 Height 의 새로운 블록이 생성되고 Voting validation 이 완료되어 새로운 블록이 생성되는 과정의 로그이다.

```
node3    | I[11-21|13:47:41.569] enterNewRound(99/0). Current: 99/0/RoundStepNewHeight module=consensus height=99 round=0
node3    | I[11-21|13:47:41.569] enterPropose(99/0). Current: 99/0/RoundStepNewRound module=consensus height=99 round=0
node3    | I[11-21|13:47:41.569] enterPropose: Our turn to propose           module=consensus height=99 round=0 proposer=
node3    | I[11-21|13:47:41.574] Signed proposal                             module=consensus height=99 round=0 proposal=
node3    | I[11-21|13:47:41.579] Received proposal                           module=consensus proposal="Proposal{99/0 1:1
node3    | I[11-21|13:47:41.581] Received complete proposal block            module=consensus height=99 hash=6C36D1CB9F64
node3    | I[11-21|13:47:41.581] enterPrevote(99/0). Current: 99/0/RoundStepPropose module=consensus
node3    | I[11-21|13:47:41.582] enterPrevote: ProposalBlock is valid        module=consensus height=99 round=0
node3    | I[11-21|13:47:41.589] Signed and pushed vote                      module=consensus height=99 round=0 vote="Vot
node3    | I[11-21|13:47:41.598] Added to prevote                            module=consensus vote="Vote{2:645AB334CEE4 9
node3    | I[11-21|13:47:41.787] Added to prevote                            module=consensus vote="Vote{3:7DF7317A9FB2 9
node3    | I[11-21|13:47:41.926] Ignoring inbound connection: error while adding peer module=p2p address=172.18.0.6:47794
node3    | I[11-21|13:47:42.014] Added to prevote                            module=consensus vote="Vote{0:35D784137255 9
node3    | I[11-21|13:47:42.014] enterPrecommit(99/0). Current: 99/0/RoundStepPrevote module=consensus height=99 round=0
node3    | I[11-21|13:47:42.014] enterPrecommit: +2/3 prevoted proposal block. Locking module=consensus height=99 round=0
node3    | I[11-21|13:47:42.025] Signed and pushed vote                      module=consensus height=99 round=0 vote="Vot
node3    | I[11-21|13:47:42.026] Added to prevote                            module=consensus vote="Vote{1:500FB1138FC7 9
node3    | I[11-21|13:47:42.026] Added to precommit                          module=consensus vote="Vote{3:7DF7317A9FB2 9
node3    | I[11-21|13:47:42.029] Added to precommit                          module=consensus vote="Vote{2:645AB334CEE4 9
node3    | I[11-21|13:47:42.129] Added to precommit                          module=consensus vote="Vote{0:35D784137255 9
node3    | I[11-21|13:47:42.129] enterCommit(99/0). Current: 99/0/RoundStepPrecommit module=consensus height=99 commitRoun
node3    | I[11-21|13:47:42.129] Commit is for locked block. Set ProposalBlock=LockedBlock module=consensus height=99 comm
node3    | I[11-21|13:47:42.131] Finalizing commit of block with 1 txs       module=consensus height=99 hash=6C36D1CB9F64
```

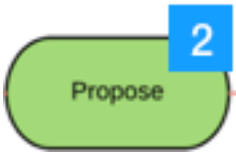
〈그림 5.7. Proposer 의 Validation 과정〉

〈그림 5.7〉의 로그를 Voting 순서에 맞도록 나누어서 〈 텐더민트 Voting Process 〉 7 가지의 과정으로 확인해보자.

```
node3    | I[11-21|13:47:41.569] enterNewRound(99/0). Current: 99/0/RoundStepNewHeight module=consensus height=99 round=0
node3    | I[11-21|13:47:41.569] enterPropose(99/0). Current: 99/0/RoundStepNewRound module=consensus height=99 round=0
node3    | I[11-21|13:47:41.569] enterPropose: Our turn to propose     module=consensus height=99 round=0 proposer=645AB334CEE4FA0B1CAA
node3    | I[11-21|13:47:41.574] Signed proposal                       module=consensus height=99 round=0 proposal="Proposal{99/0 1:125
node3    | I[11-21|13:47:41.579] Received proposal                     module=consensus proposal="Proposal{99/0 1:125BAC30DCFF (-1,:0:0
node3    | I[11-21|13:47:41.581] Received complete proposal block      module=consensus height=99 hash=6C36D1CB9F64E891D2B58EE423239BCF
```

〈그림 5.8. Consensus Process〉

99 번으로 NewRound 가 시작된다.



node3 이 Proposer 로 선택되었다. 이때, Proposer node 의 ID 를 확인해보면 다음과 같이 확인이 된다.

NODE1 : 7DF7317A9FB25305540BDE9B8B10BA62134782EB

NODE2 : 35D784137255623F499AE56C9530374BD00C6107

NODE3 : 645AB334CEE4FA0B1CAA45363D8A3E6F349D010E 〈-현재 블록의 Validator〉

NODE4 : 500FB1138FC7FC1CC241ED6815C6EF923E825767

〈그림 5.8. 현재 블록의 Validator〉

```
node3    | I[11-21|13:47:41.581] enterPrevote(99/0). Current: 99/0/RoundStepPropose module=consensus
node3    | I[11-21|13:47:41.582] enterPrevote: ProposalBlock is valid  module=consensus height=99 round=0
node3    | I[11-21|13:47:41.589] Signed and pushed vote                module=consensus height=99 round=0 vote="Vote{2:645AB334CEE4 99/
node3    | I[11-21|13:47:41.598] Added to prevote                      module=consensus vote="Vote{2:645AB334CEE4 99/00/1(Prevote) 6C36
node3    | I[11-21|13:47:41.787] Added to prevote                      module=consensus vote="Vote{3:7DF7317A9FB2 99/00/1(Prevote) 6C36
node3    | I[11-21|13:47:41.926] Ignoring inbound connection: error while adding peer module=p2p address=172.18.0.6:47794 err=Error{EOF
node3    | I[11-21|13:47:42.014] Added to prevote                      module=consensus vote="Vote{0:35D784137255 99/00/1(Prevote) 6C36
node3    | I[11-21|13:47:42.014] enterPrecommit(99/0). Current: 99/0/RoundStepPrevote module=consensus height=99 round=0
node3    | I[11-21|13:47:42.014] enterPrecommit: +2/3 prevoted proposal block. Locking module=consensus height=99 round=0 hash=6C36D1CB
node3    | I[11-21|13:47:42.025] Signed and pushed vote                module=consensus height=99 round=0 vote="Vote{2:645AB334CEE4 99/
node3    | I[11-21|13:47:42.026] Added to prevote                      module=consensus vote="Vote{1:500FB1138FC7 99/00/1(Prevote) 6C36
```

〈그림 5.9. Added to Prevote〉

“Added to prevote”로 전체 4 개 Validator 중 순차적으로 node3, node1, node2 가 검증자(Validator)로서 “Prevote”를 했음을 알 수 있다.

전체 노드의 +2/3 이 prevote 되길 기다리다가 , +2/3 를 충족하는 순간 prevoted proposal 이 block 이 locking 된다.

locking 이후에 들어온 prevote 

도 들어오지만 이미 +2/3 은 충족 된다.

```
node3    | I[11-21|13:47:42.026] Added to precommit                  module=consensus vote="Vote{3:7DF7317A9FB2 99/00/2(Precommit) 6C
node3    | I[11-21|13:47:42.029] Added to precommit                  module=consensus vote="Vote{2:645AB334CEE4 99/00/2(Precommit) 6C
node3    | I[11-21|13:47:42.129] Added to precommit                  module=consensus vote="Vote{0:35D784137255 99/00/2(Precommit) 6C
node3    | I[11-21|13:47:42.129] enterCommit(99/0). Current: 99/0/RoundStepPrecommit module=consensus height=99 commitRound=0
```

〈그림 5.10. Locking 이후 prevote 충족〉

다음 단계로 “Added to precommit” 메시지가 

순차적으로 들어와서 

+2/3 precommit 의 조건을 충족하게 된다.

```
node3    | I[11-21|13:47:42.129] Commit is for locked block. Set ProposalBlock=LockedBlock module=consensus height=99 commitRound=0 bloc
node3    | I[11-21|13:47:42.131] Finalizing commit of block with 1 txs module=consensus height=99 hash=6C36D1CB9F64E891D2B58EE423239BCF
```

〈그림 5.11. precommit 충족〉

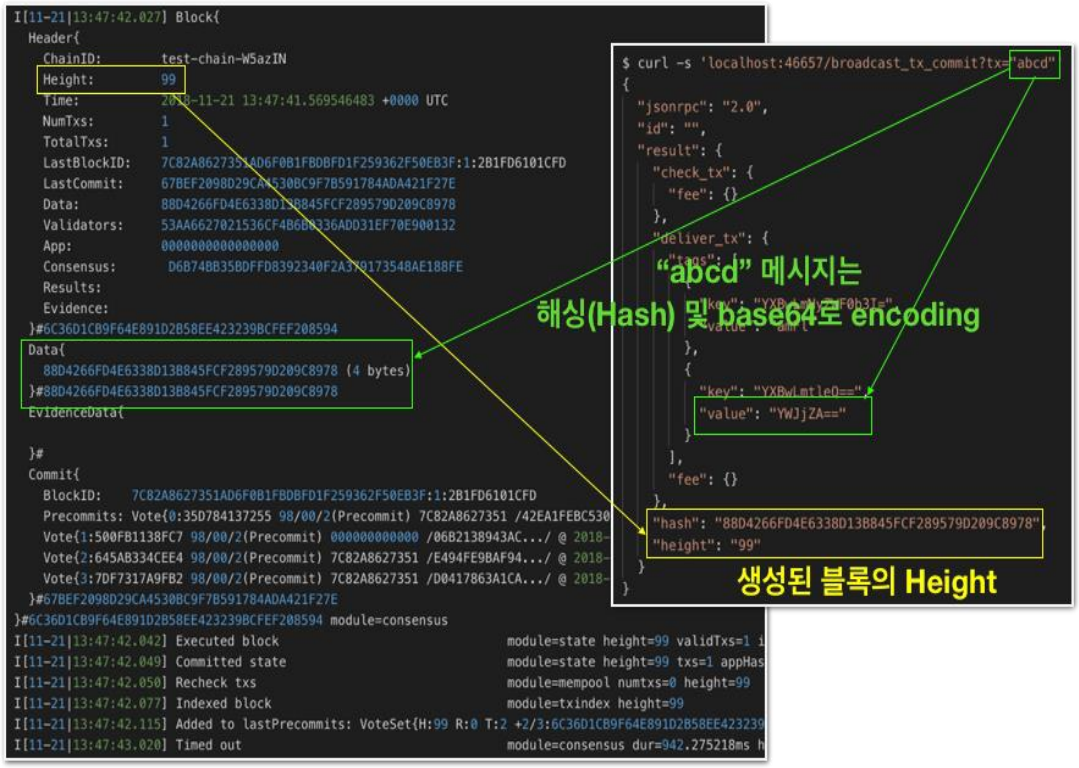
+2/3 precommit 의 조건이 완료되면, 최종으로 해당 블록은 commit 된다.

〈그림 5.12.〉의 로그를 통해 최종 생성된 블록 정보를 확인할 수 있다.

```
node3   | I[11-21|13:47:42.131] Block{
node3   |   Header{
node3   |     ChainID:        test-chain-W5azIN
node3   |     Height:         99
node3   |     Time:           2018-11-21 13:47:41.569546483 +0000 UTC
node3   |     NumTxs:         1
node3   |     TotalTxs:       1
node3   |     LastBlockID:    7C82A8627351AD6F0B1FBDBFD1F259362F50EB3F:1:2B1FD6101CFD
node3   |     LastCommit:     67BEF2098D29CA4530BC9F7B591784ADA421F27E
node3   |     Data:           88D4266FD4E6338D13B845FCF289579D209C8978
node3   |     Validators:     53AA6627021536CF4B6B0336ADD31EF70E900132
node3   |     App:            0000000000000000
node3   |     Consensus:       D6B74BB35BDFFD8392340F2A379173548AE188FE
node3   |     Results:
node3   |     Evidence:
node3   |   }#6C36D1CB9F64E891D2B58EE423239BCFEF208594
node3   |   Data{
node3   |     88D4266FD4E6338D13B845FCF289579D209C8978 (4 bytes)
node3   |   }#88D4266FD4E6338D13B845FCF289579D209C8978
node3   |   EvidenceData{
node3   |
node3   |   }#
node3   |   Commit{
node3   |     BlockID:    7C82A8627351AD6F0B1FBDBFD1F259362F50EB3F:1:2B1FD6101CFD
node3   |     Precommits: Vote{0:35D784137255 98/00/2(Precommit) 7C82A8627351 /42EA1FEBC530.../ @ 2018-11-21T13:47:40.38
node3   |     Vote{1:500FB1138FC7 98/00/2(Precommit) 000000000000 /06B2138943AC.../ @ 2018-11-21T13:47:40.572Z}
node3   |     Vote{2:645AB334CEE4 98/00/2(Precommit) 7C82A8627351 /E494FE9BAF94.../ @ 2018-11-21T13:47:40.383Z}
node3   |     Vote{3:7DF7317A9FB2 98/00/2(Precommit) 7C82A8627351 /D0417863A1CA.../ @ 2018-11-21T13:47:40.462Z}
node3   |   }#67BEF2098D29CA4530BC9F7B591784ADA421F27E
node3   | }#6C36D1CB9F64E891D2B58EE423239BCFEF208594 module=consensus
node3   | I[11-21|13:47:42.169] Executed block                          module=state height=99 validTxs=1 invalidTxs
node3   | I[11-21|13:47:42.172] Committed state                         module=state height=99 txs=1 appHash=0200000
node3   | I[11-21|13:47:42.172] Recheck txs                             module=mempool numtxs=0 height=99
node3   | I[11-21|13:47:42.217] Added to lastPrecommits: VoteSet{H:99 R:0 T:2 +2/3:6C36D1CB9F64E891D2B58EE423239BCFEF2085
node3   | I[11-21|13:47:42.218] Indexed block                           module=txindex height=99
node3   | I[11-21|13:47:43.131] Timed out                               module=consensus dur=912.168462ms height=100
```

〈그림 5.12. 로그 최종 생성된 블록 정보〉

생성된 블록과 트랜잭션 정보를 확인해보면 〈그림 5.13.〉과 같이 체크 해볼 수 있다.



〈그림 5.13. 생성된 블록과 트랜잭션 정보 확인〉

텐더민트를 실제로 도커를 이용해 4 개의 노드로 구성하여 테스트를 진행하였다. 이를 통해 실제로 텐더민트가 어떻게 합의과정을 이루는지 알 수 있었다. 텐더민트의 경우 2/3+1 개의 합의가 각 스텝에서 모여야하며 어떤 경우에도 서로 다른 두 블록이 동시에 생성되지 않는다. 그러나 이러한 전송 메시지는 시간 안에 도달하는 것을 보장 못하는 비동기 네트워크에서는 사용되기 어려울 수 있다.

앞서서 언급하였듯이, 비동기 시스템 모델의 경우 메시지가 전송되는 것을 보장하는 상한 시간이 없기 때문에 메시지가 전달 실패 또는 거의 무한히 지연될 수 있다. 반대로 시간 안에 메시지를 보장하는 동기성 시스템 모델을 분산 시스템에서 사용하는 것도 무리가 있다. 따라서 텐더민트에서는 동기성 시스템과 비동기 시스템의 특성을 번갈아가면서

사용하는데, 이를 부분 동기성(Partial Synchrony)이라 한다. 부분 동기성 시스템에서는 정해진 시간(fixed time)안에 메시지가 도착하는 것은 보장하지만 얼마나 걸리는지에 대한 시간(unknown amount of time)은 알 수 없다. 다시 말해, 네트워크 안에서 정해놓은 시간 안에는 도착하지만 그 정해진 시간을 노드들은 파악하지 못함을 이야기한다. 이처럼 분산 시스템 혹은 블록체인과 같은 P2P 컴퓨팅 방식을 합의하는 과정에서 어려움을 많이 겪는다.

6. Conclusion

본 리서치를 통해 분산 시스템에서의 합의 과정은 매우 제한된 조건에서 만족 함을 알 수 있었다. 이는 블록체인에서도 완벽하게 해결을 하지 못한다. 그러나 최초에 나온 비트코인의 작업증명방식은 FLP Impossibility 를 우회적으로 해결하기 위해 Liveness over Safety 를 선택하며, BFT 기반의 텐더민트는 Safety over Liveness 를 선택한다.

이더리움(Ethereum)의 경우 현재 작업증명 방식(Proof-of Work)을 통해 Liveness 를 지분증명 방식(Proof of Stake)의 형태인 Beacon chain 을 통해 Safety 까지 보장하려는 'Casper the Friendly Finality Gadget (Casper FFG)'을 목표로 개발을 하고 있다. **(Casper FFG 는 토큰이코노미 분과 자료를 통해 경제학적인 모델 설계까지 고려하고 있음을 확인할 수 있다.)**

현재 블록체인에서는 수많은 합의 알고리즘 방식이 등장하고 있다. 그러나 흔히들 혼동되는 것은 합의를 하는 과정과 합의를 가지는 주체의 차이이다. 가령 BFT 계열의 합의 알고리즘은 이더리움, EOS, 텐더민트 등이 있다. 이들이 합의하는 과정을 BFT 계열로 한다면 이를 실제로 처리하는 이들은 우리가 알고 있는 PoS, DPoS, BPoS 등이다. 합의 과정과 주체를 혼동하지 않도록 주의가 필요하다.

앞선 몇 가지의 논문을 가지고 분석을 진행하였지만, 본 리서치만으로 합의하는 과정을 모두 정리했다고 볼 수 없다. 추후 리서치를 통해 Paxos 혹은 하이퍼랫져(Hyperledger) 등에서 사용하는 Raft 등에 대한 리서치를 통해 분산 시스템에서의 합의 과정을 더욱 심도 있게 연구할 계획이다.

APPENDIX

CAP and PACELC: The Classifying Criterion for Distributed Computer System

1. Introduction of CAP theorem

기술리서치 분과는 분산 시스템 내에서 합의 알고리즘에 대한 리서치를 수행하였다. 이 과정에서 CAP Theorem 을 이용한 Private Blockchain 의 합의 알고리즘을 분류하려는 시도가 있음을 발견하였다. 해당 문서는 Proof-of-Authority(PoA)와 PBFT 를 CAP Theorem 을 기준으로 유형화하는 논문(PBFT vs proof-of-authority: applying the CAP theorem to permissioned blockchain: University of Southampton) [24]이다.

PBFT 리서치의 일환으로 해당 논문에 대한 리서치를 진행하던 중, 많은 이들이 CAP 과 FLP 의 개념에 대한 혼동이 있음을 발견할 수 있었다. 해당 이론은 블록체인의 근간이 되는 분산 컴퓨팅의 기초에 직결되는 아이디어이므로 본 분과에서는 깊은 관심을 가지고 리서치를 진행하였다. 그러나, 이러한 논문은 거시적 합의 알고리즘 자체에 집중하고 있는 본 연구 보고서에 적합하지 않다고 보았다. 미시적 통신 이론인 CAP 과 PACELC 을 자세히 설명하는 것은 본 연구 보고서의 전반적인 흐름과 다르다고 판단하였다.

그러나 CAP 과 PACELC [28] 이론은 블록체인의 근간이 되는 분산 컴퓨팅 분야에서 비중 있게 다루어진다는 점, 비중 있게 다루어지는 이론임에도 불구하고 학계에서 CAP 이론에 대한 해석의 여지가 모두 다르다는 점, 국내에 PACELC 이론이 자세히 소개된 적이 없다는 점을 고려하여 Appendix(부록)에 별도로 서술하기로 결정하였다.

Appendix 에서는 미시적 통신의 가부 여부를 판단하는 CAP 이론에 대해 알아보고자 한다. 이후, CAP 의 한계점을 지적한 뒤 이를 보완하는 새로운 개념인 PACELC 를 설명하고자 한다. PACELC 에 기초해 현존하는 합의 알고리즘을 분류할 것이다. 이를 통하여 다양한 합의 알고리즘이 어떠한 부분을 강조하고 포기하는지에 대하여 알아보고 FLP 와 분명히 다른 것임을 알아본다..

2. CAP 이론 소개 [26,27,28, 33, 35]

분산 컴퓨팅 시스템에서 네트워크의 원활한 운영을 촉진하기란 어려운 법이다. 컴퓨터 과학자인 에릭 브루어 (Eric Brewer)는 [25,26] 2000 년 7 월 19 일 PODC (Symposium on Principles of Distributed Computing)에서 CAP 이론을 발표하였다. CAP 이론은 분산 데이터베이스 시스템을 바탕으로, 네트워크 분할 허용(Network Partition)이 이루어지는 경우에 일관성(Consistency), 가용성(Availability)을 동시에 만족하는 것은 불가능하며 이들 사이에 양자택일해야 한다는 이론이다.

CAP 이론에 따르면 모든 분산 네트워크는 일관성을 지키는 상황 (Consistency-Partition Tolerance: CP)과 가용성을 지키는 상황(Availability-Partition Tolerance: AP)으로 나눌 수 있다고 보았다. 모든 노드는 데이터를 업데이트하는 과정에서 다른 작업이 불가능하다. 따라서 업데이트가 진행되는 시간에는 일시적으로 데이터베이스에 접속할 수 없게 되어 서비스가 중지된다.

모든 노드에 정보를 업데이트하여 일관성(Consistency)을 지키되 가용성(Availability)을 포기할 것인가? 혹은 일부 노드에만 정보를 업데이트하여 가용성(Availability)을 지키되 일관성 (Consistency)를 포기할 것인가? 에 대한 문제이다.

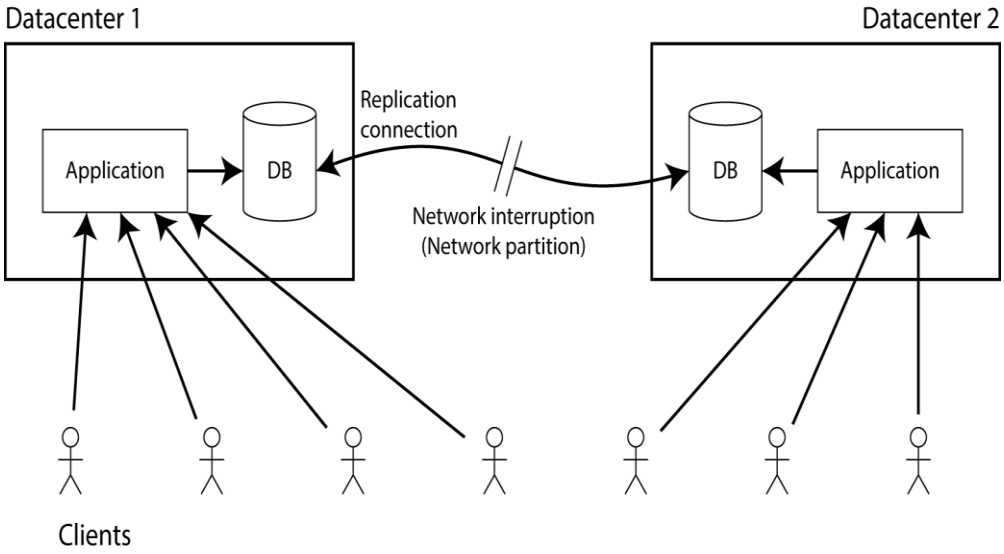
각 용어에 대한 의미에 대해 자세하게 파악해보자.

2.1 네트워크 분할 허용(Network Partition)

- *노드 간의 통신이 실질적으로 어려운 상태에 도달하여 네트워크가 사실상 분할(Partition)되었을 때도 네트워크가 정상적으로 작동할 수 있는 경우를 의미한다.*

네트워크 분할 허용을 전제하는 네트워크는 노드 간의 통신이 완전히 불가능하거나 비동기성 네트워크 (asynchronous network)에서 메시지가 유실될 수 있고, 현실성이 떨어지는 시간에 메시지가 전달될 수 있다. 이러한 상황에서도 최종 사용자(End-User)입장에서는 문제 없이 네트워크를 이용할 수 있어야 한다. 다시 말해, 최종 사용자가

하나의 분산 네트워크가 여러 가지의 노드로 나뉘었다는 사실을 인지하지 못하도록 하여야 한다. 분산 네트워크가 단일 네트워크처럼 보여야 유의미하다고 말할 수 있다.



모든 분산 네트워크는 네트워크 분할 허용성(Network Partition, P)을 보유할 수밖에 없다. 현실적으로 노드 간의 네트워크 통신은 다양한 환경적 조건에 따라서 불안정해질 수 있기 때문이다. 또한, 모든 분산 네트워크는 비동기성 네트워크 (asynchronous network) 상황에 처해질 가능성이 존재한다. 분산 네트워크는 노드 간의 통신이 불안정하더라도 네트워크가 정상적으로 동작할 수 있도록 한다는 것에 의의가 있다. 따라서 분산형 네트워크에서 분할 허용성을 가정하지 않는다면 사실상 의미가 없어지게 된다. 즉, CAP 이론에서는 네트워크 분할 허용(P)을 기본으로 전제한다.

2.2 일관성(Consistency)

- *모든 요청은 최신 데이터 또는 에러를 응답 받는다.*

일관성(Consistency, C)은 최신 데이터의 전파 속도 및 업데이트 노드 양에 따라서 그 강도가 다르게 분류된다. 그러나 CAP 이론이 초창기에 소개될 때 어느 강도의 일관성을

의미하는 지에 대해서 실질적으로 명시된 바가 없었다. 애시당초 CAP 이론은 Brewer 가 제시한 가설에 불과하였으므로 구체적인 근거를 제시하지 않았다.

CAP 이론을 추후에 Lynch 가 증명할 때 Linearizability(Atomic/Strong Consistency)를 일관성의 기준으로 설정하였다. 이 외에도 증명을 위하여 읽기와 쓰기 권한만을 싱글 스레드로 순차적으로 처리하는 Simple Service 를 가정하였다.

Linearizability 는 Operation A 가 성공적으로 종료된 이후에 Operation B 가 시작될 때, Operation B 는 Operation A 의 최종 결과치를 모든 시스템에서 하나의 동일한 상태(the same state)로 같은 논리적 시간(logical time)에 업데이트되어 관찰할 수 있어야 한다는 기준이다. 즉, 한 노드에서 특정 데이터의 처리가 완료되었을 때 다른 노드가 동시 다발적으로 업데이트된 정보에 접근할 수 있어야 한다는 것이다. 따라서 Lineraizability 는 매우 강력한 형태의 일관성을 요구한다.

쉬운 이해를 담보하기 위하여 Not Linearizable 한 시스템을 시나리오로 이야기해보자. 철수는 근처 펍에 가서 2018 월드컵 조별리그, 한국 대 독일의 경기를 보고 있다. 영희는 한국이 이기지 못할 것으로 생각하고, 학교에서 시험 공부를 하고 있던 찰나여서 경기를 보지 못하고 있다. 경기가 끝나고 한국이 2:0 으로 이겼다. 철수는 감격한 나머지 포털 사이트에 접속해서 경기 결과를 재확인했다. 철수는 영희에게 카카오톡으로 메시지를 보내 경기 결과를 알려주었다.

한국이 무려 2:0 으로 승리했다는 사실을 듣고 영희는 포털 사이트에 접속했다. 그러나 영희가 접속한 포털 사이트에는 여전히 게임이 진행되고 있다라는 표시가 뜬다. 철수는 최신 업데이트가 된 정보를 받을 수 있었다. 하지만 영희는 받지 못했다. 철수와 연결된 서버 노드는 월드컵 경기가 끝나고 결과를 확인했다. 그러나 영희와 연결된 서버 노드는 새로운 결과를 받지 못했다. 즉, 가용성을 우선시하고 Linearizable 한 일관성을 포기한 것이다.

2.3 가용성 (Availability)

- *정상적으로 이루어진 모든 요청은 정상적으로 작동하고 있는 모든 노드로부터 응답을 받을 수 있다.*

일반적으로 우리가 높은 가용성(High Availability)라고 일컫는 경우는 다음과 같다. 우선 높은 가용성은 시스템 자체가 모든 요청에 대한 정상적인 응답을 할 수 있다는 것을 전제한다. 따라서 모든 요청은 정상적으로 작동하고 있는 노드 중 **일부**만 응답을 받을 수 있어도 충분하다. 하나의 정상 노드만 응답해도 시스템은 가용이 가능하다고(Available) 말할 수 있기 때문이다. 또한 높은 가용성을 가정하는 데이터베이스는 현실적인 응답시간(realistic response time)을 고려한다.

그러나 CAP 이론에서 의미하는 가용성은 높은 가용성보다 훨씬 높은 난이도를 요구한다. 높은 가용성을 보장하는 서버는 일반적으로 CAP 이론이 주장하는 가용성에 해당하지 않는다. CAP 이론에서 주장하는 가용성은 다음과 같다. 우선 오류가 없는 노드가 발송한 메시지는 오류가 없는 ***모든*** 노드로부터 응답을 받을 수 있어야 한다. 데이터 저장소에 대한 모든 동작이 오류가 없는 모든 노드로부터 성공적으로 리턴을 받아야 한다.

또 한 가지 주목해야 할 점은 CAP 이론의 가용성은 응답시간에 대한 제한을 걸어두지 않는다(unbounded response time)는 사실이다. 다시 말하면 응답시간(Latency)에 대한 고려가 부재하다는 것이다. 분산형 네트워크는 기본적으로 응답시간이 실사용에 있어서 중요한 고려요소(consideration)이 될 수밖에 없다.

기본적으로 분산형 네트워크는 단일 네트워크보다 속도가 느리다. 그 이유는 모든 또는 일부 데이터를 복제하여 저장하고자 하기 때문에 정보의 전파 과정에서 일정한 시간이 필수적으로 요구되기 때문이다. 하나의 데이터를 복제하여 저장해야 하는 노드의 수가 많아질수록 속도가 느려진다.

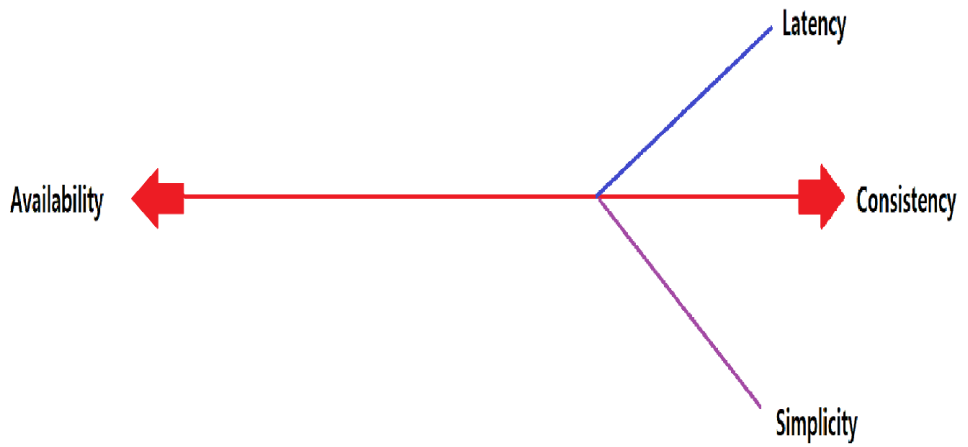
그렇다고 해서 속도가 치명적으로 느린 네트워크는 실제 사용하기에 큰 장애가 있을 것이다. 가용성은 응답 시간과 깊은 관련이 있음에도 CAP 이론에서는 이를 고려하지 않는다는 단점이 있다. CAP 이론의 가용성에 따르면 아무리 늦게 보내도 받기만 하면 충분하다. 특정 데이터를 20 시간 뒤에 받는다면 CAP 의 가용성에는 부합할 지 모르나 실제 사용시에는 부적합하다고 보아야 한다.

CAP 이론이 이야기하는 네트워크 분할 용이성 (Network Partition) 에 따르면 노드 간의 통신이 완전히 불가능한 경우도 상정한다. 이러한 경우, 완전한 CP (Consistency-

Partition Tolerance) 시스템은 존재할 수 없다. 노드 간의 통신이 끊어진 상황에서 데이터를 업데이트하는 것이 불가능하기 때문이다.

분산 네트워크의 기본적인 특징을 설명하는 ACID 이론에 따르면, 모든 네트워크는 고립성(Isolated)이 존재한다. 트랜잭션이 실행이 되는 중에 생성하는 연산의 중간 결과는 다른 트랜잭션이 접근할 수 없다는 것이다. 다시 말하면, 트랜잭션을 처리할 때 다른 트랜잭션이 연산 작업 중간에 끼어들지 못하는 것이다. 트랜잭션 실행 내역이 연속적이어야 하기 때문이다.

노드 간의 네트워크가 물리적으로는 끊어지지 않았으나 원활한 통신을 보장할 수 없는 경우에는 다음과 같은 선택지가 나타날 수 있다.



◆ **Choosing Availability over Consistency (AP 시스템)**

AP 시스템은 가용성을 우선시하므로 모든 request 에 응답하게 될 것이다. 또한 stale reads 를 허용하게 될 가능성이 높고 서로 충돌되는 데이터가 쓰여질 가능성이 높다. 완벽한 가용성을 지니는 시스템은 정상적인 모든 노드가 어떤 상황에서라도 응답할 수 있어야 한다. 하나의 노드가 네트워크 파티션으로 고립되었을 때 고립된 노드의 데이터는 일관성이 깨지므로 쓸모가 없어진다. 그러나 정상적으로 응답을 하면 완벽한 가용성을

지닌다. 이러한 사실을 모르는 사용자는 문제를 인지하지 못하고 요청을 계속해서 보낼 것이다.

◆ Choosing Consistency over Availability (CP 시스템)

CP 시스템은 완벽한 일관성을 갖는 분산 시스템으로 하나의 트랜잭션이 다른 모든 노드에 완전히 복제된 후에야 그 다음 작업으로 넘어갈 수 있다. 네트워크가 완전히 분리되어 노드 간의 통신이 불가능한 경우 가용성을 잃게 된다. 노드 간의 통신이 가능한 현실적인 경우에는 성능의 희생을 필요로 한다. 하나의 노드라도 문제가 생기면 트랜잭션은 실패한다. 노드가 늘어날 수록 지연시간은 길어질 수밖에 없다.

일부 request 는 경우에 따라 정상적인 모든 노드가 refuse 할 수 있다. 시스템 전체를 일시적으로 정지시킬 수도 있고 (single-node data store clients) 또는 아예 쓰기를 거부할 수도 있고 (Two-Phase Commit) 또는 마스터 노드만 파티션 부분(partition component)에 쓰기를 할 수도 있다. 예시로는 Membase 가 있다.

3. CAP 이론의 한계 [27,30,31, 33]

CAP 이론은 분산 네트워크 시스템에서 새로운 아이디어를 제시했다는 점에서 의미가 있으나 엄밀하지 못한 이론이라는 점에서 실제 분석에 적용하기에는 두 가지의 문제점이 존재한다.

3.1 편협한 정의로 인한 상대적인 관점의 부족

CAP 이론의 가장 큰 문제점은 일관성과 가용성이 배타적이라고 전제한다는 것이다. 일관성과 가용성은 상호 배치되는 속성인 것은 사실이다. 그러나 일관성과 가용성이 무조건 양자택일의 관계라고 주장하기에는 어폐가 있는 것이 사실이다. 완벽한 CP 시스템과 AP 시스템은 존재하지 않고 필요하지도 않다. 모든 데이터베이스는 상대적인 관점을 바탕으로 설계된다. 가용성을 중시하면서 일관성을 일부 포기하거나, 일관성을 중시하면서 가용성을 일부 포기하는 것이다. 그러나 CAP 이론은 일관성과 가용성 중 하나만을 선택하라고 강요한다.

이러한 양자택일의 속성은 CAP 이론이 증명 과정에서 채택하는 일관성과 가용성의 정의가 편협하다는 것에 비롯된다. 한 쪽이 업데이트되면 다른 쪽도 동시다발적으로 업데이트되어야 하며, 정상적인 모든 노드가 모든 요청을 100% 응답할 수 있어야 한다는 가정은 실제 세계에서 이루어질 수 없는 수준의 가정이다. 실제 설계에서 채택되는 상대적인 관점을 고려할 수 없다는 것이 CAP 이론의 가장 큰 문제점이다.

3.2 Network 에 대한 적용범위 협소

CAP 이론에 따르면 네트워크 분할 용인성은 노드 간의 통신이 완전히 불가능한 경우를 위주로 고려한다. 다시 말해서, 물리적인 이유로 인하여 노드 간의 통신이 불가능하여 상호 간의 업데이트가 전혀 일어날 수 없는 경우를 위주로 설명한다. 물론 굳이 물리적인 이유가 아니더라도 소프트웨어적인 문제로 인하여 유의미한 시간에 노드 간 데이터가 정상적으로 (실질적으로) 통신 될 수 없는 경우까지 포괄한다.

실제로 작동하는 데이터베이스에서 완전히 또는 실질적으로 통신이 불가능한 경우를 찾기는 쉽지 않다. 모든 데이터베이스는 일반적으로 노드끼리 서로 유의미한 시간에 데이터를 주고 받을 수 있는 환경에 있다. 이러한 경우에는 각 노드 간의 정보 업데이트가 원활하게 이루어질 수 있다. 이를 부분동기성 네트워크(Partial Synchronous Network)라고 지칭하겠다.

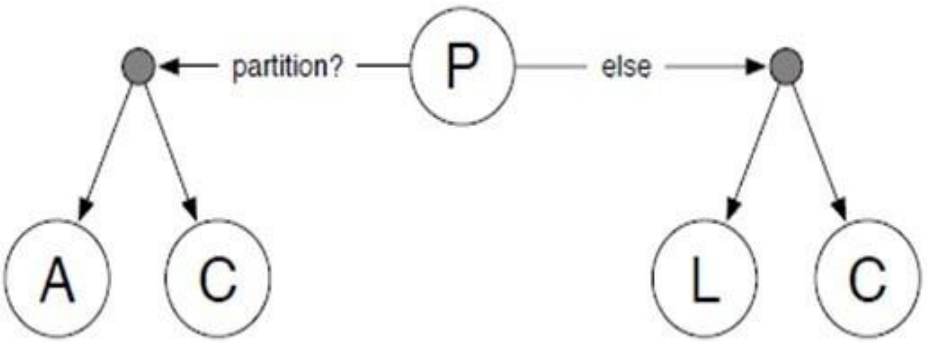
CAP 이론은 부분동기성 네트워크의 경우를 고려하지 않는다. 네트워크 분할 용이성이 포괄하는 범위가 지나치게 협소하여 노드 간의 통신이 정상적으로 이루어지는 부분동기성 네트워크의 경우를 무시하였기 때문이다. 해당 내용은 가용성 (Availability)부분에서 이미 자세하게 소개하였다.

따라서 실질적으로 CAP 이론을 적용할 수 있는 네트워크 또는 데이터베이스는 많지 않다. Single-reader replication 의 사례를 보면 다음과 같다. 이는 하나의 single leader 를 replicated database 에 쓰기 작업을 할 수 있는 일반적인 경우이다. 이러한 설정에서 client 가 leader 로부터 partition 되는 경우에 해당 데이터베이스는 전혀 사용 가치가 없게 된다. 따라서 엄밀한 의미에서 CAP-available 하지 않다는 문제점이 있다.

예를 들어서, 새로운 data 가 나왔는데 제대로 된 응답을 해 주지 못하기 때문이다. 따라서 CA 시스템이 아니다.

그렇다면 이 경우가 CP 시스템이라고 할 수 있을까? 그렇지 않다. client 가 읽기 작업을 한다는 것은 일반적으로 비동기성(asynchronous)을 전제한다. 따라서 클라이언트가 읽기 작업을 하는 순간에 쓰기 작업을 수행하는 리더보다 뒤쳐질 가능성이 있다. 그러므로 완전한 Lineraizability 를 충족하는 일관성이 있다고 보기 어렵다. 편협한 정의로 인하여 일관성과 가용성이 대치되지 않는 것이다.

4. PACELC Theorem [28, 29 ,30,34]



[사진 출처: http://eincs.com/2013/07/misleading-and-truth-of-cap-theorem/]

CAP 이론은 분산 시스템의 특징을 명료하게 드러내준다. 고민할 만한 지점을 던져준 것은 유의미한 일이다. 그러나 실제 적용하기에는 편협한 정의로 인하여 여러 이유로 무리가 있는 것이 사실이다. 일관성과 가용성은 상충 관계에 있지만 절대적인 것은 아니다라는 것이다. 다소 약한 가용성을 지니지만 상대적으로 강한 일관성을 지닐 수 있고, 다소 약한 일관성을 지니지만 상대적으로 강한 가용성을 지닐 수 있다. 가용성과 일관성은 서로 대치되지만 궁극적으로 일치되는 것을 목표로 하기 때문에 상대적이라는 것을 알 수 있다.

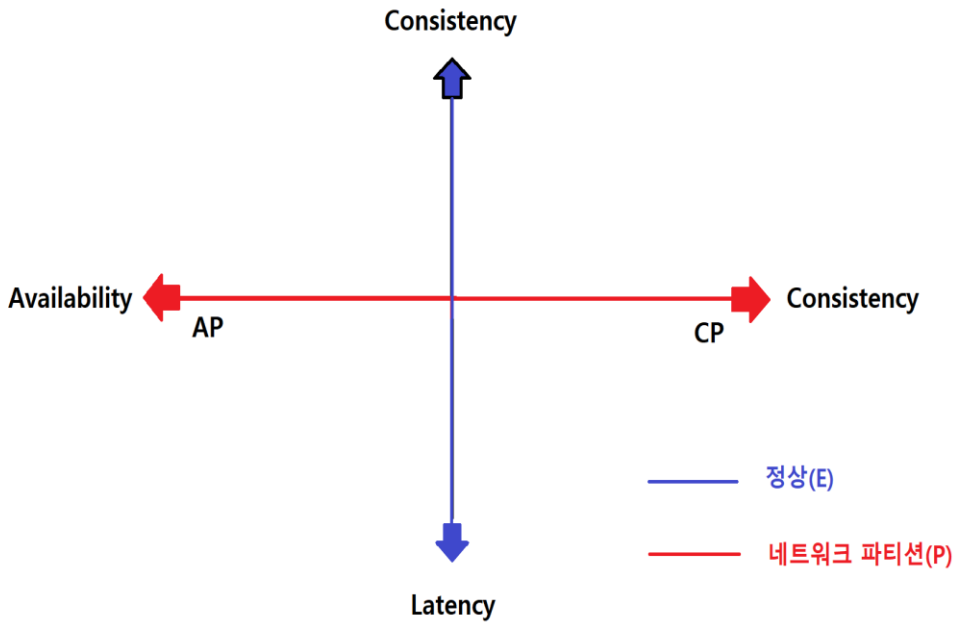
다른 한 가지는 네트워크 노드 간의 연결이 원활한 경우이다. 노드 간의 통신이 원활한 경우에는 CAP 이론이 정의하는 네트워크 분할 용이성 (Network Partition)의 정의에 해당하지 않는다.결론적으로 이러한 경우는 해석 하지 못한다. 분산 네트워크에서는 특정

노드가 추후 오류가 나서 임의의 데이터에 접근하지 못하는 경우를 배제하기 위하여 다른 노드에도 데이터를 복사하여 저장한다. 이러한 경우에도 분산 네트워크가 지니는 대칭적 특성이 드러난다. 이는 앞서 설명한 것으로, 속도(latency)와 일관성(consistency)간의 문제이다. 최대한 많은 노드에 데이터를 하나하나 전파할 수록 일관성은 높아지지만 속도는 떨어지게 된다. 네트워크에 전파되는 데이터의 가용성 역시 ACID 이론이 설명하는 고립성(Isolated)특징에 따라 낮아진다.

이러한 CAP 이론의 한계를 극복하기 위하여 나온 것이 바로 PACELC 이론이다. PACELC (파셀크라고 읽는다)는 CAP 의 한계점을 보완하기 위하여 등장한 분류방법이다. PACELC 이론은 네트워크 노드 간의 통신이 불가능한 상황과 가능한 상황을 나누어서 판단할 수 있도록 한다. 만일 불가능한 상황에는 CAP 이론의 축을 따른다. 노드 통신이 가능한 경우에는 속도와 일관성을 배치되는 축으로 놓는다.

또한, Latency 와 Availability (Partition 이 없을 경우) 및 Consistency 와 Availability (Partition 이 있을 경우) 에 대해서 양자택일을 강요하기 보다는 하나의 스펙트럼으로 바라볼 수 있도록 하는 관점을 제공한다.

PACELC 에 따라 네트워크의 유형을 구분하려면 선제적으로 노드 간의 정상적 통신이 가능한 상태인지를 확인해야 한다. 분산 시스템에서 노드끼리 정보를 교환할 때 타임아웃(time-out)이 발생하는 모든 경우를 통신의 실패라고 분류한다. 분산 시스템에서 정상적인 통신이 원천적으로 불가능한 경우를 가정하는 것을 Partition 되어 있다고 표현한다. 만약 정상적인 통신이 가능하다는 전제가 있다면 네트워크에 Partition 이 일어나지 않았다고 표현할 수 있다.



PACELC 이론은 기존의 CAP 이론이 주창하는 축에 파란색 축을 하나 더 두었다. 이는 지연 시간과 일관성을 배치되는 축으로 두는 형태이다. 아래쪽에 위치할 수록 지연시간이 짧아지며, 위쪽에 위치할 수록 지연시간은 길어지고 일관성은 높아진다. 네트워크 노드 간의 통신이 원활한 상황에서 일관성에 치우칠수록 더 많은 데이터를 복제하여 전파해야 하기 때문에 지연시간이 그만큼 길어질 수밖에 없다.

PACELC 이론은 네트워크 장애 상황과 원활 상황에서 시스템이 어떻게 작동하는지에 따라 시스템을 PC/EC, PC/EL, PA/EC, PA/EL 로 나눈다. Siddhartra Reddy 에 의하면, MySQL 은 마스터-슬레이브로 구성 된 경우 PA/EL 가 기본적이고 경우에 따라 PA/EC 로 분류될 수 있다. 기본 값은 마스터(master)가 트랜잭션을 발생시키면 슬레이브(slave)에 비동기적으로 데이터를 복제한다(asynchronous replication). (기존 네트워크를 자세하게 분류해놓은 자료는 Siddhartra Reddy 의 발표자료를 참조)

PACELC 이론에 따른 분류는 다음과 같다.

- **PC/EC:** choosing consistency all the time (regardless of network partitions)
- **PC/EL:** choosing consistency when partition, and choosing latency otherwise
- **PA/EL:** sacrificing consistency all the time (regardless of network partitions)
- **PA/EC:** choosing consistency when no partition, and choosing latency otherwise

맺는 말

기술리서치 분과에서는 Proof-of-Authority(PoA)와 PBFT 를 CAP Theorem 을 기준으로 유형화하는 논문(PBFT vs proof-of-authority: applying the CAP theorem to permissioned blockchain: University of Southampton)을 리서치하였다. 리서치를 통해 해당 논문이 CAP 이론을 기반으로 PoA 와 PBFT 를 유형화한 것이 실제와 맞지 않다는 판단을 하였다. 예를 들어 논문에서는 PBFT 를 CP (Consistency-Partition Tolerance) 시스템으로 분류하였다. 그러나 PBFT 는 전체 노드가 동의하지 않아도 네트워크의 safety 를 위해서 투표 과정을 거친다. ⅔ 이상의 노드가 동의하면 자동적으로 블록이 확정된다.

따라서 PACELC 이론을 바탕으로 다시 분류하고자 하였으나, 리서치 주제 결이 맞지 않다고 판단하여 이를 시도하지 않았다. 추후 연구를 통해 각종 합의 알고리즘을 PACELC 이론에 바탕하여 분류하는 시도를 해보고자 한다.

Acknowledgments

홍종화 (분과장) hjh93411@gmail.com

분산 시스템에서의 합의 과저을 리서치하면서 블록체인에서 합의알고리즘이라고 명시하는 PoS, DPoS 와 BFT, PBFT 의 명확한 구분을 할 수 있었고, 문서로 정리하면서 국내에 부족하고 분산되어 있는 자료들을 한글화하여 정리할 수 있어서 의미깊은 시간이었다.

전창석 jcs191072@gmail.com

마지막으로 소회를 밝히는 시점이 다가오니 끝난다는 안도감과 아쉬움이 동시에 밀려온다. 약 한 달여 동안 수십 개의 블로그 글과 논문들을 읽고 정리하면서 많은 것을 배울 수 있는 시간이었지만, 더 만족스러운 결과물을 만들 수 있지 않았을까라는 아쉬움도 남는다. 혼자였다면 절대 이 긴 여정을 끝내지 못했을 것이다. 진형, 찬현, 부영, 종화, 동식님의 헌신적인 노력과 블록체인에 대한 열정에 감사와 찬사를 아낌없이 보내고 싶다.

이부형 na4980@gmail.com

4 기 기술 리서치 분과에서는 BFT Consensus 를 주제로 초기 기술 제안과 발전 과정에 대한 연구를 진행했습니다. 바쁜 일과 중에도 늦게까지 기술 토론과 리뷰에 힘써 주신 분과원들에게 감사의 인사를 드립니다. 분과 활동 기간 동안 많이 배울 수 있는 시간이었습니다. 또한 성공적인 Devstamp 행사 운영과 함께 이더리움 연구회가 앞으로 더 번창하길 바랍니다. 감사합니다.

이동식 dongsik.lee@gmail.com

이번 기회에 BFT Consensus 에 대한 명확한 이해를 할 수 있는 문서를 만들고자 하는 리서치 분과의 목표로 시작한 이번 과제는 종화, 부형, 진형, 찬현, 창석님들의 부단한 노력과 리서치의 결과물입니다.

아마도 새로 블록체인에 입문하시는 분이나, 그동안 어렴풋이 알고있지만 명확하게 이해하지 못했던 BGP, BFT 에 대한 제대로 된 이해를 할 수 있는 좋은 기회가 될 것입니다. 수많은 논문들을 일일이 리뷰하고 토론하고 정리해주신 종화, 부형, 진형, 찬현, 창석님께 감사드립니다.

박찬현 ryanpark045@gmail.com

블록체인 기술의 가장 근본인 비잔틴 장애 허용의 기초들을 리서치하고 이해할 수 있는 시간이 되어서 의미있었던것 같습니다. 특히 많은 부분들에서 거의 당연시될 정도로 여겼던 여러 상수들과 알고리즘들의 분석과 증명을 정리할 수 있었기에 추후 연구자들에게 도움이 될 수 있는 자료가 될 것 같습니다. 이번 4 기 기술 리서치 분과에서 정리한 자료를 통해 미래 블록체인 기술의 발전과 응용에 도움이 되기를 기원하며

박진형 hophfg@yahoo.co.kr

이더리움 연구회 4 기에서 블록체인을 진지하게 연구하려는 사람들과 함께할 수 있어서 매우 기뻤습니다. 암호화폐 시장은 하락장이고 블록체인 산업은 침체기에 있지만, 이 기술이 지니는 잠재력은 무궁무진하다고 믿습니다. 따라서 블록체인의 본질을 지속적으로 연구하는 것이 필요할 것이라고 봅니다. 4 기에서는 블록체인의 본질이라고 할 수 있는 분산 네트워킹 시스템에 대해서 진지하게 연구하는 시간을 가졌습니다. BFT Consensus 에 대한 전반적인 이해와 CAP PACLEC 이론까지 밑바닥을 리서치할 수 있었습니다. 이번 분과에서 하지 못한 연구 주제는 다음 분과에 계속하여 연구함으로써 블록체인에 대한 열정을 이어나가고자 합니다.

Reference

[1] Bitcoin: Peer-To-Peer Electronic Cash System, Satoshi Nakamoto, (2008)
https://bitcoin.org/bitcoin.pdf

[2] Cypherpunk, Wikipedia
https://en.wikipedia.org/wiki/Cypherpunk

[3] Bitcoin's Academic Pedigree, Arvind Narayanan and Jeremy Clark, (2017)
https://queue.acm.org/detail.cfm?id=3136559

[4] 거인의 어깨위에서 미래를 보다. 〈비트코인에 영향을 준 기술들과 블록체인의 현재와 미래〉, 박재현, 블로그
http://wisefree.tistory.com/504

[5] 그림으로 보는 IT 인프라 구조, 김완섭, 책, JPub, (2015)

[6] 코어 이더리움(Core Ethereum), 박재현, 박혜영, 오재훈, 책, JPub, (2018)

[7] Ethereum White Paper, Github
https://github.com/ethereum/wiki/wiki/White-Paper

[8] Impossibility of Distributed Consensus with One Faulty Process, Ficher, Lynch, Paterson, (1985)
https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf

[9] A Brief Tour of FLP Impossibility, The Paper Trail, Henry Robinson,(2008)
https://www.the-paper-trail.org/post/2008-08-13-a-brief-tour-of-flp-impossibility/

[10] Byzantine Generals Problem, Lamport, Shostak, Pease, (1982)
https://people.eecs.berkeley.edu/~luca/cs174/byzantine.pdf

[11] Akkoyunlu, Eralp A., Kattamuri Ekanadham, and Richard V. Huber. "Some constraints and tradeoffs in the design of network communications." ACM SIGOPS Operating Systems Review. Vol. 9. No. 5. ACM, 1975.

[12] Two Generals' Problem Explained, Finematics, video (2018)
https://www.youtube.com/watch?v=s8Wbt0b8bwY

[13] POW 를 통한 비잔틴 오류 허용 (비잔틴 장군 문제) 해결 과정, 블로그
http://goodjoon.tistory.com/256
[14] 슭의 개발 블로그: Safety & Liveness - FLP Impossibility 으로 보는 블록체인, 블로그
https://blog.seulgi.kim/2018/05/safety-liveness-in-blockchain.html
[15] Practical Byzantine Fault Tolerance, Castro, Liskov, 1999
http://pmg.csail.mit.edu/papers/osdi99.pdf
[16] Practical Byzantine Fault Tolerance by Miguel Castro and Barbara Lisko, Matthhias Eberli, Anna Yudina
https://www.slideshare.net/annayudi/pbft-7987140
[17] Practical Byzantine Fault Tolerance, Standford edu
http://www.scs.stanford.edu/14au-cs244b/notes/pbft.txt
[18] What are high and low water marks in bit streaming, Stackoverflow
https://stackoverflow.com/questions/45489405/what-are-high-and-low-water-marks-in-bit-streaming
[19] Tendermint github,
https://github.com/tendermint/tendermint
[20] Tendermint: Byzantine Fault Tolerance in the Age of Blockchains, Ethan Buchman, (2016)
https://atrium.lib.uoguelph.ca/xmlui/bitstream/handle/10214/9769/Buchman_Ethan_201606_MAsc.pdf?sequence=7&isAllowed=y
[21] 코스모스 사용자 커뮤니티, 웹사이트, 전창석, 이기호
www.cosmostalk.io
[22] Tendermint-starterkit, Github
https://github.com/blockchainvn/tendermint-starterkit
[23] L6: Byzantine Fault Tolerance, Distributed Systems Course
https://www.youtube.com/watch?v=_e4wNoTV3Gw

[24] PBFT vs proof-of-authority: applying the CAP theorem to permissioned blockchain: University of Southampton : https://eprints.soton.ac.uk/415083/

[25] Armando Fox and Eric A Brewer. Harvest, yield, and scalable tolerant systems. In 7th Workshop on Hot Topics in Operating Systems (HotOS), pages 174–178, March 1999. doi:10.1109/HOTOS.1999.798396.

[26] Eric A Brewer. CAP twelve years later: How the "rules" have changed. IEEE Computer Magazine, 45(2):23–29, February 2012. doi:10.1109/MC.2012.37.

[27] A Critique of the CAP Theorem, Martin Kleppmann https://arxiv.org/pdf/1509.05393.pdf

[28] Daniel J Abadi. Consistency tradeoffs in modern distributed database system design. IEEE Computer Magazine, 45(2):37– 42, February 2012. doi:10.1109/MC.2012.33.

[29] Replication and the latency-consistency tradeoff, Daniel Abadi, 블로그 http://dbmsmusings.blogspot.com/2011/12/replication-and-latency-consistency.html

[30] CAP and PACLEC: Thinking More Clearly About Consistency, Marc Brooker, 블로그 https://brooker.co.za/blog/2014/07/16/pacelc.html

[31] CAP Theorem: you don't need CP, you don't want AP, and you can't have CA, Siddhartha Reddy, Speaker Presentation https://speakerdeck.com/sids/cap-theorem-you-dont-need-cp-you-dont-want-ap-and-you-cant-have-ca?slide=38

[32] CAP Theorem, 오해와 진실: Jung-Haeng Lee, 블로그 http://eincs.com/2013/07/misleading-and-truth-of-cap-theorem/

[33] Please stop calling databases CP or AP, Martin Kleppmann, 블로그

https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html

[34] Proving PACLEC, University of Waterloo,

https://ece.uwaterloo.ca/~wgolab/research/proving_PACELC.pdf

[35] Perspectives on the CAP Theorem, Seth Gilbert, Nancy A. Lynch (2012)

https://groups.csail.mit.edu/tds/papers/Gilbert/Brewer2.pdf