

월러스(Walrus)

해당 번역본은 이용자가 디지털자산 관련 정보를 편리하게 확인할 수 있도록 단순 참고용으로 번역된 것이며, 투자 권유를 목적으로 하지 않습니다.

그 내용이 정확하지 않을 수 있으므로 정확한 정보 습득을 위해서는 원문을 참고하시거나 원문 작성 주체 측에 문의하시기 바랍니다.

두나무는 해당 번역본 내용의 진실성 및 정확성을 보장하지 않고, 제공된 정보에 의한 투자 결과에 대하여 어떠한 책임도 지지 않으며, 투자 손실은 투자자에게 귀속됩니다.

WALRUS: 효율적인 탈중앙화 스토리지 네트워크

The MystenLabs Team
hello@mystenlabs.com
v2.0 – 2025년 4월 11일

초록 (Abstract)

탈중앙화 스토리지 시스템은 복제 오버헤드, 복구 효율성, 보안 보장 간의 근본적인 트레이드오프에 직면해 있다. 기존 접근 방식은 전체 복제에 의존하여 상당한 저장 비용을 유발하거나, 복구 효율성이 떨어지는 단순한 삭제 부호(eraser coding) 방식에 의존하며, 특히 저장 노드의 변동(churn)이 클 때 이러한 문제가 두드러진다. 본 논문에서는 이러한 한계를 다수의 기술적 혁신을 통해 해결하는 새로운 탈중앙화 블록 스토리지 시스템인 *Walrus*를 제안한다.

*Walrus*의 핵심에는 *Red Stuff*가 있으며, 이는 이차원 삭제 부호(two-dimensional erasure coding) 프로토콜로서 4.5배의 복제 계수(replication factor)만으로 높은 보안을 달성하며, 손실된 데이터에 비례하는 대역폭만으로 자가 치유(self-healing) 복구를 가능하게 한다($O(|blob|/n)$), 기존 시스템에서는 $O(|blob|)$. 특히 *Red Stuff*는 비동기 네트워크 환경에서의 스토리지 챌린지를 지원하는 최초의 프로토콜로, 실제로 데이터를 저장하지 않고도 검증을 통과하려는 공격자가 네트워크 지연을 악용하는 것을 방지한다.

Walrus는 또한 다단계(epoch-based) 에폭 전환 프로토콜을 도입하여 저장 노드의 변동이 심한 환경에서도 위원회 전환 동안 연속적인 가용성을 유지한다. 본 시스템은 인증된 데이터 구조(authenticated data structures)를 통합하여 악의적인 클라이언트를 방어하고, 저장 및 검색 과정 전반에 걸쳐 데이터 일관성을 보장한다. 실험 결과 Walrus는 대규모 환경에서 실용적인 성능을 달성하며, 고무결성(high-integrity)과 높은 가용성을 요구하는 다양한 탈중앙화 애플리케이션에 적합한 블록 스토리지를 적정한 오버헤드로 제공한다.

서론 (Introduction)

블록체인은 상태 기계 복제(State Machine Replication, SMR) 패러다임 [37]을 통해 탈중앙화된 계산을 지원한다. 그러나 실제로는 운영에 필요한 데이터가 적은 분산 애플리케이션에만 실질적으로 제한된다. SMR은 모든 검증자 validator가 전체 데이터를 복제해야 하므로, 블록체인별 검증자 수에 따라 복제 계수가 100에서 1000까지 커지게 된다.

상태 계산을 위한 전체 데이터 복제는 실질적으로 필요하지만, 계산되지 않는 바이너리 대형 객체(blob)의 저장 및 검색만 필요한 애플리케이션의 경우 과도한 오버헤드를 유발한다¹. 이를 해결하기 위해 블록을 보다 효율적으로 저장할 수 있는 전용 탈중앙화 스토리지 네트워크 [6]가 등장했다. 예를 들어, 초기 네트워크인 IPFS [30]는 소수의 노드에만 복제함으로써 검열 저항성, 장애 발생 시 안정성 및 가용성을 제공한다 [47].

탈중앙화 블록 스토리지는 현대 탈중앙화 애플리케이션에서 필수적이다. 아래는 주요 활용 사례이다:

- 블록체인에서 관리되는 디지털 자산, 예를 들어 대체불가능토큰(NFT)은 탈중앙화 블록 스토리지가 제공하는 높은 무결성과 가용성 보장이 필요하다. 현재는 데이터가 온체인이 아닌 전통적 저장소에 저장되며 메타데이터만 보호되고, 실제 NFT 데이터는 브라우저에 따라 삭제되거나 위변조될 수 있다².
- 데이터 자산의 디지털 출처(provenance)는 AI 시대에서 점점 더 중요해지고 있다. 예를 들어, 문서 자료의 진위 확인, 학습 데이터셋의 조작 및 오염 방지, 특정 모델이 특정 데이터 인스턴스를 생성했는지 증명해야 하는 경우가 많다 [45]. 이러한 응용 사례는 탈중앙화 저장소가 제공하는 진위성, 추적 가능성, 무결성, 가용성의 이점을 활용할 수 있다.
- 웹 기반 또는 바이너리 형태의 탈중앙화 애플리케이션은 탈중앙화 저장소에서 배포될 필요가 있다. 현재 대부분의 탈중앙화 앱은 프론트엔드 및 클라이언트 측 코드를 전통적인 웹 호스팅에 의존하며, 이는 무결성과 가용성이 떨어진다. 탈중앙화 저장소는 웹 및 dApp 콘텐츠를 직접 서빙하면서 무결성과 가용성을 보장할 수 있으며, 소프트웨어의 바이너리 투명성을 확보하고 전체 재현 가능한 빌드 파이프라인의 저장 요구를 충족시켜 소프트웨어 감사를 위한 강력한 체인 오브 커스터디를 가능하게 한다 [22, 29].
- 탈중앙화 저장소는 이더리움의 현재 확장 전략인 롤업(roll-ups) [1]에서 데이터 가용성을 보장하는 데 핵심적인 역할을 한다. 이 구조에서는 저장 노드가 데이터를 임시로 보관하고, 블록체인 검증자가 이를 가져와 실행할 수 있도록 한다. 따라서 전체 업데이트 시퀀스(예: 거래)를 저장하는 대신, 롤업의 넷 상태(net state)에만 복제 비용이 부과된다.
- 탈중앙화 소셜 네트워크 플랫폼 [18]은 중앙화된 기존 플랫폼에 도전하고자 한다. 그러나 소셜 네트워크의 특성상, 긴 텍스트, 이미지, 영상 등 리치 미디어 사용자 콘텐츠에 대한 지원이 필요하다. 소셜을 넘어, 협업 플랫폼이나 시민 참여형 플랫폼 [4] 또한 공익 데이터 및 애플리케이션 데이터를 신뢰할 수

있는 중립적 저장소에 저장할 수 있는 수단이 필요하다.

- 마지막으로, 탈중앙화 저장소와 암호화 기술의 통합은 중요한 패러다임 전환을 이끈다 [19]. 이 접근법은 기밀성(Confidentiality), 무결성(Integrity), 가용성(Availability)을 갖춘 종합적 데이터 관리를 제공하며, 클라우드 서비스를 신뢰할 수탁자로 삼을 필요를 없앤다. 이는 주권형 데이터 관리, 탈중앙화 데이터 관리, 탈중앙화 데이터 마켓플레이스, 암호화된 데이터셋에 대한 계산 작업 등 수많은 가능성을 열어준다. 본 논문은 이러한 응용 사례를 중심으로 하지 않지만, 우리가 제안하는 탈중앙화 저장 시스템인 *Walrus*는 암호화된 블록의 저장 계층으로 자연스럽게 동작할 수 있다. 이 접근법은 계층형 구조를 제공하며, 암호화 레이어가 데이터 가용성 문제를 걱정하지 않고 안전하고 효율적인 키 관리 시스템(KMS)을 구현할 수 있도록 한다.

요약하자면, 보안성 있는 탈중앙화 블록 스토리지는 서로를 완전히 신뢰하지 않는 다수의 주체가 데이터를 공유하고 접근하는 모든 응용 환경에서 필수적이며, 높은 진위성, 무결성, 감사 가능성, 가용성을 신뢰할 수 있는 중립적 저장소를 통해 낮은 복잡성과 적절한 비용으로 제공되어야 한다.

탈중앙화 스토리지 접근 방식

탈중앙화 스토리지 프로토콜은 일반적으로 두 가지 주요 범주로 나뉜다. 첫 번째 범주는 *전체 복제(full replication)* 방식을 사용하는 시스템으로, 대표적인 예로 Filecoin [30]과 Arweave [46]가 있다. 이러한 시스템의 주요 장점은 저장 노드에 블록(blob)이 완전히 존재하므로, 저장 노드가 오프라인이 되더라도 데이터에 쉽게 접근할 수 있고 원활한 마이그레이션이 가능하다는 점이다. 이 구조는 파일 복구 시 다른 노드에 의존할 필요가 없으므로 *permissionless* 환경을 가능하게 한다. 그러나 이러한 시스템의 신뢰성은 선택된 저장 노드의 강건성에 크게 의존한다. 예를 들어, 고전적인 1/3 정적 공격자 모델과 무한한 수의 후보 저장 노드 풀을 가정할 경우, “99.9999999999%” 수준의 보안성(파일 접근 실패 확률이 10^{-12} 이하)을 달성하려면 네트워크에 25개 이상의 복제본을 저장해야 한다³. 이는 25배에 달하는 저장 오버헤드를 초래한다. 또한 Sybil 공격 [16]이라는 추가적인 문제가 존재한다. 이는 악의적인 행위자가 동일한 파일의 복제본을 여러 개 저장하고 있다고 가장함으로써 시스템의 무결성을 저해하는 공격이다.

두 번째 범주의 탈중앙화 저장 서비스 [23]는 *Reed-Solomon(RS) 인코딩* [32]을 사용한다. RS 인코딩은 복제 요구사항을 크게 줄일 수 있다. 예를 들어, 블록체인 작동 방식과 유사한 시스템에서 n 개의 노드 중 $1/3$ 이 악의적일 수 있고, 네트워크가 비동기적이라고 가정할 때, RS 인코딩은 단지 3배의 저장 오버헤드만으로 충분한 보안성을 달성할 수 있다. 이는 RS 인코딩이 파일을 *슬리버(slicer)*라고 불리는 더 작은 조각들로 나누기 때문이다. 이 조각들은 원본 파일의 일부를 나타내며, 이들 중 총합 크기가 원본보다 큰 슬리버 집합이 존재하면 해당 파일을 재구성할 수 있다.

그러나 삭제 부호 방식에는 문제점도 존재한다. 저장 노드가 오프라인 상태가 되어 다른 노드로 교체되어야 할 경우, 전체 복제 시스템에서는 단순히 다른 노드로 복사하면 되지만, RS 인코딩 시스템에서는 기존 모든 저장 노드가 해당 슬리버를 대체 노드로 전송해야 한다. 대체 노드는 전송받은 정보를 바탕으로 손실된 슬리버를 복구할 수 있지만, 이 과정에서 $O(|blob|)$ 에 해당하는 데이터가 네트워크를 통해 전송되어야 한다. 복구가 자주 발생할 경우, 복제 감소로 인한 저장 절감 효과가 무력화될 수 있으며, 이로 인해 이러한 시스템은 저장 노드의 변동(*churn*)이 적어야만 제대로 동작하므로 *permissionless* 특성이 약화된다.

복제 방식에 상관없이, 기존 모든 탈중앙화 저장 시스템은 또 다른 공통된 과제에 직면해 있다. 바로 저장 노드가 데이터를 유지하고 삭제하지 않도록 지속적으로 챌린지를 제공하여 인센티브를 부여해야 한다는 것이다. 이는 저장에 대해 보상을 제공하는 오픈 탈중앙화 시스템에서 매우 중요한 요소이며, 단순히 정직한/악의적인 참가자 모델을 넘어서는 구조가 요구된다. 현재 대부분의 솔루션은 네트워크가 동기적(synchronous)이라고 가정하며, 이로 인해 공격자가 정직한 노드로부터 누락된 데이터를 읽고 챌린지에 제시간에 응답하는 것을 방지할 수 있다고 본다.

WALRUS 소개

Walrus는 탈중앙화 블록 스토리지를 위한 새로운 접근 방식을 제안한다. 이는 삭제 부호(erasure code) 기반 아키텍처를 따르며, 수백 개의 저장 노드로 확장 가능한 구조를 통해 높은 복원력을 낮은 저장 오버헤드로 달성한다. Walrus의 핵심에는 새로운 인코딩 프로토콜인 Red Stuff가 있다. 이는 자가 치유(self-healing) 기능을 갖춘 이차원(2D) 인코딩 알고리즘을 사용한다. 구체적으로는 손실된 슬리버(slider)를 복구할 때 손실된 데이터의 양에 비례하는 대역폭만을 사용하도록 한다(우리 시스템에서는 $O(\frac{|\text{blob}|}{n})$). 또한 Red Stuff는 인증된 데이터 구조(authenticated data structures)를 통합하여 악의적인 클라이언트를 방어하고, 저장 및 검색되는 데이터의 일관성을 보장한다.

Red Stuff의 가장 독창적인 특징 중 하나는, 비동기 네트워크에서도 동작하며 스토리지 챌린지를 지원할 수 있다는 점이다. 이는 각 차원마다 서로 다른 인코딩 임계값(threshold)을 설정할 수 있도록 하는 이차원 인코딩 덕분에 가능하다. 쓰기(write) 흐름 중 심볼을 받지 못한 노드는 낮은 임계값 차원을 통해 누락된 데이터를 복구할 수 있고, 높은 임계값 차원은 챌린지 기간 동안 정직한 노드를 느리게 만들려는 공격자로부터 방어하면서 충분한 정보를 수집하여 챌린지에 응답할 수 있도록 한다.

Walrus와 일반적인 인코딩 기반 탈중앙화 저장 시스템이 직면하는 마지막 과제는, 서로 다른 저장 노드 위원회가 관리하는 에폭(epoch) 간에 시스템을 안전하게 운영하는 것이다. permissionless 시스템에서는 노드의 자연스러운 변동(churn)이 발생하므로, 읽기 및 쓰기 작업이 중단 없이 가능해야 한다. 그러나 데이터를 계속해서 곧 떠날 노드에 쓰게 되면, 해당 노드는 이를 후속 노드로 이전해야 하며, 이는 리소스를 놓고 경쟁하는 상황을 만든다. 이로 인해 어떤 노드는 쓰기 요청을 더 이상 수락하지 않거나, 책임 이전을 끝내지 못할 수 있다. Walrus는 이러한 문제를 탈중앙화 저장 시스템의 원칙에 잘 부합하는 다단계 에폭 전환 프로토콜을 통해 해결한다.

요약하면, 본 논문에서는 다음과 같은 공헌을 한다:

- 비동기 완전 데이터 공유(Asynchronous Complete Data-Sharing) 문제를 정의하고, Byzantine 장애 환경에서도 이를 효율적으로 해결할 수 있는 최초의 프로토콜인 Red Stuff를 제안한다 (3절).
- 낮은 복제 비용과 오류 및 노드 변동에 따른 데이터 손실을 효율적으로 복구할 수 있도록 설계된 최초의 permissionless 탈중앙화 스토리지 프로토콜 Walrus를 소개한다 (4절).
- Walrus가 Red Stuff를 활용하여 최초의 비동기 챌린지 프로토콜을 구현하는 방식을 제시한다 (4.6절).
- Walrus의 프로덕션 수준 구현을 제공하고, 공개 테스트넷을 배포하며, 실제 환경에서의 성능 및 확장성을 측정한다 (7절).

모델 및 정의

Walrus는 다음과 같은 가정에 기반한다.

암호학적 가정(Cryptographic assumptions). 본 논문에서는 `hash()`를 충돌 저항성을 갖는 해시 함수로 정의한다. 또한 보안 디지털 서명과 바인딩 커밋먼트(binding commitment)의 존재를 가정한다.

네트워크 및 적대적 가정(Network and adversarial assumptions). Walrus는 고정된 저장 노드 집합으로 구성된 에폭(epoch) 단위로 동작한다. 각 에폭의 종료 시점에는 $n = 3f + 1$ 개의 저장 노드가 저장 위원회(storage committee)의 구성원으로 선출되며, 각 노드는 하나의 저장 샤드(shard)를 담당한다. 악의적인 공격자는 이 중 최대 f 개의 노드를 제어할 수 있다고 가정한다.

손상된 노드는 프로토콜에서 임의로 이탈할 수 있으며, 나머지 노드는 정직하고 프로토콜을 엄격히 따르는 것으로 가정한다. 만약 에폭 e 에서 공격자가 제어하던 노드가 에폭 $e + 1$ 의 저장 노드 집합에 포함되지 않는다면, 공격자는 에폭 변경이 완료된 이후 새로운 노드를 타겟으로 적응형 공격을 수행할 수 있다.

Table 1: Comparing Replication Algorithms

	Replication for 10^{-12} Security	Write/Read Cost	Single Shard Recovery Cost	Asynchronous Challenges
Replication	25x	$O(n blob)$	$O(blob)$	Unsupported
Classic ECC	3x	$O(blob)$	$O(blob)$	Unsupported
RedStuff	4.5x	$O(blob)$	$O(\frac{ blob }{n})$	Supported

우리는 모든 정직한 노드 쌍이 신뢰 가능하고 인증된 채널에 접근할 수 있다고 가정한다. 네트워크는 비동기적이므로, 공격자는 정직한 노드 간의 메시지를 임의로 지연시키거나 순서를 바꿀 수 있지만, 에폭이 종료되기 전까지는 모든 메시지를 결국 전달해야 한다. 에폭이 종료된 경우에는 메시지가 폐기될 수 있다.

우리의 목표는 단순히 보안성 있는 탈중앙 시스템을 제공하는 것에 그치지 않고, 할당된 데이터를 보관하지 않는 저장 노드를 탐지하고 처벌하는 것이다. 이는 탈중앙화 저장 시스템에서 표준적으로 추가되는 가정으로, 정직한 참가자가 영구적으로 은밀하게 손상되는 상황을 방지하기 위함이다.

삭제 부호 (Erasure codes)

Walrus의 일환으로, 우리는 삭제 부호 방식을 사용하는 비동기 완전 데이터 저장(Asynchronous Complete Data Storage, ACDS)을 제안한다. 이 프로토콜의 핵심 구성요소에는 필요하지 않지만, 일부 최적화를 위해 인코딩 방식이 체계적(systematic)이라고 가정한다. 이는 인코딩의 입력 심볼(source symbols)이 출력 심볼에도 포함된다는 의미이다.

$Encode(B, t, n)$ 를 인코딩 알고리즘이라 하자. 이 알고리즘은 n 개의 심볼을 출력하며, 이 중 임의의 t 개를 이용해 B 를 복원할 수 있다. 이는 먼저 B 를 $O(\frac{|B|}{t})$ 크기의 t 개의 소스 심볼(source symbols)로 분할한 다음, $n - t$ 개의 복구 심볼(repair symbols)을 생성하여 총 n 개의 출력 심볼을 생성하는 방식으로 이루어진다. 디코딩 측면에서는, t 개 이상의 올바르게 인코딩된 심볼 집합 T 를 입력으로 $Decode(T, t, n)$ 을 호출하면 블록 B 를 복원할 수 있다.

블록체인 기반 구조 (Blockchain substrate)

Walrus는 모든 제어 연산을 위해 외부 블록체인을 블랙박스로 활용한다. 블록체인 프로토콜은 병렬로 발생하는 거래 집합을 입력 메시지 $Tx(M)$ 로 수용하고, 상태에 적용할 업데이트들을 총합 순서 $Res(seq, U)$ 로 출력하는 계산 블랙박스로 추상화할 수 있다. 우리는 이 블록체인이 이러한 추상 모델을 벗어나지 않으며, $Tx(M)$ 을 무기한 검열하지 않는다고 가정한다. 현대 고성능 SMR 프로토콜은 이러한 요건을 충족하며, 우리의 구현에서는 Sui [8]를 사용하였고, 핵심 Walrus 조정 프로토콜을 Move 스마트 계약 언어 [7]로 구현하였다.

비동기 완전 데이터 저장 (Asynchronous Complete Data Storage, ACDS)

우리는 먼저 분산 시스템에서의 완전 데이터 저장(Complete Data Storage) 문제를 정의하고, 비동기 네트워크를 위한 해결책으로 ACDS를 설명한다. 이어서, 그 정확성과 복잡성을 증명한다.

3.1 문제 정의 (Problem Statement)

요약하자면, 완전 데이터 저장 프로토콜은 작성자가 블롭(blob)을 저장 노드 네트워크에 기록할 수 있도록 하고 (쓰기 완전성), 저장 노드 내 일부 오류나 비잔틴 행위가 있더라도 읽는 사용자가 해당 데이터를 읽을 수 있도록 보장하며 (정당성), 작성자가 비잔틴일 수 있는 경우에도 읽기 결과가 일관되게 유지되도록 한다 (읽기 일관성). 보다 공식적으로는 다음과 같다:

정의 1 (Complete Data Storage). $n = 3f + 1$ 개의 노드로 구성된 네트워크가 주어졌고, 이 중 최대 f 개가 비잔틴일 수 있다고 하자. 작성자 W 가 블롭 B 를 이 네트워크에 저장하고자 하며, 이를 읽기 대상 독자 집합 R 에 공유하고자 할 때, Complete Data Storage 프로토콜은 다음 세 가지 속성을 보장한다:

- **쓰기 완전성 (Write Completeness):** 작성자 W 가 정직한 경우, 블롭 B 에 대한 커밋먼트를 가진 모든 정직한 노드는 결국 B 로부터 파생된 일부 조각 p 를 보유하게 되며, B 는 총 $O(\frac{|B|}{|p|})$ 개의 조각으로부터 복원 가능하다.
- **읽기 일관성 (Read Consistency):** 두 명의 정직한 독자 R_1 과 R_2 가 성공적으로 기록된 블롭 B 를 읽을 경우, 두 사람 모두 B 를 성공적으로 반환하거나, 모두 $\perp$ 를 반환해야 한다.
- **정당성 (Validity):** 정직한 작성자 W 가 B 를 성공적으로 기록한 경우, B 에 대한 커밋먼트를 보유한 정직한 독자 R 은 B 를 성공적으로 읽을 수 있어야 한다.

우리는 ACDS 프로토콜을 저장 노드 집합이 고정되어 있는 환경에서 먼저 설명하고, 이후 Walrus 내에서 이를 노드 위원회의 변동(churn)에 대응하여 어떻게 확장하는지 후속 절에서 다룬다.

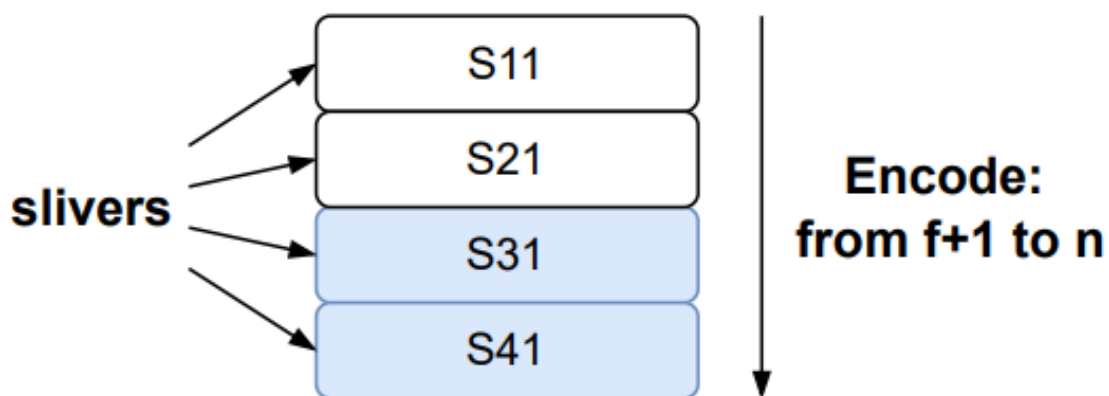
3.2 스트로우맨 설계 (Strawman Design)

이 절에서는 두 가지 스트로우맨(strawman) 설계를 순차적으로 살펴보고, 각 접근 방식의 비효율성을 논의한다.

스트로우맨 I: 전체 복제 (Full Replication). 가장 단순한 프로토콜은 Filecoin [30]과 Arweave [46] 방식과 유사한 전체 복제(full replication)를 사용한다. 작성자 W 는 자신의 블록 B 와 이 블록에 대한 바인딩 커밋먼트(예: $H_B = \text{hash}(B)$)를 모든 저장 노드에 브로드캐스트하고, 이후 $f + 1$ 개의 수신 확인 응답을 기다린다. 이러한 응답은 가용성 증명(availability certificate)을 구성하며, 최소한 하나의 응답이 정직한 노드로부터 발생했음을 의미하므로 가용성을 보장한다. 작성자 W 는 이 증명서를 블록체인에 게시할 수 있으며, 이로 인해 다른 모든 정직한 노드가 이를 인지하고 $\text{Read}(B)$ 요청을 통해 블록을 성공적으로 읽을 수 있다. 이 과정은 쓰기 완전성을 달성하며, 결국 모든 정직한 노드는 로컬에 블록 B 를 보유하게 된다. 나머지 속성들도 자명하게 만족된다. 독자는 절대로 $\perp$ 를 반환하지 않는다.

전체 복제 프로토콜은 단순하지만, 작성자는 $O(n|B|)$ 양의 데이터를 네트워크에 전송해야 하며, 이는 곧 전체 저장 비용과 같다. 또한 네트워크가 비동기적일 경우, 정합한 복제본을 확보하기 위해 최대 $f + 1$ 번의 요청이 필요하며, 이는 복구 중인 저장 노드 당 $O(n|B|)$ 의 비용, 전체적으로는 $O(n^2|B|)$ 의 네트워크 비용으로 이어진다. 마찬가지로, 읽기 작업조차도 비효율적일 수 있으며, 독자는 $f + 1$ 개의 요청을 보내야 할 수 있으므로 $O(n|B|)$ 의 비용이 발생할 수 있다.

스트로우맨 II: 인코딩 및 공유 (Encode & Share). 초기 데이터 전파 비용을 줄이기 위해, Storj [39]나 Sia [43]와 같은 일부 분산 스토리지 프로토콜은 RS 인코딩 [32]을 사용한다. 작성자 W 는 블록 B 를 $f + 1$ 개의 슬라이버로 분할하고, 추가로 $2f$ 개의 복구 슬라이버(repair slivers)를 인코딩한다. 인코딩 특성상, 아무 $f + 1$ 개의 슬라이버만으로도 B 를 복원할 수 있다. 각 슬라이버의 크기는 $O(\frac{|B|}{n})$ 이다. 이후 작성자 W 는 머클 트리 [27] 등의 바인딩 커밋먼트를 사용해 모든 슬라이버에 커밋하고, 각 노드에 별도의 슬라이버와 포함 증명(proof of inclusion)⁴을 전송한다. 노드는 자신의 슬라이버를 수신하고, 커밋먼트와 비교하여 확인한다. 슬라이버가 유효하면 커밋먼트를 서명하여 수신 확인 응답을 보낸다. 이후 작성자 W 는 $2f + 1$ 개의 서명을 수집하여 가용성 증명서를 생성하고, 이를 블록체인에 게시한다.



독자는 노드들로부터 유효한 슬라이버를 $f + 1$ 개 받을 때까지 반복적으로 요청을 전송한다(즉, 커밋먼트에 대해 검증 가능한 응답). 최소 $f + 1$ 개의 정직한 노드가 슬라이버를 저장하고 있으므로, 독자는 이를 반드시 수신할 수 있다. 독자는 받은 슬라이버로부터 블록 B 를 재구성한 후, 복원된 값을 다시 인코딩하고 커밋먼트를 재계산한다 [10, 27]. 만약 작성자 W 가 정직하다면, 재계산된 커밋먼트는 가용성 증명서의 커밋먼트와 일치하게 되며, 독자는 B 를 출력한다. 반면, 작성자 W 가 유효한 인코딩을 커밋하지 않았을 경우 커

밋먼트가 일치하지 않아, 독자는 $\$ \perp \$$ 를 반환하게 된다.

앞서와 마찬가지로, 공유 단계에서 슬리버를 받지 못한 노드들은 $\$B\$$ 를 읽는 방식으로 슬리버를 복구할 수 있다. 만약 읽기 결과가 $\$ \perp \$$ 일 경우, 해당 노드는 이후 모든 읽기에서도 $\$ \perp \$$ 를 반환한다. 그렇지 않은 경우에는 인코딩된 슬리버를 저장하고 나머지 $\$B\$$ 는 폐기한다. 이 복구 과정은 비용이 크며, 복구에 $\$O(|B|)\$$ 의 비용이 발생하지만, 저장 비용 자체는 이후 $\$O(\frac{|B|}{n})\$$ 에 불과하다.

이 두 번째 프로토콜은 인코딩/디코딩 및 슬리버 커밋에 필요한 추가 연산을 감수하면서 데이터 전파 비용을 크게 줄인다. 블록 $\$B\$$ 를 전파하는 데 $\$O(|B|)\$$ 의 비용만 발생하며 ⁵, 이는 읽기 비용과 동일하다. 하지만 여전히 전체 분산 비용은 $\$O(n|B|)\$$ 로 남는데, 이는 누락된 슬리버를 복구하는 과정에서 전체 블록 $\$B\$$ 를 다운로드해야 하기 때문이다. 작성자 $\$W\$$ 로부터 슬리버를 수신하지 못한 저장 노드가 최대 $\$f\$$ 개까지 존재할 수 있으며, 이들이 복구 프로토콜을 실행해야 한다. 따라서 총 비용은 $\$O(n|B|)\$$ 이다. 이러한 특성은 초기 전파뿐만 아니라, 에폭 경계(epoch boundaries)에서 저장 노드 집합이 변경되는 경우에도 중요하다. 새로운 노드 집합은 이전 저장 노드들로부터 슬리버를 복구하여야 하기 때문이다.

3.3 최종 설계: RED STUFF

위에서 설명한 인코딩 프로토콜은 매우 높은 보장 하에 낮은 저장 오버헤드를 달성하지만, 여전히 장기적이고 대규모로 운영되는 시스템에는 적합하지 않다. 주요 문제는 장기간 운영되는 대규모 시스템에서는 저장 노드가 주기적으로 장애를 겪거나, 자신이 보유한 슬리버를 유실하고 교체되어야 한다는 점이다. 또한 permissionless 시스템에서는 충분한 인센티브가 제공되더라도 저장 노드에 자연스러운 churn(변동)이 존재한다.

이러한 상황은 모두 새로 참여한 저장 노드를 위해 전체 데이터를 복구해야 하므로, 저장된 전체 데이터 크기만큼의 데이터를 네트워크 상에서 전송해야 하는 결과를 초래한다. 이는 현실적으로 감당할 수 없을 만큼 비싸다. 대신, 우리는 시스템이 자가 치유(self-healing) 기능을 갖추어, churn 상황에서의 복구 비용이 오직 복구 대상 데이터의 양에만 비례하고, $\$n\$$ 에 반비례하도록 만들기를 원한다.

이를 위해 *Red Stuff*는 블록(blob)을 이차원으로 인코딩한다(2D 인코딩). 1차원(primary) 차원은 기존 시스템에서 사용된 RS 인코딩과 동일하다. 하지만 $\$B\$$ 의 슬리버를 효율적으로 복구할 수 있도록 하기 위해 2차원(secondary) 인코딩도 적용한다. *Red Stuff*는 선형 삭제 부호(linear erasure coding, 2절 참조)와 Twin-code 프레임워크 [31]에 기반한다. Twin-code는 신뢰된 작성자 환경에서 장애 허용을 고려한 삭제 부호 저장소 및 효율적인 복구 방식을 제공한다. 우리는 이 프레임워크를 비잔틴 장애 허용 환경에 적합하도록 수정하고, 하나의 저장 노드 집합(single committee)만으로 동작 가능하게 하였으며, 여기에 다양한 최적화 기법들을 추가하였다(아래에서 자세히 설명한다).

인코딩 (Encoding)

출발점은 두 번째 스트로우맨 설계이다. 이 설계에서는 블록을 $\$f + 1\$$ 개의 슬리버로 분할한다. 단순히 복구용 슬리버만 인코딩하는 대신, 여기서는 추가적인 차원을 인코딩 과정에 도입한다. 즉, 원래의 블록은 $\$f + 1\$$ 개의 **1차 슬리버(primary slivers)**(도식에서는 수직 방향)와 $\$2f + 1\$$ 개의 **2차 슬리버(secondary slivers)**(수평 방향)로 분할된다. 이 과정을 Figure 2에서 보여준다. 그 결과, 파일은 $\$(f + 1)(2f + 1)\$$ 개의 심볼로 나뉘며, 이는 $\$[f + 1, 2f + 1]\$$ 크기의 행렬로 시각화할 수 있다.

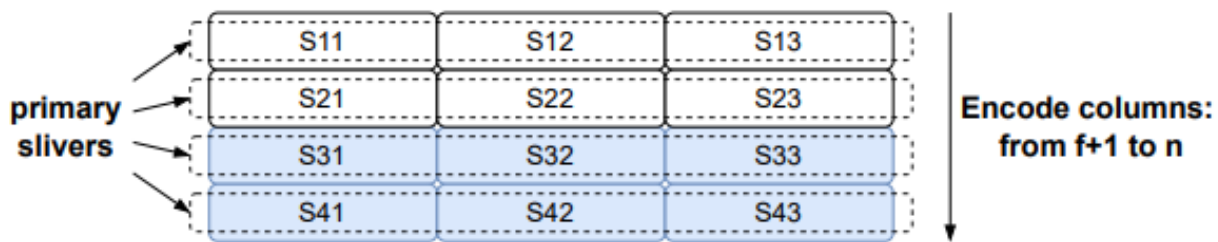
이 행렬을 기반으로 두 차원에서 복구용 심볼(repair symbols)을 생성한다. 먼저, $2f + 1$ 개의 열(각 열의 길이는 $f + 1$)을 각각 n 개의 심볼로 확장하여 총 n 개의 행을 생성한다. 각 행은 해당 노드의 **1차 슬리버(primary sliver)**로 할당된다 (Figure 2a 참조). 이 방식은 우리가 전송해야 할 데이터 총량을 거의 세 배 증가시키며, 앞서 3.2절의 1차원 인코딩과 유사한 수준의 부하를 갖는다.

각 슬리버를 효율적으로 복구하기 위해, 초기의 $[f + 1, 2f + 1]$ 행렬을 기준으로 $f + 1$ 개의 행(각 행의 길이는 $2f + 1$)도 마찬가지로 복구 심볼을 생성하여 n 개의 심볼로 확장한다 (Figure 2b 참조). 이로써 n 개의 열이 생성되며, 이들은 각각 해당 노드의 **2차 슬리버(secondary sliver)**로 할당된다.

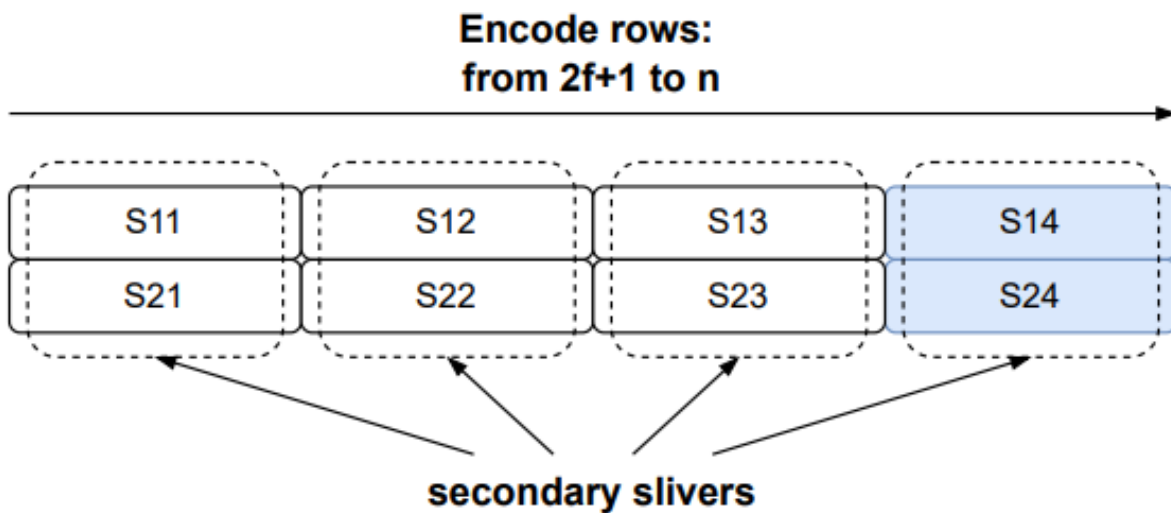
메타데이터 처리 (Handling Metadata)

각 슬리버(primary 및 secondary)에 대해 작성자 W 는 해당 심볼들에 대한 벡터 커밋먼트(vector commitments)를 계산한다. 각 1차 슬리버에 대해서는 확장된 행의 모든 심볼에 커밋하고, 각 2차 슬리버에 대해서는 확장된 열의 모든 심볼에 커밋한다. 마지막 단계로, 클라이언트는 이들 슬리버 커밋먼트의 목록에 대해 하나의 커밋을 수행하며, 이를 **블롭 커밋먼트(blob commitment)**라고 한다.

이러한 슬리버별 벡터 커밋먼트들은 **블롭 메타데이터(blob metadata)**를 형성한다. 이를 활용하면, 노드는 향후 특정 심볼에 대한 쿼리를 받았을 때, 자신이 반환하는 심볼이 실제로 작성자에 의해 작성된 것임을 증명할 수 있다. 단, 이러한 증명을 위해서는 해당 슬리버에 대한 커밋먼트와 블롭 커밋먼트(그리고 이 커밋이 해당 슬리버 커밋에 대응됨을 보여주는 증명)를 모두 열람할 수 있어야 한다.



(a) 이차원에서의 1차 인코딩. 파일은 $2f + 1$ 개의 열과 $f + 1$ 개의 행으로 분할된다. 각 열은 $2f$ 개의 복구 심볼을 포함하는 별도의 블롭으로 인코딩된다. 이후 각 확장된 행은 해당 노드의 1차 슬리버가 된다.



(b) 이차원에서의 2차 인코딩. 파일은 $2f + 1$ 개의 열과 $f + 1$ 개의 행으로 분할된다. 각 행은 f 개의 복구 심볼을 포함하는 별도의 블록으로 인코딩된다. 이후 각 확장된 열은 해당 노드의 2차 슬라이버가 된다.

모든 슬라이버를 보유한 노드는 자신의 슬라이버 커밋먼트 및 오프닝을 쉽게 재계산할 수 있지만, 블록 커밋먼트를 열람하려면 전체 노드의 슬라이버 커밋먼트가 모두 필요하다.

보안을 갖춘 자가 치유(self-healing)를 가능하게 하기 위해 이러한 메타데이터를 모든 저장 노드에 단순 복제할 경우, 노드 수에 대해 이차적인(quadratic) 오버헤드가 발생하게 된다. 왜냐하면 각 노드가 모든 다른 노드의 슬라이버 커밋먼트를 저장해야 하기 때문이다. 특히 작은 블록의 경우, 이러한 상대적 오버헤드는 매우 커질 수 있다. 예를 들어, 1000개의 노드로 구성된 시스템에서 32바이트 해시를 사용할 경우, 각 노드는 64KB의 추가 데이터를 저장해야 하며, 전체적으로는 64MB의 오버헤드가 발생한다.

이러한 오버헤드를 줄이기 위해, 저장 노드들은 메타데이터의 인코딩된 버전을 유지한다. 저장 노드는 쓰기(write) 또는 복구(recovery) 과정을 수행할 때 메타데이터 전체를 받아야 하므로, 클라이언트가 별도로 인코딩하거나 2차원 인코딩을 수행할 필요는 없다. 대신 저장 노드는 단순히 메타데이터를 로컬에서 1차원 $(f+1) \times n$ 인코딩 방식으로 인코딩하고, 자신에게 할당된 샤드만 보관하면 된다⁶. 이를 통해 전체 시스템의 오버헤드는 이차적(quadratic)에서 선형(linear) 수준으로 줄어들며, 노드당 일정한 상수 수준의 오버헤드만 발생하게 된다.

쓰기 프로토콜 (Write Protocol)

*Red Stuff*의 쓰기 프로토콜은 RS 코드 프로토콜과 동일한 패턴을 따른다. 작성자 W 는 먼저 블록을 인코딩하고, 각 노드에 대해 슬라이버 쌍(sliver pair)을 생성한다. 슬라이버 쌍 S_i 는 S_i 번째 1차 및 2차 슬라이버의 쌍이다. $n = 3f + 1$ 개의 노드가 존재하므로, 슬라이버 쌍도 n 개가 된다.

그런 다음 W 는 모든 슬라이버 커밋먼트와 해당 노드의 슬라이버 쌍을 모든 노드에게 전송한다. 각 노드는 수신한 슬라이버가 커밋과 일치하는지 확인하고, 블록 커밋먼트를 재계산한 후, 서명된 수신 확인 응답을 보낸다. 작성자 W 는 $2f + 1$ 개의 서명을 수집한 후 인증서(certificate)를 생성하여 온체인에 게시함으로써 해당 블록이 가용함을 공식화(certify) 한다.

이론적인 비동기 네트워크 모델에서 신뢰 가능한 전송이 보장된다면, 위 프로토콜은 결국 모든 정직한 노드가 정직한 작성자로부터 슬리버 쌍을 수신하게 된다. 그러나 실제 프로토콜에서는 작성자가 재전송을 중단해야 할 수도 있다. $2f + 1$ 개의 서명이 수집된 이후에는 재전송을 중단해도 안전하며, 이 경우 최소 $f + 1$ 개의 정직한 노드가 해당 블록에 대한 슬리버 쌍을 보유하게 된다.

읽기 프로토콜 (Read Protocol)

읽기 프로토콜은 RS 코드와 동일한 방식이다. 비동기 챌린지를 지원하기 위해, 노드들은 2차 슬리버만을 사용한다. 만약 비동기 챌린지가 필요하지 않다면 1차 슬리버를 사용하여 더 낮은 복원 임계값($f + 1$)으로 복원할 수 있다.

독자 R 은 먼저 메타데이터를 수집한다. 즉, 블록 커밋먼트에 대한 슬리버 커밋먼트들의 목록을 수신해야 한다. 이를 위해 R 은 이웃 노드들로부터 1차원으로 인코딩된 메타데이터 조각들과 오프닝 증명을 요청한다.

메타데이터가 디코딩된 후, R 은 수신된 커밋 목록이 블록 커밋먼트와 일치하는지 확인한다. 그 다음 R 은 모든 노드에 대해 블록 커밋먼트에 대한 읽기 요청을 보내고, 노드들은 자신이 보유한 2차 슬리버를 응답으로 반환한다(대역폭을 절약하기 위해 점진적으로 전송될 수 있음). 각 응답은 해당 커밋과 비교하여 검증된다. $2f + 1$ 개의 유효한 2차 슬리버가 수신되면, R 은 이를 이용해 B 를 디코딩하고, 다시 인코딩하여 블록 커밋먼트를 재계산한다. 이 커밋이 작성자 W 가 온체인에 게시한 커밋과 동일하다면, R 은 B 를 출력하고, 그렇지 않다면 $\perp$ 를 출력한다.

슬리버 복구 (Sliver Recovery)

*Red Stuff*가 RS 코드 방식과 비교해 갖는 주요 장점은 **자가 치유(self-healing)** 기능이다. 이는 작성자 W 로부터 슬리버를 직접 받지 못한 노드가 해당 슬리버를 복구하고자 할 때 발휘된다. 어떤 저장 노드는 자신의 2차 슬리버를 복구하기 위해, 자신의 행에 존재하는 심볼을 $f + 1$ 개의 저장 노드에 요청할 수 있다. 이 심볼들은 요청 노드의 확장된 열에도 존재해야 한다 (Figure 3b, 3c 참조). 이 과정을 통해 결국 $2f + 1$ 개의 정직한 노드는 모두 2차 슬리버를 보유하게 된다.

이후, 어떤 노드든 $2f + 1$ 개의 정직한 노드에 자신의 열에 해당하는 심볼을 요청함으로써 1차 슬리버도 복구할 수 있다 (Figure 3d 참조). 이 심볼은 요청 노드의 확장된 행에도 존재해야 한다. 이때 응답 노드는 요청된 심볼에 대한 커밋 오프닝 증명도 함께 전송한다. 이를 통해 수신 노드는 자신이 수신한 심볼이 작성자 W 가 작성한 올바른 심볼임을 검증할 수 있으며, 이는 W 가 정직할 경우 디코딩의 정확성을 보장한다.

각 심볼의 크기는 $O(\frac{|B|}{n^2})$ 이고, 각 저장 노드는 총 $O(n)$ 개의 심볼을 다운로드하므로, 노드당 복구 비용은 $O(\frac{|B|}{n})$ 이고, 전체 파일 복구 비용은 $O(|B|)$ 로 읽기/쓰기 비용과 동일하다. 결과적으로, *Red Stuff*를 사용하면 프로토콜의 통신 복잡도는 (거의 ⁷) n 과 무관하게 되어, **프로토콜의 확장성**이 확보된다.

RED STUFF는 ACDS이다

5장에서 *Red Stuff*가 ACDS의 모든 속성을 만족함을 증명한다. 비공식적으로 설명하면, 쓰기 완전성 (Write Completeness)은 정직한 작성자가 최소한 $f + 1$ 개의 정직한 노드가 슬리버 쌍을 수신했음을 확인한 후에 재전송을 멈춘다는 점에서 보장된다. 슬리버 복구 알고리즘은 남아 있는 정직한 노드들이 이러한 노드들로부터 효율적으로 슬리버를 복구할 수 있도록 하며, 결국 모든 정직한 노드가 자신에게 해당하는 슬리버를 보유하거나 인코딩이 잘못되었음을 증명할 수 있다. 정당성(Validity)은 $2f + 1$ 개의 정직한 노드가 결국 올바른 슬리버 쌍을 보유하게 되므로 보장되며, 모든 노드에 요청을 보낸 독자는 충분한 슬리버를 획득하여 블록을 복구할 수 있게 된다. 읽기 일관성(Read Consistency)은 두 정직한 독자가 서로 다른 슬리버 집합으로부터 블록을 디코딩하고 다시 인코딩한 후, 인코딩의 일치 여부를 확인함으로써 보장된다. 만약 인코딩이 올바르면 둘 다 동일한 블록을 출력하고, 그렇지 않으면 둘 다 $\perp$ 를 출력한다.

S11	S12	S13	S14
S21		S23	
S31	S32	S33	
S41		S43	

(a) Nodes 1 and 3 collectively hold two rows and two columns

S11	S12	S13	S14
S21		S23	
S31	S32	S33	S34
S41		S23	

(b) Each node sends the intersection of their row/column with the column/row of Node 4 to Node 4 (Red). Node 3 needs to encode the row for this.

S11	S12	S13	S14
S23	S22	S23	S24
S31	S32	S33	S34
S41	S42	S43	S44

Recover from $f+1$

(c) Node 4 uses the $f + 1$ symbols on its column to recover the full secondary sliver (Green). It will then send any other recovering node the recovered intersections of its column to their row.

S11	S12	S13	S14
S23	S22	S23	S24
S31	S32	S33	S34
S41	S42	S43	S44

Recover from $2f+1$

(d) Node 4 uses the $f+1$ symbols on its row as well as all the recovered secondary symbols send by other honest recovering nodes (Green) (which should be at least $2f$ plus the 1 recovered in the previous step) to recover its primary sliver (Dark Blue)

4 WALRUS: 탈중앙 보안 블록 저장소

*Walrus*는 메타데이터 및 거버넌스를 위한 제어 평면(control plane)으로 블록체인을 통합하고, 블록 데이터의 인코딩 및 디코딩은 별도의 저장 노드 위원회가 담당하는 구조이다. 이 아키텍처는 3.3절에서 설명한 *Red Stuff* 인코딩/디코딩 알고리즘과, 머클 트리(Merkle trees) [27]를 집합 커밋(set commitment)으로, Sui 블록체인 [8]을 기반으로 한다. 다만 *Walrus*는 2절에서 명시된 최소 요건을 충족하는 한, 어떤 블록체인과 인코딩/디코딩 알고리즘으로도 일반화가 가능하다.

우리는 먼저 단일 에폭(epoch) 내 *Walrus*의 흐름을 설명하고, 이후 재구성을 통해 저장 노드의 동적 가용성을 어떻게 지원하는지를 논의한다. 마지막으로, 단순한 정직/악의 모델을 넘어 *스토리지 챌린지(storage challenge)*를 제공하는 방식을 살펴본다. 하나의 에폭 내에서 *Walrus*와 클라이언트 간 상호작용은 (a) 블록 쓰기과 (b) 블록 읽기로 이루어진다.

4.1 블록 쓰기 (Writing a Blob)

*Walrus*에서 블록을 쓰는 과정은 알고리즘 3과 그림 4에 나타나 있다.

해당 프로세스는 작성자가 (❶) Figure 2에서 보인 것처럼 *Red Stuff*를 이용해 블록을 인코딩하면서 시작된다. 이 과정에서 슬리버 쌍(silver pairs), 슬리버에 대한 커밋 목록, 그리고 블록 커밋(blob commitment)이 생성된다. 작성자는 파일 길이, 인코딩 유형 등의 메타데이터와 함께 블록 커밋을 해시하여 *블록 식별자(blob id)* \$id_B\$를 파생시킨다.

그 다음 작성자는 (❷) 블록체인에 거래를 제출하여 블록을 여러 에폭에 걸쳐 저장할 수 있는 충분한 공간을 확보하고, 블록을 등록(register)한다. 이때 블록의 크기 및 커밋먼트가 전송되며, 이를 통해 \$id_B\$를 다시 파생할 수 있다. 블록체인 스마트 계약은 각 노드에 인코딩된 슬리버를 저장할 공간뿐 아니라, 해당 블록에 대한 커밋과 관련된 모든 메타데이터를 저장할 수 있는 공간을 확보해야 한다. 이 거래에는 공간 확보를 위한 결제가 함께 이루어질 수 있으며, 또는 다가오는 에폭의 빈 공간 리소스를 이 요청에 할당할 수도 있다. 우리의 구현은 이 두 가지 옵션을 모두 지원한다.

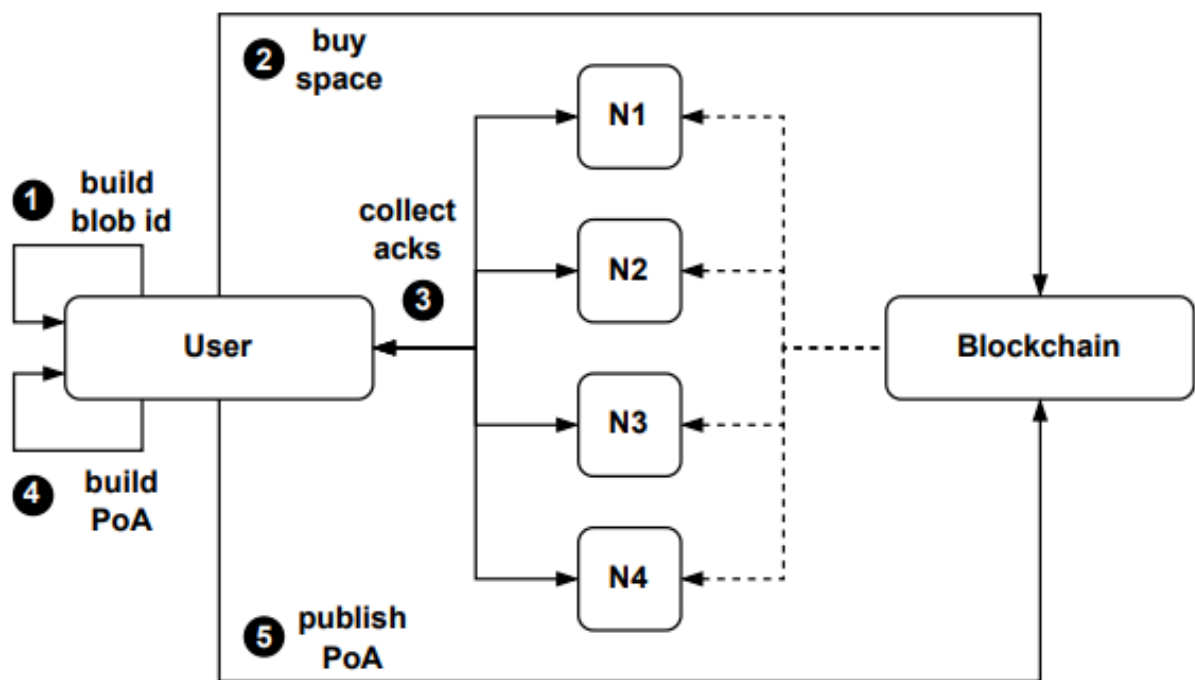


Figure 4: Walrus의 쓰기 흐름. 사용자는 저장할 파일의 블록 ID를 생성하고, 블록체인을 통해 저장 공간을 확보하고, 인코딩된 파일을 Walrus에 제출하고, $2f + 1$ 개의 수신 확인을 수집한 후, 그것을 가용성 증명으로 블록체인에 제출한다.

등록 트랜잭션이 커밋되면 (❸), 작성자는 저장 노드에게 \$id_B\$로 식별되는 블록의 슬리버를 저장할 의무가 있음을 알린다. 이때 해당 트랜잭션과 함께 커밋먼트, 각 저장 노드에 할당된 1차 및 2차 슬리버, 그리고 해당 슬리버가 게시된 \$id_B\$와 일치함을 증명하는 증거들을 함께 전송한다. 저장 노드는 커밋먼트를 검증한 후, 커밋과 슬리버 쌍을 저장하면 \$id_B\$에 대해 서명된 수신 확인 응답을 보낸다.

마지막으로 작성자는 $2f + 1$ 개의 서명된 수신 확인 응답을 수집할 때까지 기다린다 (❹). 이들은 하나의 쓰기 인증서(write certificate)를 구성하며, 이 인증서는 온체인에 게시된다 (❺). 이는 Walrus 내에서 해당 블록이 가용성 시점(Point of Availability, PoA)에 도달했음을 나타낸다. PoA는 저장 노드가 명시된 에폭 동안 읽기를 위해 슬리버를 유지할 의무가 있음을 의미한다. 이 시점에서 작성자는 로컬 저장소에 있

는 블록을 삭제하고 오프라인으로 전환할 수 있다. 또한 이 PoA는 작성자가 제3자 사용자나 스마트 계약에 해당 블록의 가용성을 증명하는 수단으로 활용될 수 있다.

노드들은 블록이 PoA에 도달했다는 이벤트를 감지하기 위해 블록체인을 모니터링한다. 만약 해당 블록에 대한 슬리버 쌍을 보유하고 있지 않다면, PoA를 지난 모든 블록의 커밋먼트와 슬리버 쌍을 복구하기 위한 복구 프로세스를 수행한다. 이 과정을 통해 결국 모든 정직한 노드가 모든 블록에 대한 슬리버 쌍을 보유하게 된다.

4.2 블록 읽기 (Reading a Blob)

읽기 경로에서, 독자는 어떤 저장 노드에든지 $\$id_B\$$ 로 식별되는 블록의 커밋먼트 및 2차 슬리버를 요청할 수 있다 (1). $\$2f + 1\$$ 개의 유효한 증명과 함께 응답을 수신하면 (2), 블록을 재구성할 수 있다. 이후 (3) 독자는 블록을 재인코딩하고 블록 ID $\$id_B\$$ 를 재계산한다. 만약 $\$id_B = id_B\$$ 이면 블록을 출력하고, 그렇지 않으면 블록이 일관되지 않다고 판단하여 $\$perp\$$ 를 출력한다.

*Red Stuff*의 속성 덕분에 읽기는 모든 독자에게 일관되게 수행된다. 오류가 발생하지 않는 경우, 전체적으로 원본 블록보다 약간 큰 크기의 슬리버 데이터만 다운로드하면 된다.

4.3 슬리버 복구 (Recovery of Slivers)

비동기 네트워크에서 블록을 기록할 때, 또는 노드가 크래시 후 복구되는 경우, 모든 노드가 쓰기 도중 슬리버를 받을 수 있는 것은 아니다. 이 점은 완전성(completeness)을 필요로 하지 않는 프로토콜이라면 문제가 되지 않는다. 그러나 *Walrus*는 이차원 인코딩 방식을 사용하여 완전성을 확보하기로 하였다. 즉, 모든 정직한 저장 노드는 PoA 이후 모든 블록에 대해 슬리버를 복구하고 보유할 수 있어야 한다. 이는 (1) 읽기 요청의 부하를 더 잘 분산시키고 — 모든 노드가 응답 가능하게 하며, (2) 저장 노드의 동적 가용성 확보 — 모든 블록을 재구성하거나 재작성하지 않고도 재구성이 가능하며, (3) 부분 저장 증명을 위한 최초의 완전 비동기 프로토콜 — 4.6절에서 설명됨 — 을 가능하게 한다.

이러한 모든 이점은 저장 노드가 슬리버를 효율적으로 복구할 수 있어야만 실현된다. 이 프로토콜은 Figure 3에 설명된 *Red Stuff* 복구 프로토콜을 그대로 따른다. 저장 노드가 자신이 슬리버를 수신하지 못한 블록의 인증서를 감지하면, 나머지 저장 노드들로부터 자신의 슬리버 쌍을 복구하려 시도한다. 이때 복구 노드는 자신의 1차/2차 슬리버와 서명 노드들의 2차/1차 슬리버 간 교차점에 해당하는 심볼을 모든 저장 노드에게 요청한다. 인증서에 서명한 노드가 $\$2f + 1\$$ 개이므로, 이 중 최소 $\$f + 1\$$ 개는 정직하며 응답할 것이다. 이로써 모든 $\$2f + 1\$$ 정직한 노드가 2차 슬리버를 보유하게 된다.

그 결과, 모든 정직한 노드가 2차 슬리버를 보유하게 되면, 해당 노드들은 복구 노드의 1차 슬리버에 해당하는 심볼을 공유할 수 있으며, 복구 노드는 $\$2f + 1\$$ 개의 응답을 수신하여 1차 슬리버 또한 복구할 수 있게 된다.

4.4 악의적인 작성자의 비밀관된 인코딩 처리

마지막으로 다루어야 할 사례는, 클라이언트가 악의적이어서 블록의 올바른 인코딩과 일치하지 않는 슬리버를 업로드하는 경우이다. 이 경우, 노드는 자신이 수신한 심볼로부터 커밋먼트와 일치하는 슬리버를 복구할 수 없게 된다. 하지만 이 경우, $\$id_B\$$ 에 연관된 제3자가 검증 가능한 불일치 증명(**proof of inconsistency**)이 반드시 생성된다.

정직한 독자에 의해 수행되는 읽기 프로세스는 기본적으로 불일치한 인코딩을 거부하기 때문에, 일관된 읽기를 보장하기 위해 반드시 이 증명을 공유할 필요는 없다. 하지만, 해당 불일치에 대해 노드들이 동의하게 되면, 해당 블록의 데이터를 삭제하고 이후 챌린지 프로토콜(4.6절)에서 제외시킬 수 있게 된다.

불일치를 증명하기 위해, 저장 노드는 복구를 위해 수신한 심볼과 해당 포함 증명(inclusion proof)으로 구성된 불일치 증명을 다른 노드들과 공유한다. 다른 노드들은 이를 통해 자체적으로 복구 시뮬레이션(trial recovery)을 수행하여 증명을 검증할 수 있다. 이러한 사기 증명을 검증한 후, 해당 노드는 온체인에 $\$id_B\$$ 가 무효임을 선언한다. $\$f + 1\$$ 개의 이러한 선언이 온체인에서 확인되면, 모든 노드는 해당 블록의 슬라이버 요청에 대해 $\$perp\$$ 를 반환하고, 온체인 불일치 증거를 가리키는 참조를 함께 제공한다.

4.5 위원회 재구성 (Committee Reconfiguration)

Walrus는 탈중앙화 프로토콜이므로, 에폭(epoch) 간 저장 노드 집합이 변동하는 것은 자연스러운 현상이다. 새로운 위원회가 기존 위원회를 교체하는 시점에서 **재구성(reconfiguration)**이 발생한다. 재구성 프로토콜의 목적은 저장 노드 집합이 변경되더라도 **PoA 이후 모든 블록이 계속해서 가용하게 유지되는 불변 조건**을 보장하는 것이다. 물론 각 에폭마다 최소 $\$2f + 1\$$ 개의 노드가 정직하다는 가정 하에서다. 저장 노드 간 많은 양의 데이터를 이전해야 할 경우, 재구성에는 수 시간이 걸릴 수 있다. 이 기간 동안에도 Walrus는 블록에 대한 읽기 및 쓰기 작업을 계속 수행해야 하며, 중단이 발생해서는 안 된다.

핵심 설계 (Core Design)

상위 수준에서 Walrus의 재구성 프로토콜은 블록체인 시스템의 재구성 방식과 유사하다. Walrus 역시 저장 노드의 쿼럼을 기반으로 동작하기 때문이다. 그러나 Walrus의 재구성은 독자적인 과제를 수반한다. 왜냐하면 상태(state)의 마이그레이션이 기존 블록체인 시스템에 비해 훨씬 많은 비용이 들기 때문이다.

주요 문제는 에폭 $\$e\$$ 에서 블록을 기록하는 것과, 재구성 이벤트 도중 $\$e\$$ 에서 $\$e+1\$$ 로 넘어갈 때 기존 저장 노드로부터 신규 저장 노드로 슬라이버를 이전하는 과정 간의 **경쟁 상태(race condition)**이다. 구체적으로, 에폭 $\$e\$$ 동안 기록된 데이터의 양이 기존 노드가 새 노드로 전송할 수 있는 능력을 초과할 경우, 에폭이 종료되지 못하게 된다.

이 문제는 $\$e\$$ 에서 나가는(outgoing) 일부 저장 노드가 오프라인일 경우 더욱 심각해진다. 이 경우, 새로 참여하는(incoming) 저장 노드들은 에폭 $\$e\$$ 의 위원회로부터 슬라이버를 복구해야 한다. 다행히 Red Stuff를 사용하면, 오류 발생 시 대역폭 비용이 오류가 없는 경우와 동일하게 유지된다. 하지만, 그럼에도 더 많은 메시지가 네트워크로 전송되어야 하고, 더 많은 증명 검증 및 심볼 → 슬라이버 디코딩 계산이 요구된다.

쓰기 경로를 차단하지 않고 이 문제를 해결하기 위해, 우리는 **읽기와 쓰기를 동시에 넘기는 방식 대신**, 재구성이 시작되자마자 **쓰기 요청은 $\$e + 1\$$ 의 위원회로 직접 보내고, 읽기 요청은 기존 위원회에 유지하는 접근**을 선택하였다. 이는 새롭게 기록된 블록을 읽을 때 어떤 노드가 데이터를 보유하고 있는지 명확하지 않아 읽기 혼선을 초래할 수 있다.

이를 해결하기 위해, 우리는 각 *블록의 메타데이터에 해당 블록이 처음 작성된 에폭 정보를 포함시킨다*. 만약 해당 블록이 에폭 $e + 1$ 에 처음 작성되었다면, 클라이언트는 읽기 요청을 새로운 위원회에 보내도록 한다. 그렇지 않다면, 기존 위원회로 요청을 보낼 수 있다. 이 방식은 **오직 핸드오버 기간**(두 위원회가 모두 활성화되어야 하는 기간)에만 적용된다.

새 위원회의 구성원 중 한 명이 자신의 샤드 상태를 완전히 동기화(즉, 모든 슬리버를 수신)하면, 해당 노드는 준비가 완료되었음을 신호로 보낸다. $2f + 1$ 명의 새 위원회 구성원이 준비 완료를 신호하면, 재구성 프로세스는 종료되고, 모든 읽기 요청은 새 위원회의 저장 노드로 리디렉션된다.

보안 측면 요약: 재구성은 에폭 간 ACDS의 모든 속성을 보장한다. 핵심 불변 조건은 다음과 같다: 블록이 여러 에폭에 걸쳐 가용하기 위해서는 **각 에폭마다 서로 다를 수 있는 $f + 1$ 개의 정직한 저장 노드가 슬리버를 보유하고 있어야 한다**. 이것이 $2f + 1$ 개의 노드에 의한 명시적 신호를 통해 에폭 전환을 연락하는 이유다. 이로 인해, 모든 다른 정직한 저장 노드는 결국 슬리버 쌍을 복구할 수 있고, 다음 에폭에서도 $f + 1$ 개의 정직한 노드가 슬리버 쌍을 복구할 수 있게 된다.

4.6 스토리지 챌린지 (Storage Challenges)

Walrus는 데이터를 저장하거나 제공하지 않으면서 보상을 받으려는 부정행위를 방지하고, 정직한 노드에 인센티브를 제공하기 위해 **챌린지 프로토콜(challenge protocol)**을 사용한다. 우리가 아는 한, 본 문서에서 제안하는 방식은 **네트워크의 동기성(synchrony)에 대한 어떠한 가정도 하지 않는 최초의 저장 증명 프로토콜**이다. 이 프로토콜은 *Red Stuff*의 완전성(completeness) 속성과 $2f + 1$ 임계값으로 블록을 복원할 수 있는 능력을 기반으로 한다. 본 절에서는 먼저 이론적으로 안전하지만 비용이 큰 기본 프로토콜을 설명하고, 이후 보안성을 확률적으로 완화하되 챌린지 비용을 현저히 줄일 수 있으며, 읽기 실패가 발생할 경우 동적으로 조정할 수 있는 방식을 논의한다.

완전 비동기 챌린지 프로토콜 (Fully Asynchronous Challenge Protocol)

에폭이 끝나기 직전, 저장 노드들은 온체인 상의 특정 블록 높이 등으로 표시되는 “챌린지 시작” 이벤트를 감지한다. 이 시점에서 노드들은 읽기 및 복구 요청을 중단하고, 챌린지 모드에 진입했음을 브로드캐스트한다. $2f + 1$ 개의 정직한 노드가 챌린지 단계에 진입하면 챌린지가 시작된다.

챌린지를 받은 각 노드는 블록당 공통 심볼(common symbols)과 작성자의 커밋에 대한 증명을 함께 다른 노드들에게 전송한다. 수신 노드는 심볼을 검증하고 확인 서명(confirmation signature)을 반환한다. 증명 중인 저장 노드가 $2f + 1$ 개의 서명을 수집하면, 이를 바탕으로 인증서(certificate)를 생성하고 온체인에 제출한다. $2f + 1$ 개의 유효한 인증서가 제출되면 챌린지 기간이 종료되고, 읽기 및 복구 서비스가 재개된다.

챌린지 기간 동안, 챌린지 시작 메시지를 감지한 노드는 읽기나 복구 요청에 응답하지 않는다. 챌린지 시작 임계값이 $2f + 1$ 이므로, 챌린지 파일이 정해진 이후에도 최소 $f + 1$ 개의 정직한 노드는 응답하지 않게 된다. 따라서, 공격자가 f 개의 슬리버만 저장하고 f 개의 정직한 노드를 지연시켜 챌린지 시작 메시지를 받지 못하게 하더라도, 공격자는 2차 슬리버에서 최대 $2f$ 개의 심볼과 $2f$ 개의 서명밖에 확보할 수 없다. 이는 전체 2차 슬리버를 복원하거나, 나머지 정직한 노드에게 인증서를 받아내기에는 부족하며, 결국 챌린지에 실패하게 된다. 이 증명의 구체적인 내용은 5.4절에 설명되어 있다.

완화 방안 (Relaxations)

이 프로토콜은 보안성이 뛰어나지만, 챌린지 기간 동안 읽기를 중단해야 하며, 전체 챌린지를 수행하려면 저장된 데이터 전체 크기만큼의 대역폭이 필요하다는 단점이 있다. 이러한 영향을 줄이기 위해, 우리는 **모든 블록에 대해 챌린지를 수행하지 않고 대부분의 블록은 읽기 가능하도록 두면서 보안성을 완화**하는 접근을 고려한다.

경량 챌린지 프로토콜(lightweight challenge protocol)을 위해, 우리는 저장 노드들이 $2f + 1$ 임계값을 갖는 **랜덤 코인(random coin)**을 설정할 것을 요구한다. 이는 어떤 종류의 비동기 DKG(Distributed Key Generation) [13, 14, 20] 또는 난수 생성 프로토콜 [17, 40]을 통해 가능하다.

이 코인은 **의사 난수 함수(PRF)**의 시드를 제공하며, 이를 통해 각 저장 노드가 챌린지해야 할 블록의 목록을 정의한다. PRF의 결과로 선택되지 않은 블록은 챌린지 없이도 자유롭게 접근할 수 있다. 챌린지 대상 블록의 수는 전체 블록 수에 비해 충분히 커야 하며, 그렇지 않으면 저장 노드가 챌린지 대상 블록을 모두 저장하고 있을 확률이 무시할 수 있을 정도로 낮아야 한다. 예를 들어, 한 노드가 전체 블록의 90%(또는 99%)를 보유하고더라도, 640개(또는 7000개)의 파일 챌린지에서 성공할 확률은 10^{-30} 이하로 떨어진다.

만약 챌린지가 통과되는데도 읽기 요청이 실패하는 경우, 이는 챌린지 대상 블록 수가 충분하지 않다는 의미이며, 이때 *Walrus*는 챌린지 범위를 늘려 결국 전체 챌린지 프로토콜을 다시 활성화할 수 있다. 그러나 이러한 상황은 공격자 노드가 저장 공간을 거의 절약하지 못한 경우(즉, 일정 상수 이하의 저장만 생략한 경우)에만 발생하므로, 실제로는 자원 비용에 큰 영향을 미치지 않을 가능성이 높다.

5 RED STUFF 및 WALRUS에 대한 증명 (Proofs)

이 절에서는 3장에서 다룬 내용을 보완하며, *Red Stuff*가 ACDS의 모든 속성을 만족함을 보여준다. 일반 독자는 이 절을 건너뛰어도 무방하다.

5.1 쓰기 완전성 (Write Completeness)

우리는 *Red Stuff*가 쓰기 완전성(Write Completeness)을 만족함을 증명한다. 비공식적으로, 정직한 작성자가 블록 B 를 네트워크에 기록하면, 모든 정직한 저장 노드는 결국 B 의 올바르게 인코딩된 1차 및 2차 슬라이버를 보유하게 된다. 이 절에서는 작성자가 정직하며, 올바른 벡터 커밋먼트 M 를 제공한다고 가정한다.

보조정리 1 (Lemma 1) (1차 슬라이버 복원).

*어떤 참가자가 1차 슬라이버 $S^{\{p,i\}}$ 로부터 $(2f + 1)$ 개의 심볼 $\{E(i,)\}_{2f+1}$ 을 보유하고 있다면, **이로부터 전체 1차 슬라이버 $S^{\{p,i\}}$ 를 복원할 수 있다.**

증명. 이는 복원 임계값이 $(2f + 1)$ 인 삭제 부호의 복원 성질로부터 직접적으로 따라온다.

보조정리 2 (Lemma 2) (2차 슬라이버 복원).

어떤 참가자가 2차 슬라이버 $S^{\{s,i\}}$ 로부터 $(f + 1)$ 개의 심볼 $\{E(i,)\}_{f+1}$ 을 보유하고 있다면, **이로부터 전체 2차 슬라이버 $S^{\{s,i\}}$ 를 복원할 수 있다.***

증명. 이는 복원 임계값이 $(f + 1)$ 인 삭제 부호의 복원 성질로부터 직접적으로 따라온다.

정리 1 (Theorem 1). *Red Stuff*는 쓰기 완전성(정의 1)을 만족한다.

증명. 정직한 작성자 W 가 블록 B 를 작성할 때, W 는 적어도 $(2f + 1)$ 개의 올바르게 인코딩된 슬리버와, 이들에 대한 벡터 커밋먼트 M 를 다양한 저장 노드에게 전송한다. 이 노드들에 대해서는 정의상 성질이 성립한다.

이제, 초기 $(2f + 1)$ 개의 수신자가 아닌 노드 j 를 고려하자. 이 노드는 모든 노드 i 에게 자신의 1차 슬리버(즉, $E(i, j)$) 및 2차 슬리버(즉, $E(i, j)$)와 관련된 공유 심볼을 요청할 수 있다. 커밋먼트 M 이 바인딩되어 있으므로, 노드 i 는 올바른 심볼을 보내거나 아무 응답도 하지 않을 것이다. M 을 수락한 노드가 최소 $2f + 1$ 개이므로, 노드 j 는 자신의 1차 슬리버 $E(j, j)$ 및 2차 슬리버 $E(j, j)$ 에 대해 적어도 $f + 1$ 개의 올바른 심볼을 얻게 된다. 보조정리 2에 의해, j 는 자신의 전체 2차 슬리버 $S^{\{s, j\}}$ 를 복원할 수 있다.

이러한 논리는 임의의 노드 i 에 대해 성립하므로, 모든 노드에 대해 동일하게 적용된다. 결과적으로, 모든 $2f + 1$ 개의 정직한 노드는 결국 자신의 2차 슬리버 $S^{\{s, *\}}$ 를 복원하게 된다. 각 노드가 자신의 2차 슬리버를 복원할 때마다, 해당 노드는 노드 j 에게 j 의 1차 슬리버 일부인 공유 심볼 $E(j, *)$ 을 응답으로 보낸다. 결과적으로, j 는 $E(j, *)$ 에서 $E(j, *)$ 으로 확대되며, 이는 보조정리 1을 적용하여 j 의 1차 슬리버 $S^{\{p, j\}}$ 를 복원할 수 있게 한다.

이 논리는 모든 일반적인 노드 i 에 대해 동일하게 적용되므로, 모든 정직한 노드는 결국 자신의 1차 및 2차 슬리버를 보유하게 되며, 이에 따라 정리가 증명된다.

5.2 읽기 일관성 (Read Consistency)

우리는 *Red Stuff*가 읽기 일관성(Read Consistency)을 만족함을 증명한다. 비공식적으로 말하면, 두 명의 정직한 독자가 네트워크에 기록된 동일한 블록 B 를 읽는 경우, 둘 다 결국 B 를 얻거나, 둘 다 실패하여 $\perp$ 를 얻는다.

정리 2 (Theorem 2). *Red Stuff*는 읽기 일관성(정의 1)을 만족한다.

증명. 인코딩 방식은 결정적(deterministic)이므로, 읽기의 마지막 단계는 인코딩을 다시 실행하고 커밋먼트 M 을 재구성하는 것이다. 따라서, 읽기를 성공적으로 수락한 독자는 반드시 B 를 출력해야 한다.

읽기 일관성에 대한 도전은 작성자가 서로 다른 슬리버 집합을 수집한 서로 다른 독자에게 각각 B 와 $\perp$ 를 출력하게끔 만들 수 있는지 여부에 있다. 두 정직한 독자 R_1 과 R_2 가 네트워크에서 블록 B 를 읽었고, R_1 은 결국 B 를 얻었으며, R_2 는 실패하여 $\perp$ 를 얻었다고 가정하자.

R_2 가 $\perp$ 를 출력할 수 있는 두 가지 경우는 다음과 같다:

1. R_2 는 커밋 M 과 일치하는 $2f + 1$ 개의 응답을 받고 복원을 시도한다. 복원 도중, 복원된 커밋이 M 과 일치하지 않는다.
2. 일부 노드가 자신의 2차 슬리버를 복원하지 못했다. 알고리즘에 따라 이 노드는 불일치 증명을 보유하

고 있으며, 이를 $\$R_2\$$ 에게 전송한다.

이 두 경우 중 어느 경우라도 $\$R_1\$$ 이 복원을 시도할 때에도 동일한 불일치를 감지하고 $\$perp\$$ 를 출력했어야 한다. 그렇지 않다면 벡터 커밋의 바인딩 속성이 성립하지 않게 되므로 모순이다.

5.3 정당성 (Validity)

우리는 *Red Stuff*가 정당성(Validity)을 만족함을 증명한다. 비공식적으로 설명하면, 정직한 작성자가 올바르게 인코딩된 블록 $\$B\$$ 를 네트워크에 기록하면, 모든 정직한 독자는 결국 $\$B\$$ 를 얻는다.

정리 3 (Theorem 3). *Red Stuff*는 정당성 (정의 1)을 만족한다.

증명. 작성자 $\$W\$$ 가 블록 $\$B\$$ 를 작성할 때, $\$W\$$ 는 $\$n\$$ 개의 올바르게 인코딩된 슬라이버들과 이들에 대한 바인딩 벡터 커밋 $\$M\$$ 을 생성한다. 정리 1에 따라 작성자가 정직하다면, 적어도 $\$2f + 1\$$ 개의 정직한 저장 노드는 자신의 해당 슬라이버를 보유하게 된다. 저장 노드 전체 집합을 $nodes$ 라 하자. 정직한 독자는 각 저장 노드 $\$n \in \text{nodes}\$$ 에게 2차 슬라이버를 요청하고, 이들을 커밋 $\$M\$$ 과 비교하여 검증하며, $\$2f + 1\$$ 개의 유효한 응답을 확보하면 이를 이용해 $\$B\$$ 를 복원한다. 모든 정직한 저장 노드는 결국 독자에게 응답하게 되며, 작성자 $\$W\$$ 또한 정직했기 때문에 독자는 결국 $\$B\$$ 를 얻는다.

5.4 비동기 챌린지 (Asynchronous Challenges)

우리는 블록의 모든 슬라이버를 보유하지 않은 악의적인 저장 노드는 챌린지에 성공적으로 응답할 수 없음을 증명한다. 공격자의 저장 노드가 공통 심볼의 단일 버전을 보유하며 서로 공모할 수는 있지만, 각 슬라이버마다 $\$2f + 1\$$ 개의 심볼을 보유해야 하므로, 이는 사실상 전체 슬라이버를 저장하는 것과 동일한 저장 비용이 요구된다.

정리 4 (Theorem 4). (*Secure Challenge Protocol*)

블록의 모든 슬라이버를 보유하지 않은 WALRUS 상의 악의적 저장 노드는 챌린지에 성공적으로 응답할 수 없다.

증명. 저장 노드 중 하나가 자신이 보유해야 할 심볼을 삭제하였고, 해당 심볼은 정직한 노드 중 한 곳에 존재한다고 가정하자. 챌린지 시점에서 이 노드는 해당 심볼을 정직한 노드에게 전송해야 인증서를 획득할 수 있다. 이를 위해 이 노드는 자신의 슬라이버를 복구해야 한다.

복구를 위해, 이 노드는 다른 악의적 저장 노드로부터 $\$f - 1\$$ 개의 심볼을 얻을 수 있다. 그리고 느리게 응답하는 정직한 노드로부터 $\$f\$$ 개의 심볼을 읽기 경로를 통해 얻을 수 있다. 그러나 이 경우 총 $\$f + f - 1 = 2f - 1\$$ 개의 심볼에 불과하다. 보조정리 1(Lemma 1)에 따르면, 1차 슬라이버를 복원하려면 $\$2f + 1\$$ 개의 심볼이 필요하다. 따라서 이 노드는 해당 심볼을 복구할 수 없고, 응답에 실패하게 된다.

응답에 실패한 경우, 인증서를 얻을 수 없다. 왜냐하면 $\$2f - 1\$$ 개의 다른 악의적 노드와 자신의 서명을 합쳐도 총 $\$2f\$$ 개로, $\$2f + 1\$$ 개의 서명 요건을 만족하지 못하기 때문이다.

6 경제 및 인센티브 (Economics and Incentives)

단순한 읽기 및 쓰기 기능을 넘어, *Walrus*는 경쟁력 있는 가격 책정, 효율적인 자원 할당, 최소화된 악의적 행위 유도를 위한 강건한 경제 및 인센티브 메커니즘을 설계한다. 개념적으로, *Walrus*의 경제적 과제는 일반적인 블록체인과는 다르다. 왜냐하면 *Walrus*는 블록체인을 제어 평면(control plane)으로 활용하므로 블록체인 합의의 보안성을 자연스럽게 상속받기 때문이다.

대신, *Walrus*는 장기 계약의 집행과 관련된 과제에 직면한다. *Walrus*의 노드와 사용자는 미리 일정 기간 동안 데이터를 저장하고 이후 이를 제공할겠다는 계약을 체결한다. 하지만 노드는 시간이 지남에 따라 계약을 위반하고, 예컨대 다른 사용자에게 더 높은 가격에 저장 공간을 재판매하거나, 데이터를 인질 삼아 추가 비용을 요구할 유인이 있다. 더 나아가 *Walrus*는 탈중앙화된 시스템이기 때문에, 노드는 시간이 지나며 단순히 시스템을 이탈하고, 향후 위원회에게 책임을 떠넘기는 일이 발생할 수 있다. 이는 잘 알려진 **공유지의 비극 (tragedy of the commons)** 문제와 유사하며, 개인적 이익을 추구하면서 장기 평판 비용을 내부화하지 않는 상황에서 나타난다. 전통 시장에서는 법률 시스템을 통해 장기 계약이 집행되지만, *Walrus*는 이러한 점을 인센티브와 디센티브에 의존해 해결해야 한다.

계약의 집행을 보장하는 주요 수단은 **스테이킹된 자본(staked capital)**이며, 이는 신규 토큰인 **WAL 토큰**을 중심으로 구성된다. 정직한 행동에 대해서는 보상을, 악의적 행동에 대해서는 슬래시(slash)를 통해 벌칙을 부과한다. 이 자본은 노드가 직접 제공할 수도 있고, 잘 운영되는 노드에 사용자가 위임하여 제공할 수도 있다.

동시에, *Walrus*의 경제 모델은 전체 노드 중 최대 $f = \lfloor \frac{n}{3} \rfloor$ 개 (정확하게는 전체 샤드의 $\lfloor \frac{n}{3} \rfloor$ 까지를 제어하는 노드)가 비잔틴일 수 있음을 가정하고, 악의적 행위를 포함한 임의의 장애에 대한 복원력을 갖추도록 설계되어 있다. 마지막으로, 이러한 핵심 메커니즘은 가격 책정, 접근 제어, 샤드 마이그레이션, 무결성 검사를 조율하는 다양한 보조 메커니즘과 함께 작동한다.

이 절의 구성은 다음과 같다.

6.1절에서는 시스템 전체의 보안성과 효율성을 뒷받침하는 **스테이킹 메커니즘**을 설명한다.

6.2절에서는 새로운 노드가 진입하거나 기존 노드가 이탈/지분 감소할 때 발생하는 **샤드 마이그레이션**을 다룬다.

6.3절에서는 노드가 **공정하고 경쟁력 있게 가격을 책정**하는 방법을 설명한다.

마지막으로 6.4절에서는 **토큰 거버넌스 프로세스**를 간략히 설명한다.

6.1 스테이킹 (Staking)

스토리지 노드가 직접 또는 대리인을 통해 **WAL 토큰을 스테이킹**함으로써 *Walrus*의 보안성이 뒷받침된다. 약속된 저장 의무를 이행한 노드는 보상을 받고, 그렇지 않은 노드는 슬래시(slash)를 당한다. *Walrus*의 스테이킹 시스템은 다음의 네 가지 핵심 구성 요소를 포함한다: (1) 스테이크 및 샤드 할당, (2) 언스테이킹 (unstaking) 절차, (3) 보상 및 벌칙의 발생, (4) 자체 보관 객체(self-custodial objects)에 필요한 조정.

스테이크 및 샤드 할당 (Stake and Shard Assignment)

Walrus는 위임 스테이킹(delegated staking) 계층을 내장하고 있어, 사용자가 직접 스토리지 서비스를 운영하지 않더라도 네트워크의 보안에 참여할 수 있도록 한다. 노드들은 사용자로부터 스테이크를 유치하기 위해 경쟁하며, 이는 노드에게 샤드를 할당하는 기준이 된다.

사용자는 노드의 평판, 노드 자체의 스테이킹 금액, 설정된 수수료율, 기타 요소들을 기준으로 위임할 대상을 선택한다. 에폭이 변경되기로 확정되는 시점(도표 Figure 5의 포인트 **c**)에 도달하면, 스테이크는 해당 노드에 고정되며, 이후 언스테이킹을 원할 경우 별도로 정의된 언스테이킹 절차를 따라야 한다. 샤드는 노드가 보유한 스테이크의 비율에 따라 다음 에폭에 할당되며, 이는 6.2절에서 더 자세히 설명된다.

Walrus는 노드가 일정 수준 이상의 자본을 반드시 직접 예치해야 한다는 요구사항을 두지 않는다. 즉, 노드는 자신의 자본을 일부, 전부 또는 전혀 사용하지 않는 방식 중 자유롭게 선택할 수 있다. 이는 의도적인 설계다. 위임자는 위임 여부를 결정할 때 노드가 보유한 자체 자본 수준을 고려할 가능성이 있지만, Walrus는 이를 의무화하지 않는다.

다만, 수수료율 설정과 관련해서는 하나의 안전장치를 둔다. **수수료율은 해당 커트오프 시점보다 최소 한 개의 전체 에폭 이전에 설정되어야 한다.** 따라서 어떤 노드가 과도하게 수수료를 인상하더라도, 위임자는 새 수수료가 적용되기 전에 충분히 언스테이킹할 수 있는 시간을 갖게 된다.

언스테이킹 (Unstaking)

언스테이킹 절차는 스테이킹과 유사하게, 특정 에폭의 커트오프 포인트(도표 Figure 5의 포인트 **c**) 이전에 지갑이 언스테이킹 요청을 등록해야 한다. 커트오프에 도달하면 해당 스테이크는 샤드 할당 계산에서 제외되지만, **즉시 반환되지는 않는다.**

대신, 해당 스테이크는 6.2절에서 설명할 **샤드 마이그레이션 과정이 완료된 이후에 반환**된다. 만약 이 과정에서 해당 노드가 의무를 이행하지 못하면 슬래시될 수 있다. 반대로, 샤드 마이그레이션에 연루되지 않은 노드의 경우, 언스테이킹은 에폭이 끝날 때 자동으로 완료된다. Walrus는 이러한 절차 외에 추가적인 요구사항은 두지 않는다.⁸

보상 및 벌칙 (Rewards and Penalties)

각 에폭이 종료되면, 노드는 해당 에폭 동안의 활동에 따라 보상 또는 벌칙을 받는다. 챌린지(4.6절) 응답을 통해 저장 증명에 성공하거나, 시스템 내 쓰기를 촉진하거나, 샤드 복구에 참여한 노드는 보상을 받는다. 반대로 챌린지에 올바르게 응답하지 못한 노드는 벌칙을 받는다. 이러한 보상 및 벌칙의 크기는 프로토콜 수익과 토큰 거버넌스에 의해 조정된 매개변수의 조합에 따라 결정되며, 이에 대해서는 후속 절에서 다룬다.

또한, **샤드 마이그레이션 과정에서 실패한 노드에 대한 벌칙은 해당 마이그레이션이 종료되는 시점에 미리 부과된다.** 이는 언스테이킹하려는 자본에 슬래시가 적용될 수 있도록 하여, 부정행위를 막는다.

사용자는 자신의 스테이크가 활성화되어 있는 전체 에폭 기간 동안 보상과 벌칙을 **비례적으로 공유**한다. 저장 노드가 직접 자본을 스테이킹할 경우, 동일한 방식으로 보상 및 벌칙을 받으며, 추가로 **위임자에게 발생한 보상에서 수수료를 별도로 수취**할 수 있다.

Walrus는 초기에는 모든 스테이크를 동등하게 취급하여, 주니어/시니어 스테이크 구분 없이 동일한 보상률과 벌칙률을 적용한다. 다만, 이는 프로토콜 차원에서 커뮤니티의 판단에 따라 향후 변경될 수 있으며, 개별 노드가 Walrus 위에 자체적인 기능으로 구현하는 것도 가능하다.

스테이킹에서의 자산 자가 보관 (Self-Custody in Staking)

Sui [7]와 마찬가지로, Walrus는 스테이킹을 **자가 보관 객체(self-custodied objects)** 기반으로 구현한다. 사용자가 자금을 스테이킹하면, 해당 자금은 사용자가 직접 보유하는 객체 형태로 감싸져 저장된다. 이는 Walrus 시스템의 보안 취약 면적을 줄여주며, 사용자가 이 객체를 기반으로 별도의 기능을 구현할 수 있도록 한다.

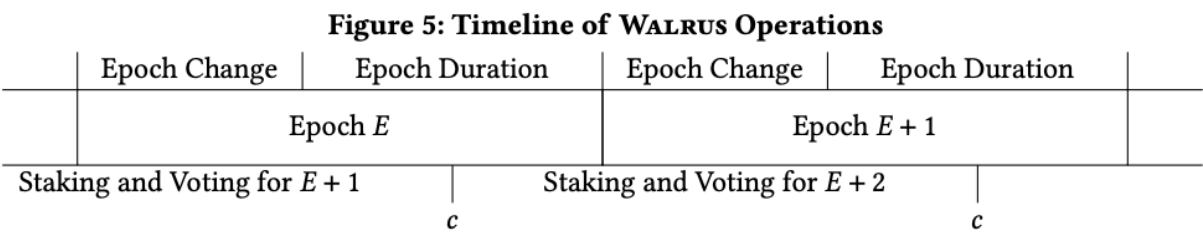
그러나 이 방식은 운영상 문제를 유발한다. Walrus는 스테이킹 원금을 슬래시할 수 있지만, 이 설계 방식에서는 Walrus가 실제로 해당 원금을 보관하지 않기 때문이다.

이를 해결하기 위해 Walrus는 **모든 미납 벌칙금(outstanding penalties)**을 추적한다. 사용자가 WAL 토큰을 회수하려면, 객체를 Walrus 스마트 계약에 제출해 이를 해제(unwrapping)해야 한다. 이 해제 시점에서, 해당 스테이크에 누적된 모든 벌칙이 정산된다.

이 구조는 두 가지 문제를 유발할 수 있다. 첫째, Walrus는 **일시적인 현금 흐름 부족(cash flow problem)** 상황에 직면할 수 있다. 특히 벌칙금이 손해를 입은 다른 참가자에게 보상으로 전달되는 경우 더욱 그러하다.

이를 해결하기 위한 Walrus의 두 가지 보완책은 다음과 같다:

- 1. **보상 차감 방식:** 스테이크에 대한 보상이 발생할 때, Walrus는 미수금으로 남은 보상을 회수하거나, 향후 보상 분배를 차감하여 벌칙금을 충당한다.
- 2. **이자 부과 방식:** 미납 벌칙금은 시간이 지날수록 이자가 붙으며 누적된다. 사용자가 해당 금액을 직접 Walrus에 송금하거나, 언스테이킹 및 해제 시점에 해당 금액을 정산할 수 있다.



두 번째 문제는, **슬래시로 인해 가치가 0 또는 0에 가까워진 객체가 사용자의 손에 그대로 남아 반환되지 않는 상황**이다. 이 경우 Walrus는 해당 토큰을 회수할 수 없다.

이를 방지하기 위해 Walrus는 스테이킹 객체를 해제할 때, 초기 원금의 일정 비율(예: 5%)은 무조건 지급되도록 한다. 이 소액 인센티브는 사용자가 결국 모든 스테이킹 객체를 Walrus에 반환하여 해제하게 만들며, Walrus가 슬래시된 토큰을 최종적으로 회수할 수 있도록 해준다.

6.2 샤드 마이그레이션 (Shard Migration)

샤드는 노드 간 상대적인 스테이크의 증감에 따라 이동한다. 이 마이그레이션은 Walrus의 보안성에 매우 중요하다. 만약 스테이크의 일부만 가진 소수 노드가 비잔틴 행동을 하더라도 샤드 마이그레이션이 없으면 시스템 전체가 정지할 수 있기 때문이다. 샤드 마이그레이션은 다음 세 가지 구성 요소로 이루어진다: (1) 할당 알고리즘, (2) 협력 기반 마이그레이션 경로, (3) 복구 기반 마이그레이션 경로.

할당 알고리즘 (Assignment Algorithm)

에폭 종료 전 커트오프 포인트(즉, Figure 5의 포인트 **c**)에 도달하면 샤드 할당 알고리즘이 작동한다. 이 시점에서 스테이킹 요청은 확정되어 해당 스테이크가 포함되고, 언스테이킹 요청 또한 확정되어 해당 스테이크는 제외된다(단, 실제로 반환되지는 않음). 샤드는 노드 간 **상대적인 스테이크**를 기준으로 할당된다. 단, 샤드 이전은 다음 에폭이 시작되어야 실제로 진행된다.

할당 알고리즘은 **노드와 샤드 간의 안정성을 최대한 유지하며, 이동량을 최소화**하는 방식으로 설계되어 있다. 구체적으로, 샤드를 새롭게 받는 노드는 기존 샤드를 유지하면서 다른 노드가 잃은 샤드를 추가로 받는다. 반대로, 샤드를 잃는 경우에도 어느 정도 허용 오차가 적용된다. 예를 들어, 아주 약간의 스테이크 (ϵ) 증가로 샤드를 하나 추가로 받게 된 노드는 이후 해당 ϵ 만큼을 잃었다고 해서 즉시 그 샤드를 잃지는 않는다.

향후에는 샤드 할당 알고리즘을 더 유연하게 개선할 수 있다. 예를 들어, 노드가 선호하는 샤드를 지정하고, 할당 알고리즘이 이를 반영하거나, 노드 간 샤드 마이그레이션 의무를 **상호 교환할 수 있는 기능**을 허용할 수 있다. 예컨대, 동일한 데이터 센터에 위치한 노드들은 **내부 전송 비용이 없기 때문에 서로 샤드를 교환**하는 것을 선호할 수 있다. 또는 두 노드가 각각 하나의 샤드를 두 명의 다른 노드에게 이전해야 하는 경우, 네 개의 노드가 상호 동의하여 마이그레이션 경로를 교환함으로써 효율을 높일 수 있다(이 경우 제3자가 전체 전송 비용을 최적화하고, 교환에 필요한 보상을 중개할 수도 있음).

샤드 할당은 저장 노드의 **저장 용량과 무관하게 오직 스테이크만으로 이루어진다**.

즉, 특정 노드가 갑작스러운 스테이크 유입으로 인해 **자신의 저장 한계를 초과한 샤드를 배정받을 수도 있다**.⁹

이러한 이유로 할당 알고리즘은 에폭 종료 전에 실행되어, 해당 노드가 **충분한 저장 공간을 미리 확보할 수 있는 시간**을 제공한다. 또한, 노드는 새로운 스테이킹 요청을 거절함으로써 단기적인 저장소 확장의 리스크를 스스로 조절할 수 있다.

협력 경로(Cooperative Pathway)

샤드 마이그레이션 프로세스는 에폭이 종료된 후 시작된다. 이는 노드들이 샤드를 이전하도록 기대되는 일정 기간으로 시작된다. 만약 이 작업이 협력적으로, 즉 송신 노드와 수신 노드가 조율하여 수행된다면, 토큰의 이동은 발생하지 않으며 추가적인 조치는 없다.

구체적으로, 각 양자 관계(송신자, 수신자)에 대해, 수신자는 전송 기간 동안 샤드를 보유하고 있음을 증명한다. 이 과정이 완료되면 전송은 완료된 것으로 간주되며, 이후 샤드에 대한 이의 제기에 대한 책임은 수신자에게 있다. 하나의 노드가 관련된 각 양자 전송이 모두 종료되면, 해당 노드의 언스테이킹 요청(있을 경우)이 처리된다.

이 메커니즘은 시스템으로부터의 자발적인 노드 탈퇴를 내포하고 있다. 해당 노드는 자신의 전체 스테이크를 출금 대상으로 표시하고, 샤드를 협력적으로 외부로 마이그레이션한 후, 수신자가 이를 증명하면 모든 토큰을 회수한다.

복구 경로(Recovery Pathway)

Walrus는 개방형 시스템이기 때문에 샤드 마이그레이션 실패에 견딜 수 있는 강인함이 요구된다. 전송 기간이 종료되었을 때, 수신 노드가 모든 샤드를 수신하지 않았음을 증명하거나, 아예 증명을 하지 않으면, 전송은 실패한 것으로 간주되며 복구 경로가 초기화된다.

복구 경로 하에서는 송신 노드의 스테이크가 실질적으로 슬래시되며, 슬래시되는 페널티 양은 거버넌스에 의해 설정된다. 다른 모든 노드는 샤드를 복구하는 데 참여하며 (4.3절), 슬래시된 자금은 그들의 비용을 보전하기 위해 분배된다. 중요한 점은, 수신 노드의 스테이크 또한 슬래시된다는 것이다. 이는 상대적으로 소량이며, 마찬가지로 거버넌스에 의해 설정된다. 이는 협력적 마이그레이션에 대해 허위 보고를 하지 않도록 올바른 인센티브를 설정하기 위한 조치이다. 복구가 완료되면, 수신 노드는 해당 샤드에 대한 이의 제기의 책임을 진다.¹⁰

비마이그레이션 복구(Non-Migration Recovery)

Walrus의 복구 경로는 주로 샤드 마이그레이션을 위한 것이지만, 마이그레이션 외의 복구 상황(예: 하드 드라이브의 물리적 손상)에도 사용할 수 있다. 이 흐름에서는 샤드를 유실한 노드가 자발적으로 온체인 요청을 제출하며, 이는 샤드 마이그레이션 복구 시 송신 노드가 부담하는 페널티와 동일한 금액만큼 슬래시된다. 노드가 이 과정을 자발적으로 선택할 수 있는 이유는, 대안으로는 해당 노드가 4.6절의 데이터 가용성 챌린지를 반복적으로 실패한 후 결국 샤드를 마이그레이션해야 하며, 이는 더 많은 비용이 들기 때문이다. 유실된 샤드의 복구에 참여한 노드들은 이 슬래시된 자금을 획득하여 자신의 비용을 상쇄한다.

추가로, 노드들은 서로에게서 blob을 임의로 복구(ad-hoc recovery)할 수 있다. 이러한 복구는 최선의 노력(best-effort) 기준으로 수행되며, 노드들은 서로에 대해 자유롭게 속도 제한(rate limit)을 적용할 수 있다. 이 경로는 특히 최근 에포크에서 누락된 개별 blob을 처리하는 데 적합하지만, 전체 샤드의 손실을 복구하는 데에는 확장되지 않을 수 있다.

6.3 저장 및 쓰기 작업에 대한 지불(Payments for Storage and Writes)

Walrus의 분산된 특성을 고려할 때, 저장 자원과 쓰기 작업은 경쟁적이며 협력적인 방식으로 가격이 책정되어야 한다. 스토리지 노드들은 낮은 가격에 충분한 저장 공간을 제공하기 위해 서로 경쟁해야 하지만, 동시에 이들의 제출 데이터를 저장 소비자에게 통합된 스케줄로 표현하는 공동의 체계가 필요하다. 이 절에서는 저장 자원의 개요, 노드가 저장 자원의 양을 결정하는 방식, 저장 및 쓰기 가격을 결정하는 방식, 노드에 대한 지불 방식 등을 다룬다.

저장 자원(Storage Resources)

스토리지는 *Walrus*에서 저장 자원(storage resources)이라는 형태로 매매되며, 이는 Sui 블록체인 상에 표현된다. 이러한 자원은 *Walrus* 내 저장 공간에 대한 예약 권한 역할을 하며, 시작 에포크, 종료 에포크, 크기를 포함한다. 사용자는 데이터를 저장할 준비가 되었을 때 특정 blob을 보관하기 위해 저장 자원을 등록하고, 데이터를 쓰고 가용성 시점(Point of Availability)을 확립한다.

blob에 부착되기 전의 저장 자원은 시간 또는 공간을 기준으로 분할할 수 있다. 저장 자원은 또한 거래될 수 있다. 마지막으로, 삭제된 blob과의 연결이 해제된 저장 자원은 새로운 blob에 재연결될 수 있다. 이는 *Walrus* 내에서 경제적 효율성을 가능하게 하는 강력한 저장 자원 2차 시장의 기반을 형성한다.

저장 용량(Storage Quantity)

노드는 먼저 시스템 전체의 샤드 크기에 대해 투표함으로써 판매 가능한 저장 용량을 결정한다. 고정된 샤드 수와 복제 계수를 고려하면, 이는 즉시 시스템 전체 저장 용량을 결정하며, 따라서 확장적으로 판매 가능한 미사용 저장 공간을 정의한다.

스테이킹과 마찬가지로, 저장 용량과 가격에 대한 투표는 에포크 시작 전에 미리 이루어진다. 스테이킹 마감 시점은 투표 마감 시점과 동일하며(즉, Figure 5의 지점 **c**), 투표하지 않은 스토리지 노드는 이전 투표가 현재 투표로 자동 적용된다. 이 시점에서 모든 샤드 크기 제출값은 내림차순으로 정렬되며, 스테이크 가중치 기준으로 66.67번째 백분위 수에 해당하는 제출값이 최종 샤드 크기로 채택된다. 이는 전체 제출값 중 2/3가 더 큰 샤드 크기를, 1/3이 더 작은 크기를 제안했음을 의미한다.

이 시점에서, 전체 저장 용량과 이미 약정된 저장 용량의 차이인 미사용 저장 공간이 계산된다. 미사용 저장 공간이 존재한다면, 해당 공간은 다음 에포크에 판매에 들어가며, 해당 에포크부터 시작된다. 사용자는 구매 시 종료 에포크(현재 에포크부터 2년 이내의 에포크까지)와 크기를 지정하여 저장 자원을 생성한다. 종료 에포크의 상한을 두는 이유는 초기 세대의 노드가 이후 여러 세대의 노드에 과도한 제약을 가하지 않도록 하기 위함이다. 만약 전체 저장 용량이 약정된 용량보다 작은 경우, 새로운 저장 공간은 판매되지 않지만 *Walrus* 노드는 기존 약속을 계속 이행해야 한다. 저장 자원을 삭제할 수 없으며, 삭제할 경우 챌린지를 실패하고 슬래시되므로 이는 허용되지 않는다(4.6절 참조).

일반적으로 저장 공간은 현재 에포크에서 시작하는 자원 형태로 판매되지만, *Walrus*는 기존 blob의 갱신을 효율적으로 처리할 수 있도록 하나의 대체 흐름을 허용한다. 기존 blob을 보유한 사용자는 현재 종료 에포크부터 향후 어느 에포크(최대 2년 이내)까지 blob의 수명을 연장할 수 있으며, 연장된 기간에 대해 공시된 가격을 지불한다. 갱신 참여자는 소규모의 묵시적 선택권(embedded option)을 가지게 되며, 이는 다음 에포크의 가격이 현재 에포크 중반에 설정되기 때문에 발생한다. 즉, 현재 저장 공간이 현재 에포크 가격으로 판매되는 동안 가격이 확정되므로, 갱신자는 일부 가격 우위를 가질 수 있다. 다만, 연속적인 에포크 간에 가격이 일반적으로 안정적인 경우, 이 효과는 미미하다. 이러한 모델은 사용자가 데이터의 장기적 또는 무한한 저장 수명을 확보할 수 있도록 하며, 커버리지가 끊기거나 중복 지불되는 상황 없이 이를 가능하게 한다.

저장 및 쓰기 가격(Pricing of Storage and Writes)

노드는 샤드 크기뿐 아니라 저장 공간과 쓰기 작업에 대한 가격도 설정한다. 마찬가지로, 전체 에포크를 앞서 Figure 5의 지점 **c** 이전에 노드는 저장 자원 가격(저장 단위당, 에포크당 기준)과 시스템 쓰기 가격(저장 단위당 기준)을 제출한다. 이 가격들은 각각 *오름차순*으로 정렬되며, 스테이크 가중치 기준으로 66.67번째 백분위 수에 해당하는 값이 선택된다. 즉, 전체 제출값 중 2/3는 더 낮은 가격을, 1/3은 더 높은 가격을 제시한 셈이다.¹¹ 저장 가격은 그대로 채택되며, 쓰기 가격은 하드코딩된 1보다 큰 계수를 곱해 적용된다.

이 하드코딩 계수는 환불 가능한 추가 보증금을 반영한다. *Walrus*는 가용성 지점(Point of Availability)이 온체인에 공개적으로 확인되면 공식적으로 해당 blob에 대한 책임을 진다. 하지만, 모든 노드가 사용자로부터 blob을 직접 수신하는 경우가 더 효율적이며, 이는 데이터를 받지 못한 노드의 복구에 필요한 오버헤드를 줄여주기 때문이다. 쓰기 작업에 대한 환불 가능한 보증금은 이러한 효율을 유도하기 위한 장치다. 사용자가 인증서에 더 많은 노드 서명을 받을수록, 더 많은 보증금이 환불된다. 따라서 사용자는 데이터를 단순히 $\$2f + 1\$$ 개의 노드에만 업로드하는 대신, 가능한 모든 노드에 업로드하도록 동기부여를 받는다.

노드에 대한 지불(Payments to Nodes)

*Walrus*의 출시 초기에는 지불 구조가 단순하다. 첫째, 사용자는 blob을 등록할 때 데이터 쓰기에 대한 현재 가격을 지불하며, 이 비용은 에포크 종료 시점에 노드에 분배된다. 둘째, 사용자는 저장 자원을 구매할 때 비용을 지불한다(이는 blob을 언제 등록하느냐와 무관하다). 이 경우 *Walrus* 스마트 계약은 해당 노드가 속한 샤드 단위 버킷으로 토큰을 분배하며, 역시 에포크 종료 시점에 노드가 이를 수령한다.

이러한 단순한 저장 자원 지불 모델은 두 가지 핵심 이점을 제공한다. 첫째, 사용자 가격이 고정되고 선불로 지불된다. 사용자는 계약 기간 동안 저장 가격이나 WAL 토큰 가격의 변동을 걱정할 필요가 없으며, 노드가 계약 중간에 사용자에게 착취적인 가격을 부과할 수도 없다. 둘째, 계약 기간이 고정되며 사용자는 종도 해지 시 지불금을 몰수당하기 때문에 임의로 계약에서 이탈할 수 없다. 만약 이러한 구조가 없다면, 노드가 향후 저장 가격을 낮추는 경우 기존 계약자들이 모두 해지하고 더 낮은 가격으로 갱신할 수 있어 위험이 된다. 따라서 양측 모두에 대한 장기적 계약 강제력은 관계의 안정성을 보장한다.

이 모델에는 두 가지 약점이 있으며, 따라서 *Walrus*는 향후 이를 개선할 가능성이 있다. 첫째, 자본 효율성이 떨어진다. 사용자는 서비스를 받기 훨씬 이전에 계약 전체 금액을 미리 충전해야 하기 때문이다. 둘째, 노드는 종종 현지 통화 기준의 비용 구조를 가지는데, WAL로 수익을 받기 때문에 WAL 가격의 변동성이 클 경우 효율적으로 경쟁하기 어렵다. 이에 대한 대안 모델은 사용자가 노드와 장기 계약을 체결하되 다음 두 가지 차이점을 가지는 방식일 수 있다. 첫 번째는 사용자가 전체 계약이 아닌 마지막 몇 개 에포크에 대해서만 선불로 지불함으로써 자본 효율성을 확보하면서도 조기 해지를 막을 수 있는 충분한 스테이크를 잠그는 것이다. 두 번째는 계약 가격이 오직 오라클 가격(예: WAL/USD)을 기반으로 프로그램적으로만 갱신되도록 함으로써 양측 모두에게 보다 안정적인 현지 통화 기반 계약을 제공하는 것이다.

6.4 토큰 거버넌스(Token Governance)

Walrus의 거버넌스는 시스템 내 매개변수를 조정하며, WAL 토큰을 통해 운영된다. 구체적으로, 노드들은 각각의 WAL 스테이크에 비례하여 다양한 페널티 수준을 결정하기 위해 투표를 진행한다. 이는 타 노드의 성능 저하로 인한 비용을 자주 부담하게 되는 Walrus 노드들이 적절한 재정적 제재를 설정할 수 있도록 해준다.

반면, Walrus의 거버넌스는 프로토콜을 직접 변경하지 않는다. 프로토콜의 변경은 $2f + 1$ 개의 스토리지 노드가 재구성 시 이를 수락해야 효력이 발생하며, 따라서 이는 스테이킹된 토큰에 의해 묵시적으로 승인되는 구조이다. 프로토콜 변경은 보통 보안, 경제성, 비즈니스 위험에 대한 충분한 논의를 거친 후에만 진행될 수 있다. 많은 L1 블록체인들도 이와 유사한 구조를 갖는다(예: Sui의 SIP, Sui Improvement Proposals).

거버넌스 흐름(Governance Flow)

Walrus의 토큰 거버넌스는 노드들이 가격을 설정하는 메커니즘과 유사하며, 파라미터의 제약 조건과 합의 도출 방식만 다르다. 구체적으로, 다음과 같은 흐름을 따른다:

1. 특정 에포크의 스테이킹 마감 시점(즉, Figure 5의 지점 **c**)까지, 어떤 Walrus 노드든 샤드 복구에 대한 비용과 데이터 챌린지 수행 실패에 따른 비용과 관련된 파라미터 집합을 다음 에포크를 대상으로 제안할 수 있다.
2. 하나 이상의 제안이 활성화되면, Walrus 노드들은 동일한 에포크 내에서 단일 제안(또는 현행 상태 유지안)에 대해 투표할 수 있으며, 이때 투표권은 위임 스테이크를 포함한 총 스테이크량에 비례한다.
3. 마감 시점 **c**에 도달하면, 전체 투표 수가 정족수를 만족하는 조건 하에 과반수(50% 초과)의 투표를 획득한 제안이 다음 에포크에 적용된다. 만약 어떤 제안도 50%를 넘지 못하거나, 현행 상태 유지안이 50%를 획득하거나, 정족수에 도달하지 못한 경우에는 어떤 제안도 채택되지 않으며 파라미터는 현재 값으로 유지된다.
4. 투표하거나 제안을 제출하기 위해 최소 스테이킹 요건은 존재하지 않는다.

거버넌스로 결정되는 파라미터(Governance-Determined Parameters)

이 거버넌스 프레임워크를 통해 조정되는 파라미터는 총 네 가지이다:

1. 샤드를 전송하는 노드에 부과되는 샤드 복구 비용. 이는 샤드 단위의 페널티다.
2. 샤드를 수신하는 노드에 부과되는 샤드 복구 비용. 이 역시 샤드 단위의 페널티다.
3. 데이터 챌린지의 50% 이상을 실패한 노드에 부과되는 비용. 이는 샤드 단위 페널티이며, 평가 시 노드가 보유한 샤드 수만큼 곱해진다.
4. 데이터 챌린지를 50% 이상 발행하지 않은 노드에 부과되는 비용. 이 또한 샤드 단위 페널티이며, 평가 시 보유한 샤드 수만큼 곱해진다.

이러한 파라미터를 수정하는 어떤 제안도 다음 두 가지 제약 조건을 만족해야 하며, 양수성(positivity)과 같은 자명한 조건 외에도 이를 따라야 한다. 첫째, 송신 노드의 샤드 복구 비용은 수신 노드의 복구 비용보다 크거나 같아야 한다. 이는 전송 책임이 송신 노드에 있음을 보장한다. 둘째, 데이터 챌린지의 50% 이상을 실패한 노드에 부과되는 비용은 수신 노드의 샤드 복구 비용보다 크거나 같아야 한다. 이는 수신 노드가 실패한 샤드 전송을 정확히 보고하도록 유도하기 위함이다.

6.5 인센티브 기반 읽기(Incentivized Reads)

*Walrus*는 무엇보다도 blob의 안정적인 저장을 우선적으로 설계하였다. 데이터를 접근하기 위해 *Walrus*는 저장 노드가 무료이면서 빠른 읽기 접근을 제공하도록 장려하지만, 이에 대한 엄격한 요구 사항은 없으며 모든 제공은 최선의 노력(best-effort) 기준에 따른다.

이 모델은 저장 노드들이 전반적으로 *Walrus*의 성공에 이해관계를 공유하고 있다는 전제에서 작동한다. 추가적으로, 캐시나 콘텐츠 전송 네트워크(CDN)와 같은 다른 제공자들이 독자 또는 작성자가 지불하는 대가로 고품질의 읽기 접근을 제공할 수 있을 것으로 예상된다. 하지만 엄밀히 말하면, 합리적인 저장 노드는 데이터를 단순히 보관만 하고 제공하지 않을 수도 있다. 이는 다른 노드가 대신 응답해 줄 것이라는 기대 때문이며, 실제로 $f + 1$ 개의 응답만 있으면 충분하기 때문이다. 이는 고전적인 공공재 문제(public goods problem) [36]로 이어질 수 있으며, 결국 어떤 저장 노드도 클라이언트 요청에 응답하지 않는 상황으로 전개될 수 있다. 이러한 역동성은 시스템을 사용할 수 없게 만든다.

이 문제에 대한 몇 가지 가능한 해결책이 있다. 본 문서에서는 세 가지를 간략히 소개한다: 노드 서비스 모델, 온체인 바운티, 그리고 경량 노드(light nodes)를 *Walrus*에 정식으로 포함시키는 방안이다.

노드 서비스 모델(Node Service Models)

하나의 해결책은 사용자가 저장 노드와 양자 유료 계약을 체결하여 데이터를 읽는 방식이다. 이 모델은 단순한 파트너십을 넘어 다양한 형태를 가질 수 있다. 예를 들어, 특정 노드 집합이 사용자에게 유료 읽기 접근 엔드포인트를 제공하면서 일정 수준의 서비스 보장을 할 수 있다. 이때 요금은 읽는 사용자가 지불하거나, blob을 공개적으로 제공하고자 하는 작성자(예: 웹사이트를 퍼블리싱하는 경우)가 지불할 수 있다. 또는 해당 노드들이 기업 수준의 계약을 체결하여 특정 기관이 데이터를 접근하고 재판매할 수 있도록 하는 방식도 가능하다. 실질적으로 이 경로는 캐시, 퍼블리셔, 기타 콘텐츠 제공자가 *Walrus* 위에 구축될 때 주된 상호작용 방식이 될 가능성이 높다. 이들은 대상 파일 집합에 대해 일상적인 사용자 접근을 처리하고, 장기적으로는 *Walrus*로부터 해당 집합을 갱신 및 업데이트받는 구조를 따른다.

이 메커니즘은 노드가 데이터를 반환할 수 있도록 고정된 인센티브를 제공하지는 않지만, 동적인 인센티브를 제공한다. 의무를 이행하지 않는 노드는 나쁜 평판을 얻게 되어 향후 비즈니스 기회를 잃게 된다. 우리는 전 세계적으로 지연이 매우 낮고, 캐시 및 CDN 기반으로 구성된 읽기 인프라가 *Walrus*를 중심으로 발전할 것으로 기대한다.

이 메커니즘의 주요 장점은 *Walrus* 자체를 수정할 필요가 없다는 점이다. 다만, 추가적인 개발 없이 실제 이 메커니즘을 사용할 경우 최종 사용자에게는 다소 복잡할 수 있다. 사용자는 자신의 인코딩된 데이터를 샤드 전반에서 어떻게 찾을지 알지 못하거나, 노드와 직접 협상하는 방법을 모를 수 있다. 그러나 중개자는 이러한 복잡성을 상당 부분 추상화할 수 있으며, 장기적인 비즈니스 관계를 노드에 제공함으로써 계약 이행을 유도하는 강력한 인센티브를 제공할 수 있다.

온체인 바운티(On-Chain Bounties)

또 다른 해결책은 최선의 노력 방식의 읽기가 실패했을 때, 사용자가 *Walrus*에서 데이터를 얻기 위해 온체인 바운티를 게시하는 것이다(이는 SLA 기반 서비스를 제공하는 캐시나 퍼블리셔에 의해 대리 수행될 수도 있다). 구체적으로, 데이터가 필요한 사용자가 바운티를 게시하고, 데이터를 수신하면 해당 응답 노드에 바운티를 할당하거나, 충분한 데이터를 받지 못한 경우 바운티 회수를 위한 챌린지를 게시한다. 이는 저장 노드가 바운티를 획득하기 위해 응답하도록 유도한다. 만약 사용자가 부정하게 바운티를 회수하려 한다면, 노드들은 데이터를 온체인에 게시함으로써 해당 챌린지를 무효화하고 사용자에게 데이터를 제공할 수 있다.

이 해결책 또한 *Walrus* 자체의 변경 없이 구현 가능하다. 이러한 바운티는 Sui 상의 스마트 계약을 통해 중개된다. 그러나 단점도 두 가지 존재한다. 첫째, 특히 바운티 지불 여부에 대해 빈번한 분쟁이 발생할 경우, 이 방식은 데이터를 수신하기에 번거로운 경로가 될 수 있다. 둘째, 최종 사용자 입장에서 이 방식은 복잡하다. 바운티를 게시하고, 크레딧을 할당하며, 챌린지를 제출하고, 이후 데이터를 다운로드해야 하기 때문이다.

6.6 라이트 노드 샘플링을 통한 탈중앙화 보안 (Decentralized Security Through Light-Node Sampling)

시스템 내에서 데이터가 극단적으로 접근 불가능한 경우를 해결하기 위한 마지막 방안은, Data Availability 시스템에서의 라이트 노드와 동일한 역할을 수행하는 두 번째 유형의 참여자들을 활성화하고 인센티브를 제공하는 것이다 [1]. 이러한 라이트 노드들은 보다 탈중앙화된 보안의 두 번째 계층을 제공한다. 저장 노드는 이 참여자들과 상호작용하기 위한 특정 요건을 충족해야 하며, 이들에게 데이터를 제공하지 못할 경우 슬래시될 수 있다.

이는 보안 보장을 충족시키기 위한 측면에서 자연스럽게 더 복잡하지만, 훨씬 더 견고한 구조이다. 또한, 커뮤니티가 전체 저장 노드를 운영하지 않고도 *Walrus*의 운영에 참여할 수 있는 경로를 제공한다. 마지막으로, 이 구조는 온체인 바운티 솔루션과도 결합될 수 있다.

프로토콜(Protocol)

이 역시 우리의 2D 인코딩 방식이 강점을 발휘하는 지점이다. 각 심볼의 크기가 단지 $O(\frac{|B|}{n^2})$ 이므로, 라이트 노드는 자신이 중요하다고 간주하는 파일에서 무작위로 샘플링된 심볼만 저장하면 되며, 해당 데이터가 접근 불가능해질 경우 바운티가 게시되기를 기대할 수 있다. 또한, 저장 노드가 요청된 심볼을 전송하지 않는 경우, 라이트 노드는 온체인 바운티 메커니즘과 유사하게 제한된 속도로 블레임(blame)을 게시할 수 있다.

프로토콜은 매우 단순하다. 데이터가 작성된 시점부터, 라이트 노드는 저장 노드로부터 최선의 노력 방식으로 심볼을 직접 읽어오거나, 캐시를 통해 blob을 다운로드한 후 재인코딩함으로써 샘플링할 수 있다. 이후 온체인 바운티가 게시되면, 누락된 심볼을 서명과 함께 클라이언트에게 가장 먼저 전송한 라이트 노드는 해결 트랜잭션(resolution transaction)에 포함된다(공정성을 원할 경우 체인을 사용할 수 있으며, 그렇지 않은 경우 클라이언트가 직접 보상할 수 있다 — Sybil 방지를 위해). 해결 트랜잭션은 데이터 복구에 기여한 모든 라이트 노드의 주소와 각 노드가 제공한 심볼 수를 포함하며, 그에 따라 보상이 분배된다. 이 보상은 인센티브 기반 읽기에 기여한 검증자들과 라이트 노드 간에 비례하여 분배된다. 각 심볼에 대한 보상은 각 슬라이버

(sliver) 단위 보상과 비례해야 하며, 이는 저장 노드가 보상을 극대화하기 위해 Sybil 라이트 노드를 생성하지 않도록 하기 위함이다.

향후 우리는 재구성 이후 샤드 복구에 실질적으로 기여한 라이트 노드에 대한 지불을 간소화할 수 있도록, 오피체인 채널 기반의 더욱 효율적인 프로토콜을 설계할 예정이다.

7 평가(Evaluation)

우리는 Rust로 구현된 *프로덕션* 수준의 네트워크 기반 멀티코어 *Walrus* 저장 노드를 구현하였다. 모든 네트워킹은 axum [35]을 통해 HTTPS를 사용하며, 암호화는 fastcrypto [21], 저장소는 rocksdb [28], 이레이저 코딩은 reed-solomon-simd [24]를 이용한다. 우리는 빠른 블록체인의 예시로 Sui [41]와 연결하는 방식을 선택하였다. 해당 코드베이스는 오픈소스로 공개하였다¹².

우리는 *Walrus*의 성능 및 확장성을 실제 공개된 테스트넷에서 평가한다. 이는 실제 환경, 실제 사용자, 우리가 통제할 수 없는 인프라에 시스템을 노출시키는 가장 현실적인 평가 방식이다. 우리는 3월 22일에 종료되는 60일 동안 *Walrus* 테스트넷을 관측하였다.

본 평가의 목적은 다음과 같은 주장들을 입증하는 것이다:

1. **C1 (저지연성):** *Walrus*는 네트워크 지연에 의해 제한되는 낮은 지연 시간을 달성한다.
2. **C2 (처리량):** *Walrus* 클라이언트는 높은 읽기 및 쓰기 처리량을 달성한다.
3. **C3 (확장성):** *Walrus*의 전체 용량은 저장 노드 수에 비례하여 확장된다.

7.1 실험 환경(Experimental Setup)

Walrus 테스트베드는 탈중앙화되어 있으며, 총 105개의 독립적으로 운영되는 저장 노드와 1,000개의 샤드로 구성되어 있다. 모든 측정값은 노드 운영자들이 자발적으로 제공한 데이터를 기반으로 한다.

샤드는 각 운영자의 스테이크에 따라 할당되며, 이는 메인넷 배포 모델을 반영한다. $f + 1$ 쿼럼을 만족하려면 최소 19개의 노드가 협력해야 하며, $2f + 1$ 쿼럼을 위해서는 38개 노드가 필요하다. 어떤 운영자도 18개를 초과하는 샤드를 보유하지 않는다. 노드는 리투아니아, 미국, 프랑스, 캐나다, 네덜란드, 태국, 아일랜드, 러시아 등 최소 17개국에 분포되어 있다. 11명의 운영자는 위치를 공개하지 않았다. Figure 6은 지역별 샤드 분포를 보여준다. “eu-west” 지역은 최소 5개국에서 수집된 샤드를 통합한 것이다. 지역 정보가 누락된 약 220개의 샤드는 “unknown”으로 분류된다. Figure 7은 호스팅 제공자별 샤드 분포를 보여준다. “Self-Hosted” 노드는 온프레미스에서 운영되며, “Unknown”은 제공자 정보가 없는 경우이다.

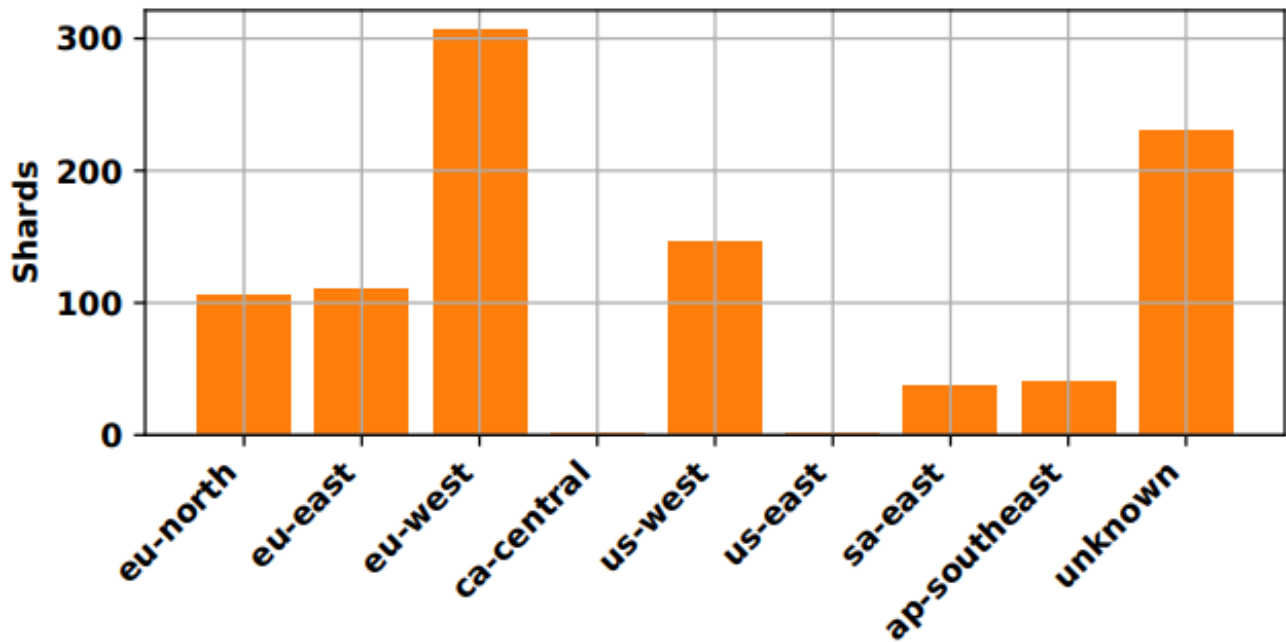
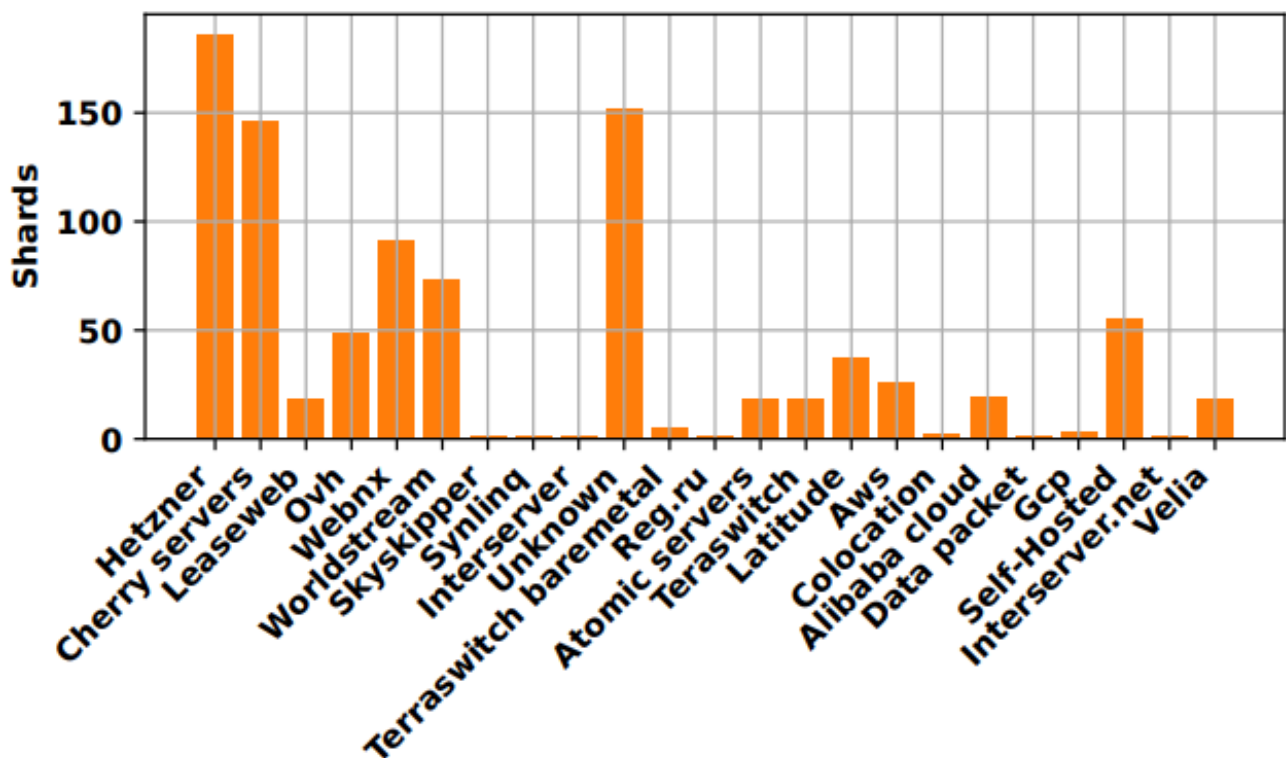


Figure 6: Geo-distribution of shards.



대부분의 노드는 Ubuntu (22.04 또는 24.04)를 실행하며, 최소 16개의 CPU 코어, 128GB RAM, 1Gbps 대역폭을 갖춘다. 하드웨어는 Intel과 AMD CPU, HDD, SSD, NVMe 저장장치 등으로 다양하다. 노드의 저장 용량은 15TB에서 400TB 사이이며, 중앙값은 56.9TB, P90은 69.9TB이다.

7.2 시스템 성능(System Performance)

우리는 클라이언트의 관점에서 성능을 평가하였다. 두 개의 클라이언트를 AWS m5d.8xlarge 인스턴스 (10 Gbps 대역폭, 32 vCPU, 128 GB RAM, Ubuntu 22.04)에 배포하였다. 하나는 미국 동부(버지니아 북부), 다른 하나는 캐나다 중부 리전에 위치한다.

Walrus 지연 시간. Figure 8은 클라이언트가 경험한 종단 간(end-to-end) 지연 시간을 보여준다. 측정은 클라이언트가 blob을 인코딩하기 직전부터 시작하여, 블록체인에서 가용성 증명(proof-of-availability)이 확인될 때까지의 시간으로 진행된다. 각 데이터 포인트는 5분 간의 실행 중 p50(중간값)을 나타내며, 에러 바(error bar)는 p90(90백분위)을 나타낸다.

그래프는 큰 blob에 대해서도 읽기 지연 시간이 낮게 유지된다는 것을 보여준다. 작은 blob(20MB 미만)의 경우 지연 시간은 15초 이하로 유지된다. 큰 blob(130MB)의 경우 지연 시간은 약 30초까지 증가한다.

쓰기 지연 시간은 일관되게 읽기 지연 시간보다 길다. 작은 blob(20MB 미만)의 경우, 쓰기 지연 시간은 상대적으로 일정하며 25초 미만으로 유지된다. 이 오버헤드는 주로 블록체인과의 상호작용 및 메타데이터를 모든 저장 노드에 업로드해야 하는 요구사항 때문이며, blob의 크기 자체와는 무관하다. 큰 blob(40MB 초과)의 경우, 지연 시간은 blob 크기에 따라 선형적으로 증가하며 이는 네트워크 전송이 주된 비용이 되기 때문이다. Figure 9와 Figure 10은 각각 작은 blob(1KB)과 큰 blob(130MB)에 대한 지연 시간을 단계별로 나누어 시각화한 것이다. 각 쓰기 작업은 다음의 다섯 가지 주요 단계로 구성된다: **인코딩(encoding)** (blob을 이레이저 코딩하는 시간), **상태 확인(check status)** (blob의 현재 상태를 확인하는 시간), **정보 요청(get info)** (blob 상태를 가져오고 공간을 예약하는 시간), **저장(store)** (슬리버를 저장 노드에 업로드하는 시간), **PoA 게시(publish PoA)** (가용성 증명을 블록체인에 커밋하는 시간). 작은 blob의 경우, 메타데이터 처리 및 블록체인 게시로 인한 고정 오버헤드가 지배적이며 약 6초—전체 쓰기 지연 시간의 약 50%—을 차지한다. 큰 blob의 경우, 저장 단계가 네트워크 전송으로 인해 지배적인 비용이 되며, 메타데이터 처리 및 블록체인 상호작용 시간은 상대적으로 일정하게 유지된다.

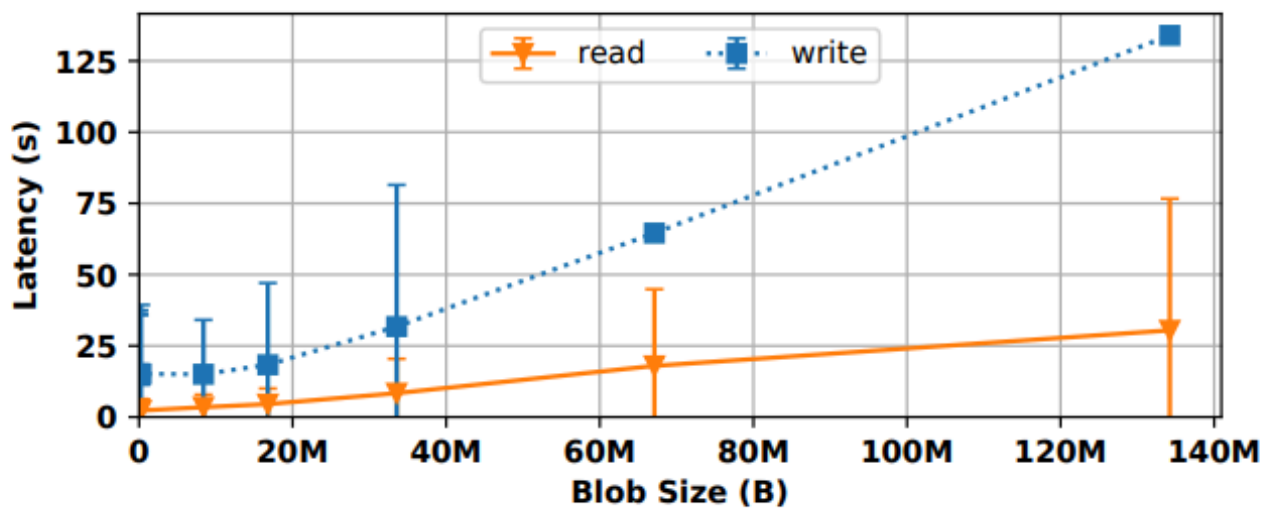


Figure 8: Latency for different blob sizes.

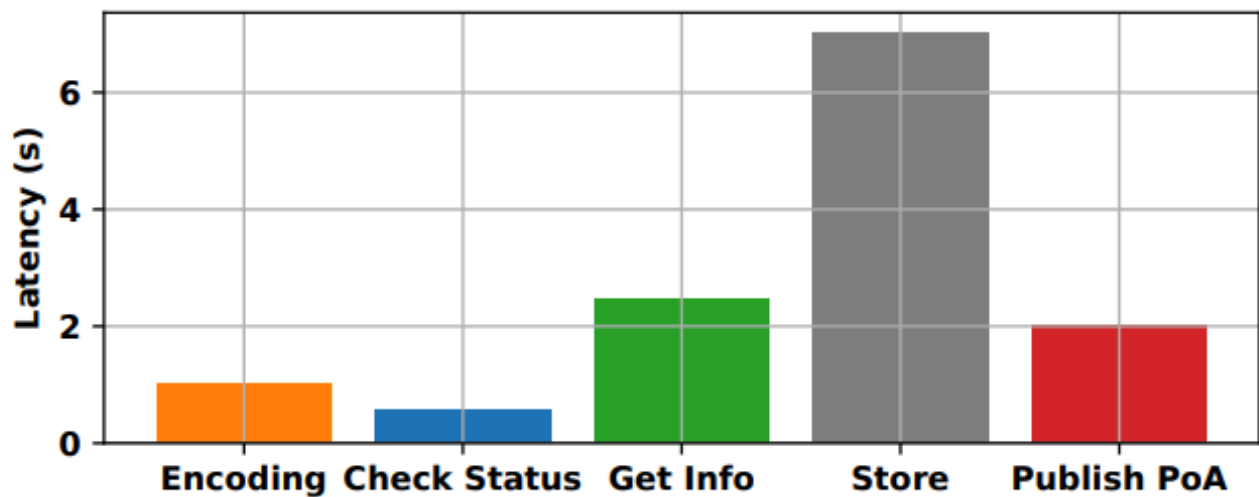


Figure 9: Latency breakdown for small blobs (1KB).

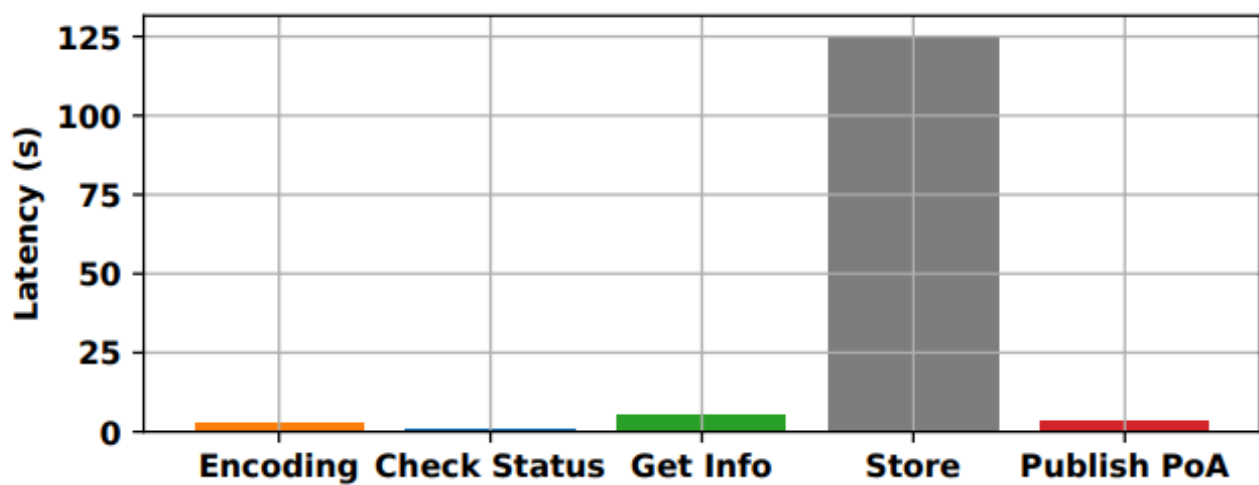


Figure 10: Latency breakdown for large blobs (130MB).

이러한 결과는 주장 **C1**을 입증한다: *Walrus*는 낮은 지연 시간을 제공하며, 지연은 네트워크 속도에 의해 제한된다.

단일 클라이언트 처리량(Single Client Throughput). Figure 11은 단일 클라이언트가 달성할 수 있는 바이트 단위 초당 처리량을 보여준다. 예상대로, 읽기 처리량은 대부분 네트워크 기반 상호작용이기 때문에 blob 크기에 따라 선형적으로 확장된다. 쓰기 처리량은 약 **18 MB/s**에서 정체되며, 이는 블록체인 및 저장 노드와 여러 번 상호작용해야 하기 때문이다. 이는 사용자가 더 빠른 업로드를 할 수 없다는 의미는 아니다. Sui는 초당 훨씬 더 많은 트랜잭션을 지원하기 때문에, 단지 *하나의 blob*이 더 빠르게 업로드될 수 없다는 의미일 뿐이다. 훨씬 더 큰 blob을 다룰 경우, 사용자는 여러 클라이언트를 병렬로 배치하여 각각 데이터를 분할 업로드함으로써 효과적으로 fan-out 패턴을 구성할 수 있다.

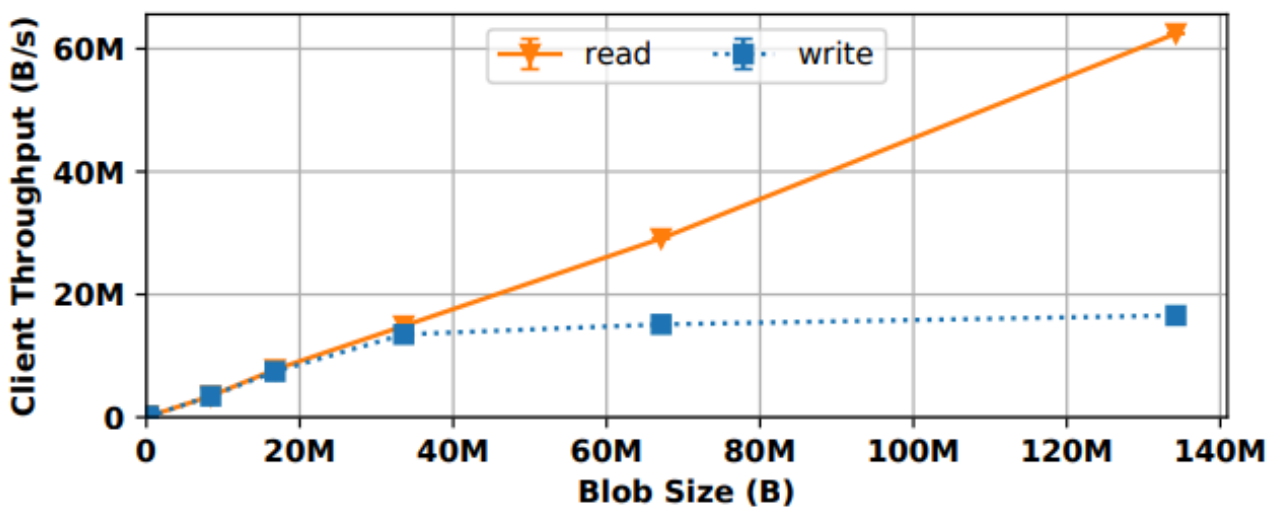


Figure 11: Single client throughput for different blob sizes.

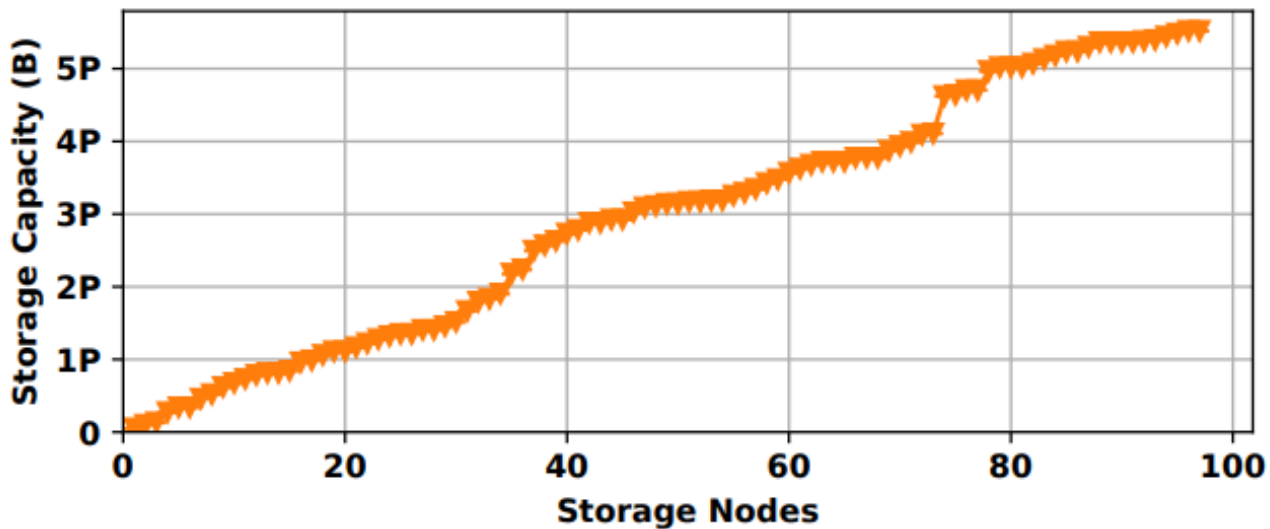


Figure 12: Storage capacity versus committee size.

이러한 결과는 주장 **C2**를 입증한다: *Walrus*는 클라이언트가 높은 읽기 및 쓰기 처리량을 달성할 수 있도록 한다.

7.3 확장성(Scalability)

60일 동안 *Walrus*는 중앙값 기준 1.18TB의 슬리버(silver)와 221.5GB의 blob 메타데이터를 저장하였다 (P90 기준 각각 1.08TB, 46.34GB). 7.1절에서 설명한 바와 같이, 각 저장 노드는 15TB에서 400TB 사이의 용량을 제공한다. 그럼에도 불구하고, 시스템 전체는 **5PB** 이상을 저장할 수 있으며, 이는 *Walrus*의 핵심 특징 중 하나이다. Figure 12는 *Walrus*의 전체 저장 용량이 커미티(committee) 크기에 따라 어떻게 확장되는지를 보여준다. 이 결과는 우리의 최종 주장 **C3**, 즉 시스템의 저장 용량이 저장 노드 수에 비례하여 성장한다는 주장을 뒷받침한다.

8 관련 연구(Related Work)

검열 저항적인 스토리지 및 blob 데이터 전파는 초기 피어-투-피어(P2P) 운동과 탈중앙화의 필요성을 촉진한 핵심 동기 중 하나였다. 학계에서는 1996년 Anderson이 Eternity 서비스 [3]를 제안하며, 문서가 억압되지 않도록 보장하고자 했다. 상업 및 오픈소스 커뮤니티에서는 Napster [9], Gnutella [33], Free Haven [15], 그리고 초기 Freenet [11]과 같은 시스템들이 비정형 토폴로지 내에서 노드를 활용하여 주로 미디어 파일을 저장, 라우팅 및 배포하기 위한 구조를 형성했다. 이러한 시스템은 검색과 탐색을 위해 중앙집중형 또는 범람(flood fill) 알고리즘에 의존하였으며, 파일의 전체 복제를 통해 운영되었고 종종 응답 라우팅에 사용되는 노드에 복사본이 저장되었다. 이들은 최선의 노력(best-effort) 방식의 보안만을 제공하며, 성능은 낮았다.

이후 2000년대 초반에는 구조화된 피어-투-피어 토폴로지가 제안되었으며, 이는 Chord [38], Pastry [34], Kademlia [26] 등의 분산 해시 테이블(DHT)의 형태로 등장하였다. 이러한 방식은 주로 검색 성능을 개선하고, 파일당 복제 계수를 줄이기 위한 목적이었다.

DHT는 합의(consensus)나 전체 상태 머신 복제 없이도 작동할 수 있다는 점에서 주목할 만하다. 하지만, 이 시스템들은 여러 공격에 취약한 것으로 나타났다. Sybil 공격 [16]은 이 문맥에서 처음 명명되고 식별되었으며, 라우팅 공격 [44]에 방어하기 어려운 구조를 가진다. 이러한 공격 중 다수는 현재 이 시스템들을 사용하는 플랫폼에도 영향을 미친다 [42]. 이후 Bittorrent [12]가 파일 배포 분야에서 지배적인 응용 프로그램으로 부상했는데, 이는 단순성과 내장된 인센티브 구조 덕분이다. 초기에는 저장소에 대해 전체 복제 전략과 중앙집중형 트래커를 사용하였고, 이후에는 Kademlia 기반의 분산형 트래커를 추가하였다.

이러한 초기 시스템들과 달리 *Walrus*는 Sui 블록체인 [8]을 통해 모든 노드와 그들의 최신 메타데이터에 대한 완전하고 일관된 목록을 유지한다. *Walrus*는 이러한 노드들이 인프라 수준의 노드이며, 높은 churn(노드의 잦은 이탈과 재진입)에 시달리지 않고, 인센티브와 지불을 위해 운영되며 재구성 프로토콜을 기반으로 시스템에 출입한다고 가정한다.

블록체인 시대에 들어서면서, IPFS [5]는 파일을 위한 탈중앙화 스토리지를 제공하고 있으며, 블록체인 시스템과 탈중앙화 앱에서 스토리지 용도로 광범위하게 사용되고 있다. IPFS는 블록에 대해 콘텐츠 주소 가능한 저장(content-addressable storage)을 제공하고, DHT를 통해 파일 복제본과 이를 저장하는 노드 간의 링크를 유지한다. 파일을 게시한 사용자는 복제본이 유지되도록 노드에 파일을 고정(pin)해야 하며, 이는 일반적으로 대가 지불을 필요로 한다. 기본 스토리지 방식은 파일당 소수 노드에 전체 복제를 수행한다.

Filecoin [30]은 IPFS를 확장한 형태로, 롱기스트 체인 블록체인과 FIL이라는 암호화폐를 통해 저장 노드가 파일 복제본을 유지하도록 인센티브를 부여한다. 게시자는 몇몇 노드와 저장 계약을 체결하고, 지불은 암호화폐로 이루어진다. Filecoin은 저장 노드가 복제본을 삭제할 위험을 줄이기 위해, 노드가 서로 다른 방식으로 인코딩된 복사본을 보관하게 하고, 이를 상호 챌린지를 통해 검증하도록 한다. 이러한 복사본은 릴레이 공격을 방지하기 위해 원본에서 재생성하는 데 시간이 오래 걸리도록 인코딩되어 있다. 결과적으로 사용자가 원본 파일에 접근하고자 할 경우, 복사본을 디코딩하기까지 시간이 오래 걸릴 수 있으며, 핫 카피(hot copy)를 보유한 저장 노드가 없다면 더욱 그렇다. 핫 카피 보관에 대한 내장 인센티브가 없기 때문에, 이러한 서비스는 일반적으로 추가 비용이 발생한다.

Arweave [46]는 Proof-of-Access 알고리즘을 통해 느린 읽기 문제를 완화하며, 가능한 많은 파일을 로컬에 보관하는 노드에 더 많은 보상을 제공함으로써 저장 노드의 참여를 유도한다. 이는 전체 복제 전략과 함께 구현되며, 전통적인 상태 머신 복제와 유사한 수준의 복제율을 달성한다. 또한, 이 시스템은 사전 지불 메커니즘을 통해 "영구 저장"만을 허용하며, 파일 수명 조절 및 삭제 기능이 없고, 자본 효율성 면에서도 비효율적이다. (왜냐하면 전체 지불이 선불로 이루어지기 때문이다.)

Filecoin과 Arweave와 달리, Walrus는 이레이저 코딩을 활용하여 $4.5\times$ 라는 매우 낮은 오버헤드로 데이터를 유지하며, 전체 샤드 중 최대 $2/3$ 가 손실되더라도 데이터를 보존할 수 있고, 최대 $1/3$ 의 샤드가 응답하지 않더라도 쓰기 작업을 계속 수행할 수 있도록 설계되었다. 또한, Walrus는 노드 관리와 인센티브를 위한 자체 블록체인을 구현하지 않고, 대신 Sui를 사용한다.

Storj [39]는 이레이저 코딩을 활용하여 낮은 복제 계수를 달성하는 또 다른 탈중앙화 저장 솔루션이다. 이 시스템은 Reed-Solomon 기반의 이레이저 코딩을 $29/80$ 구성으로 구현하는데, 하나의 파일을 80개 조각으로 인코딩하며 이 중 29개만으로 복원이 가능하다. 이 접근 방식은 $2.75\times$ 의 복제 계수를 제공하여 기존 시스템 대비 저장 비용을 크게 줄인다. 하지만 Storj의 주요 한계는 손실된 조각을 효율적으로 복구하지 못한다는 점에 있다. 이 시스템은 사용자에게 전체 파일을 복원하고 다시 인코딩하여 손실된 조각을 복구하도록 요구한다. 반면, Walrus는 Red Stuff를 사용하여 이레이저 코딩 체계의 복구를 효율적으로 수행할 수 있는 메커니즘을 내장하고 있으며, 이는 허가 없는 시스템에서 자연스럽게 발생하는 노드 churn 상황에서 특히 중요하다. Red Stuff는 Twin-code 프레임워크 [31] 위에 구축되며, 두 개의 선형 인코딩을 활용해 슬리버 복구 효율을 향상시킨다. 그러나 Twin-code 프레임워크 [25]와는 달리, Red Stuff는 서로 다른 크기의 차원에 걸쳐 데이터를 인코딩하며, 인증된 데이터 구조(authenticated data structures)를 통합하여 Section 2에서 정의한 완전성(Completeness)을 만족시키고 비잔틴 장애 허용(Byzantine Fault Tolerance)을 보장한다.

현대의 블록체인들은 일정 수준의 스토리지를 제공하지만, 모든 밸리데이터가 전체 복제를 수행해야 하는 비용과 장기간의 보존 기간(검증 가능성을 위해 필요한) 때문에 큰 blob을 저장하는 것은 현실적으로 매우 비싸다. 특히 Ethereum 생태계 내에서는 현재 L2 확장 전략이 메인 체인에 트랜잭션 blob을 게시하는 방식으로 구현되고 있으며, 이는 실행될 트랜잭션 번들을 나타내며, 영지식 증명(zero-knowledge proof)이

나 사기 증명(fraud proof)을 통해 검증된다. 이러한 필요를 오프체인에서 충족시키기 위해 Celestia와 같은 가용성 샘플링(availability sampling) 기반 특수 네트워크가 등장하였다 [2]. Celestia에서는 2차원 Reed-Solomon 코드를 사용하여 blob을 인코딩하고, 코드워드를 라이트 노드(light node)에 분산하여 ‘신뢰 불필요(trustless)’한 가용성을 지원한다. 그러나 Celestia에서는 모든 blob이 시스템 밸리데이터에 의해 완전히 복제되며, 약 한 달 정도의 제한된 기간 동안만 유지된다. Walrus는 대신, 임의의 보존 기간을 허용하는 가용성 증명과 노드당 저장 비용을 줄인 구조를 제공하여, 저렴하게 시스템 확장을 가능케 한다.

9 결론(Conclusion)

우리는 빠른 이레이저 코딩과 현대 블록체인 기술을 활용한 새로운 접근 방식의 탈중앙화 blob 저장 시스템인 Walrus를 소개한다. Red Stuff 인코딩 알고리즘과 Sui 블록체인을 활용함으로써, Walrus는 높은 복원력과 낮은 저장 오버헤드를 달성하고, 동시에 효율적인 데이터 관리와 확장성을 보장한다. 본 시스템은 에포크 단위로 동작하며, 모든 연산은 blob_id를 기준으로 샤딩되어 대용량 데이터를 효과적으로 처리할 수 있다. Red Stuff의 혁신적인 2차원 BFT 인코딩 프로토콜은 효율적인 데이터 복구, 부하 분산, 저장 노드의 동적 가용성을 지원하며, 기존 탈중앙화 저장 시스템들이 직면한 주요 문제들을 해결한다.

더불어, Walrus는 네트워크 동기화 가정에 의존하지 않는 데이터 가용성 보장을 위한 스토리지 증명을 도입하며, 커미티 재구성 프로토콜을 통해 네트워크 변화 중에도 데이터 가용성이 중단되지 않도록 한다. 이러한 기능을 결합함으로써, Walrus는 확장 가능하고 회복력 있는 탈중앙화 저장을 실현하며, 합리적인 비용으로 높은 신뢰성, 무결성, 감시 가능성, 가용성을 제공한다. 우리의 기여는 비동기 완전 데이터 공유 (Asynchronous Complete Data-Sharing) 문제 정의, Red Stuff 프로토콜 제안, 효율적인 스토리지 증명을 위한 비동기 챌린지 프로토콜 도입에 있으며, 이는 향후 탈중앙화 저장 기술의 발전을 위한 토대를 마련한다.

1. 최근 예시로는 비트코인 및 기타 체인에서의 ‘inscriptions’가 있으며, 관련 내용은 <https://medium.com/@thevalleylife/crypto-terms-explained-exploring-bitcoin-inscriptions-51699dc218d2> 참조. ↩

2. 최근 개념 증명 공격 사례는 <https://moxie.org/2022/01/07/web3-first-impressions.html> 참조. ↩

3. 25개 저장 노드가 모두 악의적이며 파일을 삭제할 확률은 $3^{-25} = 1.18 \times 10^{-12}$ 이다. ↩

4. 작성자 W 는 모든 슬리버 간의 일관성을 증명할 수도 있지만, ACDS에서는 과도한 수준이다. ↩

5. 사용된 커밋먼트 스킴에 따라 추가로 $O(\log n)$ 의 비용이 발생할 수 있다. ↩

6. 저장 노드는 인코딩된 조각과 함께 자신의 슬리버에 대한 커밋과 오프닝 증명도 계산해야 한다. ↩

7. 사용되는 커밋 스킴에 따라 상수 수준의 의존성이 있을 수 있다. ↩

8. 예를 들어, 노드가 자신의 자본을 마지막 순간에 슬쩍 언스테이킹하여 위험 회피를 하는 것을 방지하기 위해, 노드 자신의 자본에 대해서는 한 에폭 더 지연된 반환을 요구하는 확장도 고려할 수 있다. ↩

9. 이를 대신할 수 있는 대안 모델은, 노드의 스테이크와 자가 신고한 저장 용량을 모두 고려해 샤드를 할당하는 방식이다. 하지만 이 경우, 노드가 스토리지 용량을 줄일 경우 슬래시를 적용하거나, 감소분을 보완한 다른 노드에게 보상을 지급해야 하므로 운영 복잡성이 증가한다. ↩

10. 이 메커니즘은 *Walrus*에서 많은 적대적 시나리오 하에서도 노드의 탈퇴를 처리할 수 있으나, 완전히 응답하지 않는 노드의 시나리오에서는 우아하지 못한 방식으로 동작한다. 완전히 응답하지 않는 노드는 결국 스테이크에 부과된 페널티를 통해 샤드를 잃게 되지만, 이 과정은 점진적이며 여러 에포크에 걸쳐 일어나므로 그 동안 *Walrus*의 운영에 제약을 줄 수 있다. 향후 개발 아이디어 중 하나는, 여러 연속 에포크에서 데이터 챌린지의 초다수 실패를 기록한 노드로부터 모든 샤드를 몰수하는 긴급 마이그레이션 시스템을 *Walrus*가 구축하는 것이다. ↩

11. 백분위 수 기준 방식은 시장 가격을 왜곡하려는 소수 노드의 제출을 필터링하고, 대다수 노드의 의사 결정이 시장 가격 형성에 반영되도록 한다. ↩

12. 블라인드 리뷰를 위해 링크는 생략함. ↩