간단한 테스트로 대부분의 중대한 장애를 예방할 수 있습니다: 분산된 데이터 집약적 시스템의 생산 장애에 대한 분석

딩 위안, 유 루오, 신 좡, 길레르메 레나 로드리게스, 쉬 자오, 용레 장, 프라나
이 U. 자인, 마이클 스텀, *토론토 대학교*

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan
978-1-931971-16-4

논문집에 대한 공개 액세스
제11회 운영 체제 설계 및 구현에 관한 USENIX 심포지엄

간단한 테스트로 대부분의 중대한 장애를 예방할 수 있습니다. 분산 데이터 집약적 시스템의 생산 장애 분석

딩 위안, 유 루오, 신 쟝, 길레르메 레나 로드리게스, 쉬 자오, 용레 장,
프라나이U. 자인, 마이클 스텀
토론토 대학교

초록

프로덕션 품질의 대규모 분산 시스템은 여전히 주기적으로 장애를 일으키며, 때로는 대부분의 사용자 또는 모든 사용자가 중단 또는 데이터 손실을 경험하는 치명적인 장애를 일으키기도 합니다. 저희는 하나 또는 여러 개의 결함이 결국 사용자가 볼 수 있는 장애로 발전하는 과정을 이해하기 위해, 무작위로 선택된 198건의 사용자 보고 장애를 조사한 종합적인 연구 결과를 발표합니다. 테스트 관점에서 볼 때, 거의 모든 장애를 재현하는 데 3개 이하의 노드만 필요한 것으로 나타났습니다. 이러한 서비스가 일반적으로 매우 많은 수의 노드에서 실행된다는 점을 고려할 때 이는 좋은 소식입니다. 그러나 장애를 트리거하려면 여러 개의 입력이 필요하며 입력 사이의 순서가 중요합니다. 마지막으로, 이러한 시스템의 오류 로그에는 일반적으로 오류와 장애를 유발한 입력 이벤트에 대한 충분한 데이터가 포함되어 있어 프로덕션 장애를 진단하고 재현할 수 있다는 사실을 발견했습니다.

소프트웨어 설계에 대한 이해 없이도 최후의 방어선인 오류 처리 코드에 대한 간단한 테스트만 수행해도 대부분의 치명적인 장애를 쉽게 예방할 수 있었다는 사실을 발견했습니다. 우리는 치명적인 장애를 일으킨 버그에서 세 가지 간단한 규칙을 추출하고 이러한 버그를 찾아낼 수 있는 정적 검사기인 Aspirator를 개발했습니다. Aspirator를 사용하여 확인된 버그를 수정했다면 치명적인 장애의 30% 이상이 사전에 방지되었을 것입니다. 9개의 분산된 시스템 코드에서 Aspirator를 실행한 결과 개발자가 수정하거나 확인한 143개의 버그와 잘못된 관행을 찾아냈습니다.

1 소개

실제 분산 시스템은 필연적으로 중단을 경험합니다. 예를 들어, 2011년에 발생한 아마존 웹 서비스 중단으로 인해 Reddit, Quora, 뉴욕 타임즈 웹사이트의 일부인 FourSqure 및 약 70개의 다른 사이트가 다운되었으며[1], 2013년에 발생한 구글 서비스 중단으로 인해 인터넷 트래픽이 40% 감소했습니다[21]. 또 다른 사건에서는 DNS 오류로 인해 스웨덴이 인터넷에 접속할 수 없게 되어 `.se` 도메인의 모든 URL을 매핑할 수 없게 되었습니다[46].

이러한 시스템 중 상당수가 고가용성을 위해 설계되었고, 일반적으로 우수한 소프트 웨어 엔지니어링 관행을 사용하여 개발되었으며, 강도 높은 테스트를 거쳤음을 감안할 때, 이는

는 이러한 시스템에서 왜 여전히 장애가 발생하는지, 그리고 *이러한 시스템의 복원력을 높이기 위해 무엇을 할 수 있는지에 대한 의문을* 제기합니다. 이러한 질문에 대한 답을 찾기 위해 구성 요소 장애를 처리하도록 설계되었으며 생산 환경에서 널리 사용되는 5개의 데이터 집약적 분산 시스템에서 무작위로 표본을 추출하여 사용자가 보고한 198건의 장애를 연구했습니다. 조사 대상 시스템으로는 Cassandra, HBase, Hadoop 분산 파일 시스템(HDFS), Hadoop MapReduce, Redis가 선정되었습니다.

우리의 목표는 이러한 시스템에서 발생한 특정 장애 발현 순서를 더 잘 이해하여 가용성과 복원력을 개선할 수 있는 기회를 파악하는 것입니다. 특히, 하나 또는 여러 개의 오류([1])가 어떻게 구성 요소 오류로 발전하는지, 그리고 그 중 일부가 결국 서비스 전체의 치명적인 오류로 발전하는지를 이해하고자 합니다. 이전에는 근본 원인 분류[33, 52, 50, 56]를 포함하여 장애 시퀀스의 개별 요소를 개별적으로 연구해 왔으며, 다음과 같은 다양한 유형의 원인이 있습니다.
구성 오류 [43, 66, 49], 버그 [12, 41, 42, 51] 발생 하드웨어 결함[62], 장애 증상[33, 56] 등에 대해 연구했으며, 이러한 연구 중 상당수는 많은 버그를 식별할 수 있는 도구로 이어졌다는 점에서 큰 영향을 미쳤습니다(예: [16, 39]). 그러나 이들을 연결하는 전체 발현 시퀀스에 대해서는 잘 알려져 있지 않습니다.

고려한 각 장애에 대해 장애 보고서, 사용자와 개발자 간의 토론, 로그 및 코드를 면밀히 검토하고 73건의 장애를 수동으로 재현하여 발생한 특정 장애를 더 잘 파악할 수 있었습니다.

전반적으로, 우리는 오류 발현 시퀀스가 비교적 복잡한 경향이 있다는 것을 발견했습니다. 대부분의 경우, 시스템을 장애로 이끌기 위해 넓은 공간에서 특정 입력 매개변수가 있는 비정상적인 여러 이벤트의 시퀀스가 필요합니다. 이러한 시스템은 단위 테스트, 무작위 오류 주입[18], FindBugs[32]와 같은 정적 버그 찾기 도구를 사용하여 철저한 테스트를 거쳤으며 많은 조직에서 널리 배포되고 지속적으로 사용된다는 점을 고려하면 이는 놀라운 일이 아닙니다. 하지만 하향식 테스트는 다음과 같은 문제점이 있습니다.

[1]이 백서에서는 다음과 같은 표준 용어를 사용합니다[36]. *결함은* 하드웨어 오작동, 소프트웨어 *버그* 또는 잘못된 구성이 될 수 있는 초기 근본 원인입니다. 결함은 시스템 호출 오류 회귀 또는 Java 예외와 같은 *오류라고 하는* 비정상적인 동작을 생성할 수 있습니다. 일부 오류는 사용자가 볼 수 있는 부작용이 없거나 소프트웨어에서 적절히 처리될 수 있지만, 다른 오류는 최종 사용자나 운영자가 시스템 오작동을 알아차리는 *장애로* 나타납니다.

입력 및 오류 주입 기법을 사용하는 경우, 입력 및 상태 공간이 커서 시간이 오래 걸립니다. 이러한 이유로 연구된 실패 사례들이 소프트웨어 프로젝트에서 사용되는 엄격한 테스트를 피한 것일 수 있습니다.

또한 특정 장애의 하위 집합, 즉 일부 사용자만이 아닌 전체 또는 대다수의 사용자에게 영향을 미치는 치명적인 장애의 특성을 연구했습니다. 치명적인 장애는 공급업체에게 가장 많은 비용이 드는 장애이며, 이러한 분산 시스템은 구성 요소 장애를 견디고 자동으로 복구하도록 설계되었기 때문에 발생해서는 안 되는 장애이기 때문에 특히 관심을 끄는 장애입니다. 구체적으로 살펴보면 다음과 같습니다:

치명적인 시스템 장애의 거의 대부분(92%)은 소프트웨어에서 명시적으로 신호를 보내는 치명적이지 않은 오류를 잘못 처리한 결과입니다.

오류 처리 코드에 버그가 많다는 것은 잘 알려져 있지만 [24, 44, 55], 치명적인 장애의 원인에 오류가 많다는 사실은 여전히 놀랍습니다. 오류 처리 코드가 장애를 막는 최후의 방어선이라는 점을 고려하면 더욱 놀라운 사실입니다:

치명적인 장애의 58%에서 오류 처리 코드에 대한 간단한 테스트를 통해 근본적인 결함을 쉽게 발견할 수 있었습니다.

실제로 치명적인 장애의 35%에서 오류 처리 코드의 결함은 세 가지 사소한 패턴에 속합니다:
(i) 오류 처리기가 단순히 비어 있거나 로그 인쇄 문만 포함된 경우, (ii) 오류 처리기가 지나치게 일반적인 예외에서 클러스터를 중단하는 경우, (iii) 오류 처리기가 주석에 "FIXME" 또는 "TODO"와 같은 표현을 포함하는 경우입니다. 이러한 결함은 런타임 컨텍스트에 대한 깊은 이해 없이도 도구나 코드 리뷰에서 쉽게 발견할 수 있습니다. 치명적인 오류의 또 다른 23%는 치명적이지 않은 오류의 오류 처리 로직이 너무 잘못되어 있어 개발자가 문 커버리지 테스트나 보다 세심한 코드 리뷰를 수행했다면 버그를 발견할 수 있었을 것입니다.

치명적인 장애를 초래한 버그에서 추출한 간단한 규칙의 적용 가능성을 측정했습니다.
간단한 정적 검사기인 Aspirator를 구현했습니다. 이러한 시스템에서 이미 FindBugs[32] 및 오류 주입 도구[18]와 같은 최첨단 버그 찾기 도구를 사용하고 있음에도 불구하고 Aspirator는 널리 사용되는 9개의 프로덕션 품질 분산 시스템에서 121개의 *새로운* 버그와 379개의 잘못된 관행을 식별했습니다. 이 중 143개는 시스템 개발자에 의해 수정되었거나 확인되었습니다.

이 연구에는 테스트 및 버그 제거 전략을 개선하는 데 도움이 될 수 있는 여러 가지 추가 관찰 결과도 포함되어 있습니다. 74%의 장애가 적절한 입력 시퀀스로 나타나도록 보장된다는 점에서 결정론적이며, 거의 모든 장애가 3개 이하의 노드에서 나타나도록 보장되고, 77%의 장애가 단위 테스트로 재현될 수 있다는 것을 발견했습니다.

소프트웨어 랭. 실패

		합계	샘플링된	치명적
카산드라	Java	3,923	40	2
HBase	Java	5,804	41	21
HDFS	Java	2,828	41	9
MapReduce	Java	3,469	38	8
Redis	C	1,192	38	8
합계		17,216	198	48

표 1: 연구 대상 시스템에서 보고 및 표본 추출된 장애 건수 및 샘플 세트의 치명적인 장애 건수.

또한 장애의 76%에서 시스템이 명시적인 장애 메시지를 전송하고, 장애의 84%에서 장애를 일으킨 모든 트리거링 이벤트가 장애 발생 전에 로그에 인쇄됩니다. 이 모든 것은 장애를 합리적이고 간단한 방식으로 진단하고 재현할 수 있음을 나타내며, 가장 큰 문제는 상대적으로 노이즈가 많은 로그를 선별해야 한다는 것입니다.

2 방법론 및 제한 사항

분산 파일 시스템인 HDFS[27], 분산 데이터 분석 프레임워크인 Hadoop MapReduce[28], 두 개의 NoSQL 분산 데이터베이스인 HBase와 Cassandra[2, 3], 마스터/슬레이브 복제를 지원하는 메모리 내 키-값 저장소인 Redis[54] 등 5개의 유명한 분산 데이터 분석 및 스토리지 시스템에서 보고된 198건의 실제 장애를 무작위로 샘플링하여 연구했습니다. 분산 데이터 집약적 시스템에 초점을 맞춘 이유는 이러한 시스템이 많은 네트워크 소프트웨어 서비스의 구성 요소이기 때문이며, 널리 사용되고 생산 품질이 고려된 5가지 시스템을 선택했습니다.

우리가 연구한 실패는 문제에서 추출되었습니다. 이러한 시스템의 데이터베이스를 추적합니다. 이러한 데이터베이스의 티켓은 개발자가 확인한 뚜렷한 장애, 사용자와 개발자 간의 논의, 패치 또는 구성 변경의 형태로 장애를 해결한 내용을 문서화하는 등 품질이 높았기 때문에 선택했습니다. 중복 장애는 개발자가 표시한 것으로 연구 대상에서 제외되었습니다.

고려한 특정 장애 세트는 다음과 같이 문제 추적 데이터베이스로부터 추출했습니다. 첫째, 장애 티켓 *우선순위 필드가* '차단', '중요' 또는 '심각'으로 표시된 심각한 장애만 선택했습니다. 둘째, 구형 시스템이나 수명 초기의 시스템 장애가 포함되지 않도록 2010년 이후에 발생한 티켓만 고려했습니다. 셋째, 보고자와 담당자(즉, 장애를 해결하도록 배정된 개발자)가 동일한 장애는 휴리스틱적으로 거부하여 테스트 시스템의 장애를 걸러냈습니다. 마지막으로, 나머지 집합에서 장애를 무작위로 선택하여 전체 장애 집단을 대표할 수 있도록 관찰했습니다. 표 1은

다섯 가지 시스템 중 고려되는 실패 세트와 샘플링 속도입니다.

샘플링된 각 장애 티켓에 대해 장애 보고서, 사용자와 개발자 간의 토론, 관련 오류 로그, 소스 코드, 패치를 면밀히 검토하여 장애를 일으킨 근본 원인과 그 전파 경로를 파악했습니다. 또한 장애를 더 잘 이해하기 위해 73건의 장애를 수동으로 재현했습니다.

한계: 모든 특성화 연구와 마찬가지로, 연구 결과가 대표성을 갖지 못할 수 있는 내재적 위험이 존재합니다. 다음에서는 편향의 잠재적 원인을 나열하고 이를 보정하기 위해 최선을 다한 방법을 설명합니다.

(1) 선택한 시스템의 대표성. 저희는 데이터 집약적인 분산형 소프트웨어 시스템만을 연구했습니다. 따라서 통신 네트워크나 과학 컴퓨팅 시스템과 같은 다른 유형의 분산 시스템에는 일반화되지 않을 수 있습니다. 하지만 데이터 저장 및 분석 시스템, 영구 저장 및 휘발성 캐싱, Java와 C로 작성된 마스터-슬레이브 및 P2P 설계를 모두 포함하는 다양한 유형의 데이터 집약적 프로그램을 선택하기 위해 주의를 기울였습니다. (HBase, HDFS, Hadoop MapReduce, Redis는 마스터-슬레이브 설계를 사용하는 반면, Cassandra는 피어 투 피어 가십 프로토콜을 사용합니다.) 최소한 이러한 프로젝트는 널리 사용되고 있습니다: HDFS와 Hadoop MapReduce는 가장 널리 사용되는 빅데이터 분석 솔루션인 Hadoop 플랫폼의 주요 요소이며[29], HBase와 Cassandra는 가장 널리 사용되는 두 개의 와이드 컬럼 저장 시스템이고[30], Redis는 가장 널리 사용되는 키-값 저장 시스템입니다[53].

또한 생산 품질을 고려한 시스템만 연구했기 때문에 개발 주기가 더 이른 시스템에는 일반화하지 못할 수도 있습니다. 그러나 초기 시스템의 버그를 피하기 위해 2010년 이후의 티켓만 고려했지만, 버그 코드가 새로 추가되었을 수도 있습니다. 이러한 시스템의 진화를 연구하여 버그와 코드의 연식 간의 상관관계를 밝히는 것은 향후 작업으로 남아 있습니다.

(2) 선택한 실패의 대표성. 편향의 또 다른 잠재적 원인은 선택한 특정 실패 집합입니다. 저희는 소프트 웨어 버그를 문서화하기 위한 이슈 추적 데이터베이스에서 발견된 티켓만 연구했습니다. 설정 오류와 같은 다른 오류는 사용자 토론 포럼에서 보고될 가능성이 더 높지만, 이러한 오류는 엄격하게 문서화되어 있지 않고 권위 있는 판단이 부족하며 사소한 실수로 인해 발생하는 경우가 많기 때문에 연구하지 않기로 결정했습니다. 결론적으로, 본 연구에서는 결함의 분포에 대한 결론을 내리지 않았는데, 이는 선행 연구에서 잘 연구되어 왔습니다[50, 52]. 그러나 사용자가 오류 원인의 특성을 정확하게 식별하기 어려울 수 있으므로 본 연구에는 다음과 같은 오류도 포함됩니다.

증상	모두	치명적
예기치 않은 종료	74	17 (23%)
잘못된 결과	44	1 (2%)
데이터 손실 또는 잠재적 데이터 손실*	40	19 (48%)
행 시스템	23	9 (39%)
심각한 성능 저하	12	2 (17%)
리소스 누수/고갈	5	0 (0%)
합계	198	48 (24%)

표 2: 최종 사용자 또는 운영자가 관찰한 장애의 증상. 가장 오른쪽 열에는 치명적인 장애의 수가 표시되어 있으며, "%"는 해당 증상이 있는 모든 장애에 대한 치명적인 장애의 비율을 나타냅니다. *참고: 잠재적인 데이터 손실의 예로는 복제되지 않은 데이터 블록이 있습니다.

잘못된 구성과 하드웨어 결함에서 비롯됩니다.

또한 고유한 버그의 특성을 반영하기 위해 중복된 버그는 연구에서 제외했습니다. 중복된 버그는 10개 이상 발생했기 때문에 제거해서는 안 된다고 주장할 수도 있습니다. 원래 샘플 세트에서 제외된 중복 버그는 총 10개에 불과했습니다. 따라서 중복 버그를 포함하더라도 결론이 크게 달라지지는 않을 것입니다.

(3) 샘플 세트의 크기. 현대 통계학에 따르면 30개 이상의 무작위 표본 세트는 전체 모집단을 대표할 수 있을 만큼 충분히 큰 것으로 알려져 있습니다[57]. 좀 더 엄밀히 말하면, 표준 가정 하에서 중앙 한계 정리는 198개의 무작위 표본에 대해 95% 신뢰 수준에서 6.9%의 오차 마진을 예측합니다. 물론 더 많은 샘플을 연구하여 오차 범위를 더 줄일 수 있습니다.

(4) 관찰자 오류 가능성. 연구의 질적 측면에서 관찰자 오류 가능성을 최소화하기 위해 모든 조사자는 동일한 세부적인 서면 분류 방법론을 사용했으며, 모든 실패는 합의에 도달하기 전에 두 명의 조사자가 개별적으로 조사했습니다.

3 일반 결과

이 섹션에서는 장애가 어떻게 나타나는지에 대한 이해를 돕기 위해 전체 장애 데이터 세트에서 얻은 일반적인 결과에 대해 설명합니다. 표 2는 우리가 연구한 장애의 증상을 분류한 것입니다.

전반적으로 조사 결과, 장애는 비교적 복잡하지만 테스트를 개선할 수 있는 여러 가지 기회를 발견할 수 있었습니다. 또한 이러한 시스템에서 생성된 로그에는 풍부한 정보가 포함되어 있어 장애를 진단하는 것이 대부분 간단하다는 것을 보여줍니다. 마지막으로, 일반적으로 장기간 운영되는 대규모 프로덕션 클러스터에서 장애가 발생하더라도 오프라인에서 비교적 쉽게 장애를 다시 생성할 수 있음을 보여줍니다. 구체적으로, 대부분의 장애를 재현하는 데 3개 이하의 노드와 3개 이하의 입력 이벤트가 필요하며, 대부분의 장애는 결정론적이라는 것을 보여줍니다. 실제로 대부분의 장애는 단위 테스트로 재현할 수 있습니다.

이벤트 수	%	
1	23%	단일 이벤트
2	50%	
3	17%	multiple 이벤트: 77%
4	5%	
> 4	5%	

표 3: 장애를 트리거하는 데 필요한 최소 입력 이벤트 수입니다.

입력 이벤트 유형	%
서비스 시작하기	58%
클라이언트에서 파일/데이터베이스 쓰기	32%
연결할 수 없는 노드(네트워크 오류, 충돌 등)	24%
구성 변경	23%
실행 중인 시스템에 노드 추가하기	15%
클라이언트에서 파일/데이터베이스 읽기	13%
노드 재시작(의도적)	9%
데이터 손상	3%

표 4: 실패로 이어진 입력 이벤트. 열은 실패를 트리거하는 데 입력 이벤트가 필요한 실패의 비율을 다시 나타냅니다. 대부분의 실패에는 여러 선행 이벤트가 필요하므로 "%" 열의 합계는 100%보다 큽니다.

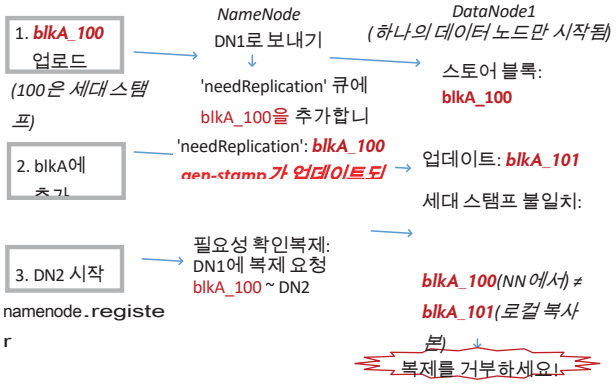


그림 1: 데이터 블록이 남아 있는 HDFS 장애 상황

3.1 장애의 복잡성

전반적으로 조사 결과 장애의 양상은 비교적 복잡한 것으로 나타났습니다.

찾기 1 *대다수(77%)의 실패는 두 개 이상의 입력 이벤트가 필요하지만 대부분의 실패(90%)는 3개 이하의 입력 이벤트가 필요합니다.* (표 3 참조).

그림 1은 실패가 나타나기 위해 세 가지 입력 이벤트가 필요한 예시를 보여줍니다.

표 4는 장애를 유발하는 입력 이벤트를 9가지 범주로 분류합니다. 테스트 및 진단 관점에서 이러한 이벤트를 "입력 이벤트"로 간주합니다. 일부 이벤트(예: "연결할 수 없는 노드", "데이터 파열")는 엄밀히 말하면 사용자 입력이 아니지만 테스터 또는 테스트 도구로 쉽게 에뮬레이션할 수 있습니다. 많은 이벤트에는 장애 발생에 대한 특정 요구 사항(예: "파일 쓰기" 이벤트는 특정 데이터 블록에서 발생해야 함)이 있으므로 테스트를 위해 탐색해야 할 입력 이벤트 공간이 엄청나게 커집니다.

단 하나의 이벤트만 발생해야 하는 장애의 23%는 거의 사용되지 않거나 새로 도입된 기능과 관련된 것이거나 동시성 버그로 인해 발생하는 경우가 많습니다.

찾기 2 *여러 입력 이벤트가 필요한 장애의 88%에서 이벤트의 특정 순서가 중요합니다.*

표 4의 개별 이벤트는 대부분 읽기 및 쓰기와 같이 실

가 시작되므로 NN은 DN1에 블록을 DN2에 복제하도록 요청합니다. 그러나 필요 복제 대기열과 DN1의 생성 스탬프가 일치하지 않기 때문에 DN1은 계속 복제를 거부합니다.

이벤트가 장애를 유도합니다. 대부분의 경우, 시스템을 장애 상태로 전환하려면 여러 이벤트의 특정 조합과 순서가 필요합니다. 그림 1에 표시된 장애 예시를 살펴보겠습니다. "파일 업로드", "파일에 추가", "다른 데이터노드 추가" 이벤트는 개별적으로는 문제가 되지 않지만, 처음 두 이벤트의 조합으로 인해 시스템이 오류 상태가 되고 마지막 이벤트가 실제로 장애를 트리거합니다.

결과 1과 2는 대규모 분산 시스템에서 장애의 복잡성을 보여줍니다. 테스트에서 실패를 노출하려면 매우 큰 이벤트 공간에서 여러 입력 이벤트의 *조합을* 탐색해야 할 뿐만 아니라 다양한 *순열을* 탐색해야 합니다.

3.2 테스트 개선의 기회

실패가 나타나는 데 필요한 입력 이벤트 유형을 고려할 때 기존 테스트 전략을 개선할 수 있는 추가적인 기회를 찾을 수 있습니다. 표 4의 입력 이벤트 유형 중 몇 가지에 대해 간략히 설명합니다.

서비스 시작: 장애의 절반 이상이 일부 서비스의 시작을 다시 요구합니다. 이는 서비스 시작, 특히 잘 알려지지 않은 서비스의 시작을 더욱 철저하게 테스트해야 함을 시사합니다. 서비스 시작으로 인해 트리거된 장애의 약 4분의 1은 오랫동안 실행되어 온 시스템에서 발생했습니다. 예를 들어, HBase "Region Split" 서비스의 경우, TB가 임계값보다 커질 때만 시작됩니다. 이러한 장애는 시스템을 오래 실행해야 하기 때문에 테스트하기 어려울 수 있지만, 테스트 중에 서비스를 강제로 시작하면 의도적으로 노출될 수 있습니다.

숫자	누적 분포 함수	
노드 수	모든 실패	치명적
1	37%	43%
2	84%	86%
3	98%	98%
> 3	100%	100%

표 5: 장애를 트리거하는 데 필요한 최소 노드 수.

소프트웨어	결정적 실패 횟수
카산드라	76% (31/41)
HBase	71% (29/41)
HDFS	76% (31/41)
MapReduce	63% (24/38)
Redis	79% (30/38)
합계	74% (147/198)

Table 6: 결정론적 실패 횟수 c.

비결정론의 원천	숫자
입력 이벤트와 내부 실행 이벤트 사이의 타이밍	27 (53%)
멀티 스레드 원자성 위반	13 (25%)
멀티 스레드 교착 상태	3 (6%)
멀티 스레드 잠금 경합(성능)	4 (8%)
기타	4 (8%)

연결할 수 없는 노드: 장애의 24%는 다음과 같은 이유로 발생합니다. 노드에 연결할 수 없습니다. 대규모 데이터 센터에서 네트워크 오류와 개별 노드 충돌이 정기적으로 발생한다는 점을 고려하면 다소 의외의 결과입니다[14]. 이는 테스트 중 다른 이벤트를 입력할 때 네트워크 오류를 체계적으로 주입할 수 있는 도구[18, 23, 65]를 더 광범위하게 사용해야 한다는 것을 시사합니다.

구성 변경: 장애의 23%가 구성 변경으로 인해 발생합니다. 구성 변경에 의한 것입니다. 이 중 30%는 잘못된 구성과 관련이 있습니다. 나머지 대부분은 거의 사용되지 않을 수 있는 특정 기능을 활성화하기 위한 유효한 변경과 관련이 있습니다. 이전 연구[22, 50, 66]에서 잘못된 구성의 중요성이 관찰되었지만, 구성 변경을 자동으로 탐색하고 그에 따른 시스템 반응을 테스트하는 기술은 몇 가지에 불과합니다[19, 40, 63]. 이는 유효한 구성 변경과 유효하지 않은 구성 변경을 다른 작업과 결합할 수 있도록 테스트 도구를 확장해야 함을 시사합니다.

노드 추가: 장애의 15%는 실행 중인 시스템에 노드를 추가함으로써 트리거됩니다. 그림 1은 그 예를 보여줍니다. 노드를 쉽게 추가하고 제거하는 것이 '클라우드 컴퓨팅'의 주요 약속 중 하나라는 점을 고려하면 다소 놀라운 결과입니다. 이는 노드 추가가 더 많은 시나리오에서 테스트되어야 함을 시사합니다.

우리가 연구한 프로덕션 장애는 일반적으로 노드 수가 많은 구성에서 발생했습니다. 이는 효과적인 테스트 및 디버깅 시스템을 위해 얼마나 많은 노드가 필요한지에 대한 의문을 제기합니다.

찾기 3 *거의 모든(98%) 장애는 3개 이하의 노드에서 나타나도록 보장됩니다. 84%는 2개 이하의 노드에서 발생합니다.* (표 5 참조)

이 수치는 치명적인 장애의 경우에도 비슷합니다. 3을 찾으면 장애를 테스트하고 재현하기 위해 대규모 클러스터가 필요하지 않다는 것을 의미합니다.

결과 3은 분산 시스템 장애가 대규모 클러스터에서 나타날 가능성이 더 높다는 일반적인 통념과 모순되지 *않는다는* 점에 유의하세요. 결국 테스트는 확률적인 작업입니다. 대규모 클러스터는 일반적으로 더 다양한 워크로드와 장애 모드를 포함하므로 장애가 나타날 가능성이 높아집니다. 그러나 이번 연구 결과에서 알 수 있는 것은 입력 이벤트의 특정 시퀀스가 발생하기만 하면 버그를 노출하기 위해 대규모의 머신 클러스터가 필요하지 않다는 것입니다.

더 큰 규모의 장애가 발생한 경우는 단 한 번뿐이었습니다.

노드 수(1024개 이상): 시뮬레이션 노드 수가 1024개를 초과하는 경

표 7: 비결정적 실패에 대한 분석. 비결정적 실패 '기타' 범주는 OS 및 타사 라이브러리의 비결정적 동작으로 인해 발생합니다.

일시적인 Redis 클라이언트 연결이 OS 제한을 초과한 경우, `epoll()`이 제대로 처리되지 않은 오류를 반환하여 전체 클러스터가 중단되는 원인이 되었습니다. 다른 모든 장애는 10개 미만의 노드에서 발생했습니다.

3.3 타이밍의 역할

테스트 및 진단의 핵심 질문은 필요한 입력 이벤트 시퀀스가 발생할 경우(즉, *결정론적 장애*) 장애가 보장되는지, 아니면 그렇지 않은지(즉, *비결정론적 장애*) 여부입니다.

오류의 74%는 결정론적 오류로, 올바른 입력 이벤트 시퀀스가 주어지면 반드시 발생한다는 것을 알 수 있습니다. (표 6 참조)

즉, 대부분의 실패에 대해 인풋 이벤트의 조합과 순열만 탐색하면 되고 추가적인 타이밍 관계는 필요하지 않습니다. 이는 특히 장기간 실행되는 시스템에서 다시 나타나는 장애를 테스트할 때 유용합니다. 일반적으로 장기 실행 시스템에서만 발생하는 이벤트를 시뮬레이션할 수 있는 한(예: HBase에서 영역 분할은 일반적으로 영역 크기가 너무 커질 때만 발생) 이러한 결정론적 오류를 노출할 수 있습니다. 또한, 추가 로그 출력을 삽입하거나 추적을 활성화하거나 디버거를 사용한 후에도 장애를 재현할 수 있습니다.

결과 5 *51개의 비결정적 실패 중 53%는 입력 이벤트에만 타이밍 제약 조건이 있습니다.* (표 7 참조).

이러한 제약 조건은 프로시저 호출과 같은 소프트웨어 내부 실행 이벤트 전후에 입력 이벤트가 발생하도록 요구합니다. 그림 2는 그 예를 보여줍니다. 네 가지 입력 이벤트의 순서(

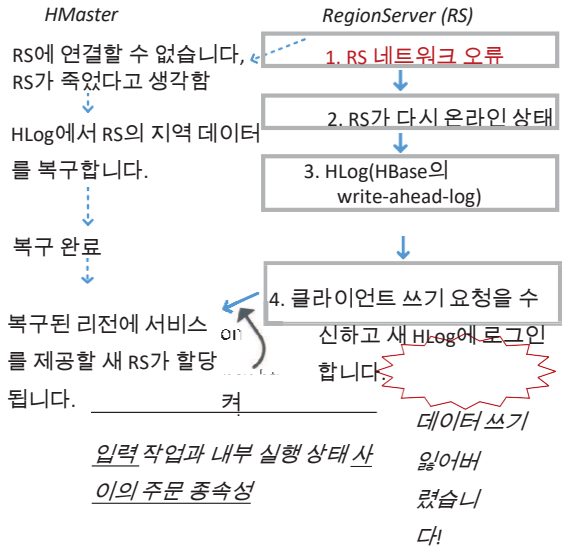


그림 2: 입력 이벤트에 대해서만 타이밍 요구 사항(실선 화살표로 표시)이 있는 HBase의 비결정적 장애(박스형). HMaster가 새 리전 서버를 할당할 때 새로 작성된 데이터가 포함되지 않은 이전 HLog만 복구하기 때문에 일부 새로 작성된 데이터가 손실됩니다.

```
쓰기_잠금();
/* 제거
 * 대형 디렉토리 */
쓰기_잠금해제();
```

중요 영역이 너무 커서 동시 쓰기 요청이 중단됩니다.

그림 3: 한 번의 대용량 디렉터리 제거 요청으로 인한 HDFS의 성능 저하.

테스터에 의해 제어됨), 추가 요구 사항은 HMaster가 새 리전 서버에 리전을 할당하기 전에 클라이언트 쓰기 작업이 발생해야 하며, 이는 사용자가 완전히 제어할 수 없다는 것입니다.

이러한 비결정적 종속성은 타이밍 종속성의 적어도 한 부분을 테스터가 제어할 수 있기 때문에 멀티스레드 인터리빙에서 비롯된 비결정성보다 테스트 및 디버깅이 더 쉽습니다. 테스터는 입력 이벤트의 타이밍을 신중하게 제어하여 오류를 유도할 수 있습니다. 단위 테스트와 모델 검사 도구는 입력 이벤트와 내부 프로시저 호출의 타이밍을 모두 제어하여 이러한 타이밍 종속성을 더욱 완벽하게 조작할 수 있습니다. 예를 들어, 그림 2의 버그를 수정하기 위한 패치의 일부로 개발자는 사용자 *입력과* HMaster의 연산에 대한 종속성을 시뮬레이션하는 단위 테스트를 사용하여 장애를 종결적으로 재현했습니다.

나머지 24건의 비결정적 오류의 대부분은 공유 메모리 다중 스레드 간 이탈에서 비롯되었습니다. 저희는 데이터 세트에서 원자성 위반[42], 교착 상태, 성능 저하를 초래하는 잠금 경합의 세 가지 범주의 동시성 버그를 관찰했습니다. 이러한 장애는 사용자나 도구가 타이밍을 제어하기 어렵고, 단일 로깅 문을 추가하면 장애가 더 이상 노출되지 않을 수 있기 때문에 이를 노출하고 재현하기가 훨씬 더 어렵습니다. 이러한 비결정론적 장애 중 10개를 재현한 결과 원자성 위반과 교착 상태가 가장 재현하기 어려웠습니다(수동으로 추가

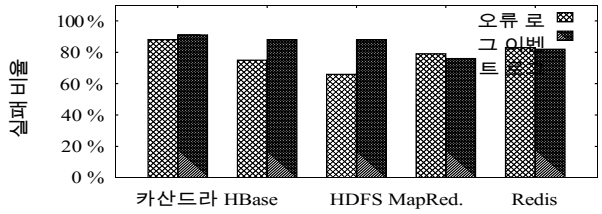


그림 4: 연구된 장애의 로깅 포괄성. 입력 이벤트와 오류의 로깅이 모두 고려됩니다. 여러 이벤트가 트리거되어야 하는 장애의 경우 다음을 계산합니다. 필수 이벤트가 *모두* 기록된 경우에만 "기록됨"으로 설정합니다.

처럼 코드에서 `Thread.sleep()`을 사용하여 버그를 트리거할 수 있습니다.)

그러나 잠금 경합 사례는 재현하기 어렵지 않습니다. 그림 3은 버그로 인해 불필요한 잠금 경합이 발생한 예를 보여줍니다.

3.4 진단 기회를 제공하는 로그

전반적으로 연구한 시스템에서 출력되는 로그는 유용한 정보가 풍부한 것으로 나타났으며, 기본 로깅 상세도 수준을 사용한다고 가정했습니다.

타이밍 지연을 도입해야 했습니다),

장애의 76%는 명시적인 장애 관련 오류 메시지를 출력합니다(그림 4 참조).

이 결과는 Apache httpd, PostgreSQL, SVN, squid, GNU Coreutils 등 비분산형 시스템의 장애에 대한 이전 연구[67]에서 장애의 43%만이 명시적인 장애 관련 오류 메시지를 기록했다는 결과와 다소 상반됩니다. 저희가 연구한 분산형 시스템에서 개발자가 로그 메시지를 더 광범위하게 출력하는 이유는 세 가지로 추정됩니다. 첫째, 분산 시스템은 더 복잡하고 디버깅하기 어렵기 때문에 개발자가 로깅에 더 많은 주의를 기울일 가능성이 높습니다. 둘째, 이러한 시스템의 수평적 확장성으로 인해 로그 메시지 출력의 성능 오버헤드가 덜 중요해집니다. 셋째, 메시지 전달을 통한 통신은 로그 메시지에 자연스러운 포인트를 제공합니다. 예를 들어 두 노드가 네트워크 문제로 인해 서로 통신할 수 없는 경우, 두 노드 모두 오류를 기록할 수 있는 기회를 갖게 됩니다.

결과 7 *장애의 대부분(84%)에 대해 모든 트리거 이벤트가 기록됩니다* (그림 4 참조).

이는 *기존 로그 메시지만으로* 대부분의 장애를 결정론적으로 다시 재생할 수 있음을 시사합니다. 결정론적 재생은 연구 커뮤니티에서 널리 연구되어 왔습니다[4, 13, 15, 26, 35, 47, 61]. 그러나 이러한 접근 방식은 런타임 오버헤드가 크고 소프트웨어/하드웨어를 수정해야 하는 인트루시브 트레이싱을 기반으로 합니다.

찾기 8 *로그가 노이즈가 많음: 각 장애로 인해 인쇄되는 로그 메시지 수의 중앙값은 824개입니다.*

소프트웨어	단위 테스트에서 재현 가능한 장애의 비율
카산드라	73% (29/40)
HBase	85% (35/41)
HDFS	82% (34/41)
MapReduce	87% (33/38)
Redis	58% (22/38)
합계	77% (153/198)

표 8: 단위 테스트에서 재현할 수 있는 실패의 비율. 단위 테스트로 재현할 수 있는 Redis 장애의 수가 상대적으로 적은 이유는 단위 테스트 프레임워크가 명령줄 명령으로 제한되어 있어 강력하지 않기 때문입니다. 따라서 노드 장애와 같은 많은 오류를 시뮬레이션할 수 없으며 일부 내부 함수를 직접 호출할 수도 없습니다.

이 수치는 198건의 장애 중 73건의 장애를 최소한의 구성과 장애를 재현하기에 충분한 최소한의 워크로드를 사용하여 재현했을 때 얻은 결과입니다. 또한 시작 및 종료 단계에서 출력된 메시지는 포함하지 않았습니다.

이는 로그 파일을 수동으로 검사하는 것이 지루할 수 있음을 시사합니다. 사용자가 오류 증상에만 관심이 있는 경우, 인쇄된 로그 메시지의 대부분이 INFO 수준에 있으므로 오류 상세도 수준에서 선택적으로 `grep`을 수행하면 노이즈를 줄일 수 있습니다. 그러나 장애를 유발한 입력 이벤트는 종종 INFO 레벨에 기록됩니다. 따라서 입력 이벤트를 추가로 추론하려면 거의 모든 로그 메시지를 분석해야 합니다. 기존의 로그 분석 기법[5, 6, 48, 64]과 도구를 확장하여 관련 없는 로그 메시지를 필터링하여 관련 오류 및 입력 이벤트 메시지를 추론할 수 있다면 도움이 될 것입니다.

3.5 장애 재현성

일반적으로 대규모의 분산된 프로덕션 시스템에서 발생하는 장애는 오프라인에서 재현하기가 매우 어렵다고 알려져 있습니다. 개인정보 보호 문제로 인해 사용자의 입력을 사용할 수 없는 경우, 프로덕션 환경과 동일한 환경을 설정하기 어려운 경우, 타사 라이브러리 비용 등이 공급업체가 프로덕션 장애를 재현하기 어려운 이유로 자주 언급됩니다. 아래 조사 결과에 따르면 장애 재현이 생각만큼 어렵지 않을 수 있습니다.

발견 9 *생산 실패의 대부분(77%)은 단위 테스트를 통해 재현할 수 있습니다.* (표 8 참조)

이 결과는 직관적이지 않게 들릴 수 있지만, (1) 결과 4에서 74%의 장애가 결정적 장애로, 동일한 작업 순서로 장애를 재현할 수 있고, (2) 나머지 비결정적 장애 중 53%의 경우 단위 테스트를 통해 타이밍을 제어할 수 있기 때문에 이전 결과를 고려하면 놀랍지 않은 결과입니다.

일반적으로 장애를 다시 생성하는 데 특정 데이터 값이 필요하지 않으며, 실제로 연구된 장애 중 *어떤* 장애도 발생하지 않았습니다.

```
public void
  testLogRollAfterSplitStart {
  startMiniCluster(3);
   // 마스터 1 개와 RS 2 개로 HBase 클러스터를 생성합니다.
  HMaster.splitHLog();
   // 로그 분할 시뮬레이션 (HMaster 의 복구
   // RS 의 지역 데이터의) RS 에 연결할 수 없는 경우
  RS.rollHLog();
   // 지역 서버의 로그 롤링 이벤트 시뮬레이션
  for (i = 0; i < NUM_WRITES; i++)
    writer.append(..); // RS 의 영역에 쓰기
  HMaster.assignNewRS();
   // HMaster 가 새 RS 에 리전을 할당합니다.
  assertEquals (NUM_WRITES, countWritesHLog());
   // 손실된 쓰기가 있는지 확인
}
```

그림 6: 그림 2에 표시된 장애에 대한 단위 테스트.

사용자 데이터 콘텐츠의 특정 값을 요구하지 않습니다. 대신 필요한 입력 시퀀스(예: 파일 쓰기, 노드 연결 해제 등)만 있으면 됩니다.

그림 6은 단위 테스트가 그림 2의 비결정적 장애를 시뮬레이션하는 방법을 보여줍니다. 이 테스트는 3개의 노드로 실행되는 3개의 프로세스를 시작하여 미니 클러스터를 시뮬레이션합니다. 또한 HMaster의 로그 분할, 리전 서버의 로그 롤링, 쓰기 요청을 포함한 주요 입력 이벤트를 시뮬레이션합니다. 마스터가 복구된 리전을 재할당하기 전에 클라이언트가 쓰기 요청을 보내야 하는 필수 종속성도 이 단위 테스트에서 제어합니다.

쉽게 재현할 수 없는 장애는 특정 실행 환경(예: OS 버전 또는 타사 라이브러리)에 의존하거나 비결정적 스레드 인터리빙으로 인해 발생한 것입니다.

4 치명적인 실패

섹션 3의 표 2는 전체 장애 세트에서 48개의 장애가 치명적인 결과를 초래한다는 것을 보여줍니다. 장애로 인해 *전체 또는 대다수의 사용자가* 시스템에 정상적으로 액세스하지 못하는 경우 치명적인 장애로 분류합니다. 실제로 이러한 장애는 클러스터 전체가 중단되거나, 클러스터가 중단되거나, 사용자 데이터의 전부 또는 대다수가 손실되는 결과를 초래합니다. 복제되지 않은 데이터 블록을 초래하는 버그는 모든 데이터 블록에 영향을 미치더라도 사용자의 정상적인 데이터 읽기 및 쓰기를 아직 방해하지 않기 때문에 치명적인 장애로 간주되지 *않습니다*. 특히 치명적인 장애는 공급업체에 비즈니스에 가장 큰 영향을 미치는 장애이기 때문에 특별히 연구합니다.

고려 대상 시스템 모두 구성 요소 장애로 인해 전체 서비스가 중단되는 것을 방지하도록 설계된 고가용성(HA) 메커니즘을 갖추고 있다는 점을 고려하면 치명적인 장애가 그렇게 많이 발생했다는 사실은 놀라운 일이 아닐 수 없습니다. 예를 들어, 마스터-슬레이브 설계가 적용된 네 가지 시스템, 즉 HBase, HDFS, MapRe-duce, Redis는 모두 마스터 노드 장애 시 자동으로 새 마스터 노드를 선택하고 장애를 처리하도록 설계되었습니다.

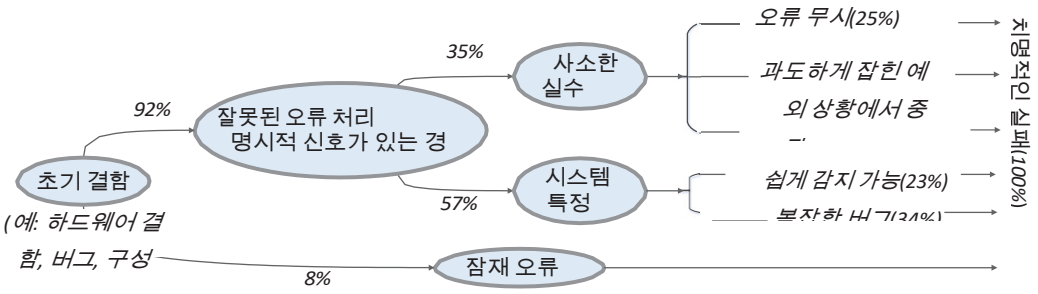


그림 5: 모든 치명적인 장애를 오류 처리별로 분류한 결과입니다.

로 이동합니다.[2] Cassandra는 P2P 시스템이기 때문에 설계상 단일 장애 지점을 피할 수 있습니다. 그렇다면 왜 여전히 치명적인 장애가 발생하나요?

발견 10 *거의 모든 치명적인 장애(92%)는 소프트웨어에서 명시적으로 신호를 보내는 치명적이지 않은 오류를 잘못 처리한 결과입니다.* (그림 5 참조)

이러한 치명적인 장애는 하드웨어 결함, 구성 오류, 버그 등으로 인한 초기 결함이 예외를 발생시키거나 시스템 호출이 오류를 반환하는 등 치명적이지 않은 오류로 명시적으로 먼저 나타나는 두 가지 이상의 오류 트리거링의 결과입니다. 하지만 대부분의 경우 명시적 오류의 처리가 잘못되어 오류가 치명적인 장애로 나타나는 경우가 많습니다.

이러한 잘못된 오류 처리의 만연은 치명적인 장애에 고유한 특징입니다. 이에 비해 본 연구에서 치명적이지 않은 장애의 25%만이 잘못된 오류 처리와 관련이 있었으며, 이는 치명적이지 않은 장애의 경우 오류 처리가 대부분 오류로 인해 전체 서비스가 중단되는 것을 방지하는 데 효과적이었다는 것을 나타냅니다.

전반적으로 개발자들이 오류 가능성을 예측하는 데 능숙하다는 것을 알 수 있었습니다. 단 한 가지 사례를 제외하고는 모두 개발자가 오류를 확인했습니다. 개발자가 오류를 확인하지 않은 유일한 사례는 Redis에서 확인되지 않은 오류 시스템 호출 반환이었습니다. 이는 파일 시스템 버그에 대한 이전 연구[24, 41, 55]에서 관찰된 특징과 다른데, 많은 오류가 확인되지도 않았습니다. 이러한 차이는 (i) Java 컴파일러가 개발자에게 체크된 예외를 모두 잡아내도록 강제하고, (ii) 대규모 분산 시스템에서는 다양한 오류가 발생할 것으로 예상되어 개발자가 보다 방어적으로 프로그래밍하기 때문일 가능성이 높습니다. 그러나 개발자들은 이러한 오류를 처리하는 데 있어 대개 허술한 경우가 많았습니다. 이는 아래의 결과 11과 12에서 더욱 확증됩니다. 공정하게 말하자면, 저희의 조사 결과가 왜곡되어 있다는 점을 지적해야 합니다.

[2]치명적인 장애를 분류할 때는 HA 기능이 항상 활성화되어 있다고 가정합니다. HA가 활성화되지 않은 상태에서 마스터 노드에 장애가 발생하면 시스템의 모든 사용자에게 영향을 미칠 수 있음에도 불구하고 치명적인 장애로 분류하지 않았습니다. 이러한 장애는 우리가 연구한 다른 장애와 비교할 때 고유한 장애가 아니라 마스터 노드에서 우연히 발생한 장애이기 때문입니다.

이 연구는 올바르게 포착되고 처리되는 많은 오류를 노출시키지 못했다는 것을 의미합니다.

그럼에도 불구하고 오류 처리 코드의 정확성은 그 영향력을 고려할 때 특히 중요합니다. 이전 연구[50, 52]에 따르면 분산 시스템 장애의 초기 결함은 매우 다양하며(예: 버그, 잘못된 계산, 노드 충돌, 하드웨어 결함), 실제로 대규모 데이터 센터에서 이 모든 결함을 제거하는 것은 불가능합니다[14]. 따라서 이러한 결함 중 일부가 오류로 나타나는 것은 불가피하며, 오류 처리는 최후의 방어선이 됩니다[45].

저희가 조사한 치명적인 장애 중 잘못된 오류 처리로 인해 트리거되지 않은 장애는 4건에 불과했습니다. 그 중 3건은 서버가 실수로 모든 클라이언트를 종료하는 치명적인 예외를 발생시켰기 때문이었습니다. 즉, 클라이언트의 오류 처리는 올바르게 이루어졌습니다. 다른 하나는 버그로 인해 DNS 조회 결과 캐싱이 비활성화되면서 성능이 크게 저하된 경우였습니다.

4.1 오류 처리기의 사소한 실수

치명적인 장애의 11 *35%는 오류 처리 로직의 사소한 실수(단순히 모범 프로그래밍 관행을 위반하는 실수)로 인해* 발생하며 다음과 같은 *원인이 있습니다*.
시스템 관련 지식 없이도 탐지할 수 있습니다.

그림 5는 실수를 (i) 오류 처리기가 명시적 오류를 무시하는 경우, (ii) 오류 처리기가 예외를 과도하게 포착하여 시스템을 중단하는 경우, (iii) 오류 처리기가 주석에 "TODO" 또는 "FIXME"를 포함하는 경우의 세 가지 범주로 더 세분화하여 보여줍니다.

치명적인 장애의 25%는 명시적 오류(오류만 기록하는 오류 처리기도 오류를 무시하는 것으로 간주됨)를 무시하여 발생했습니다. Java로 작성된 시스템의 경우 예외는 모두 명시적으로 발생했지만, Redis에서는 시스템 호출 오류 반환이었습니다. 그림 7은 예외 무시로 인한 HBase의 데이터 손실을 보여줍니다. 오류를 무시하고 방치하는 것은 나쁜 프로그래밍 관행으로 알려져 있지만[7, 60], 우리는 이것이 많은 치명적인 실패로 이어지는 것을 관찰했습니다. 적어도 개발자들은 오류를 기록하는 데 주의를 기울였습니다. Redis 개발자가 오류 시스템 호출 반환을 기록하지 않은 한 가지 경우를 제외하고 모든 오류는 기록되었습니다.

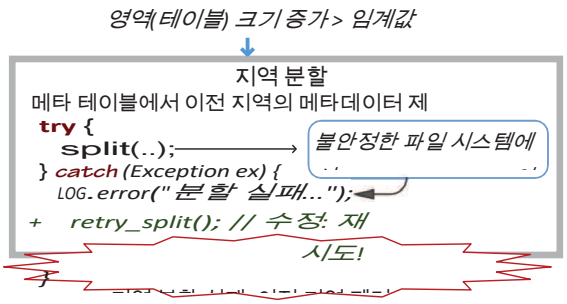


그림 7: 로깅 문을 제외하고 오류 처리가 단순히 비어 있는 HBase의 데이터 손실. 이 문제는 예외 처리기에서 다시 시도하는 것으로 해결되었습니다.

그림 8: 오버캐치로 인해 전체 HDFS 클러스터가 다운되었습니다.

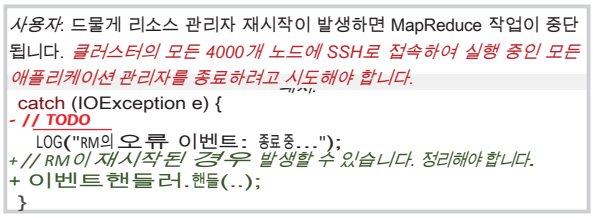


그림 9: 개발자가 오류 처리기에 '할 일'을 남긴 MapReduce의 치명적인 오류.

치명적인 장애의 또 다른 8%는 개발자가 치명적이지 않은 예외로 인해 전체 클러스터를 조기에 중단했기 때문에 발생했습니다. 원칙적으로 전체 클러스터를 중단할 시점을 결정하려면 시스템에 대한 구체적인 지식이 필요하지만, 우리가 관찰한 중단은 모두 *예외 오버캐치*(상위 수준의 예외가 하위 수준의 여러 예외를 잡는 데 사용되는 경우)에 속하는 것이었습니다. 그림 8은 이러한 예시를 보여줍니다. `exit()`는 IncorrectVersionException에 대해서만 의도되었습니다. 그러나 개발자는 상위 수준 예외를 포착합니다: Throwable. 결과적으로 네임노드의 결함으로 인해 `registerDatanode()`가 RemoteException을 던지게 되면 Throwable에 의해 과도하게 잡혀 모든 데이터노드가 다운되었습니다. 이 문제를 해결하기 위해 RemoteException을 명시적으로 처리하여 IncorrectVersionException만 발생하도록 수정했습니다. 그

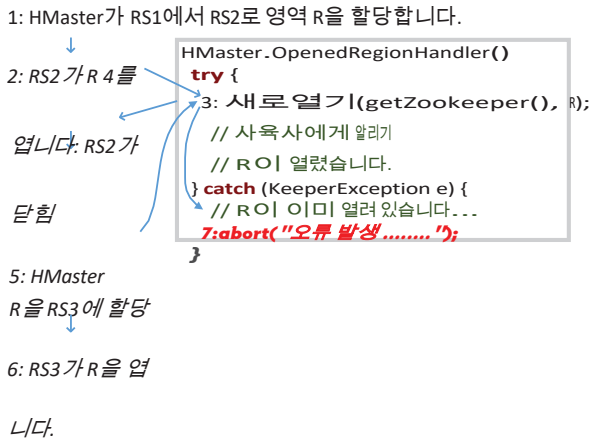


그림 10: 오류 처리 코드가 잘못되어 테스트가 전혀 이루어지지 않은 치명적인 장애. 드문 이벤트 시퀀스로 인해 새로 열림() 이 드문 KeeperException을 발생시켰고, 이로 인해 전체 HBase 클러스터가 중단되었습니다.

4.2 시스템 관련 버그

치명적인 장애의 나머지 57%는 다음과 같은 원인으로 발생합니다.

버그를 탐지하는 데 시스템별 지식이 필요한 잘못된 오류 처리로 인해 발생합니다. (그림 5 참조)

러나 나중에 코드가 발전할 때 다른 예외가 다시 과도하게 잡힐 수 있으므로 여전히 나쁜 관행입니다. 안전한 방법은 사전 예외를 잡는 것입니다[7].

그림 9는 훨씬 더 명백한 실수를 보여주는데, 개발자가 로깅 문과 함께 핸들러 로직에 "TODO"라는 주석만 남겼습니다. 이 오류는 드물게 발생하지만 4,000개의 노드로 구성된 프로덕션 클러스터를 다운시켰습니다.

발견 12 *치명적인 장애의 23%에서 오류 처리의 실수는 시스템에 따라 다르지만 여전히 쉽게 발견할 수 있습니다. 좀 더 공식적으로 말하자면, 이러한 경우 잘못된 오류 처리는 오류 처리 로직에 대한 100% 문 커버리지 테스트를 통해 노출될 수 있습니다.*

즉, 오류 처리 코드에서 문제가 있는 기본 블록이 트리거되면 오류는 반드시 노출됩니다. 이는 이러한 기본 블록에 완전히 결함이 있으며 제대로 테스트되지 않았다는 것을 의미합니다. 그림 10은 이러한 예시를 보여줍니다. 테스트 케이스가 종결적으로 KeeperException을 트리거할 수 있게 되면 치명적인 실패가 100% 확실하게 트리거됩니다.

따라서 이러한 오류를 방지하는 좋은 전략은 기존 오류 처리 로직에서 시작하여 오류를 트리거하는 테스트 케이스를 리버스 엔지니어링하는 것입니다. 예를 들어, 심볼릭 실행 기법[8, 10]을 확장하여 시스템 진입점에서 모든 실행 경로를 맹목적으로 탐색하는 대신 오류 처리 코드 블록에 도달할 수 있는 실행 경로를 의도적으로 재구성할 수 있습니다.

오류 처리 코드에서 높은 문 커버리지를 달성하기는 어려울 수 있지만, 테스트에서 높은 상태 커버리지를 목표로 하는 것이 무작위 오류 주입을 적용하는 전략보다 더 나은 전략이 될 수 있습니다. 예를 들어, 그림 10의 실패는 버그가 있는 오류 처리기를 트리거하기 위해 매우 드문 이벤트 조합이 필요합니다. 이번 연구 결과는 오류 처리 로직에서 시작하여 테스트 사례를 리버스 엔지니어링하여 오류를 노출하는 '상향식' 접근 방식이 더 효과적일 수 있음을 시사합니다.

오류 처리 로직에 대한 기존 테스트 기법은 주로 테스트 입력 또는 모델 확인을 사용하여 시스템을 시작하고[23, 65], 여러 단계에서 오류를 능동적으로 주입하는 "하향식" 접근 방식을 사용합니다[9, 18, 44]. LFI[44] 및 Fate&Destini[23]와 같은 도구는 적절한 지점에서만 오류를 주입하고 중복 주입을 피할 수 있는 지능적인 도구입니다. 이러한 기법들은 필연적으로

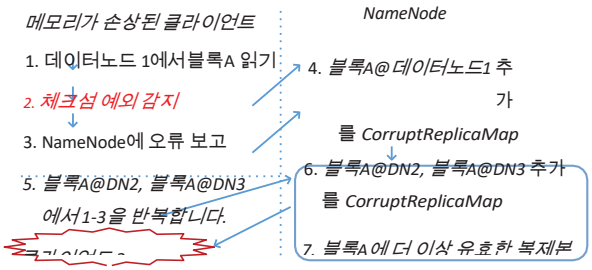


그림 11: HDFS의 모든 클라이언트에 대한 대규모 데이터 손실. RAM이 손상된 클라이언트는 네임노드에 읽는 거의 모든 블록에서 데이터 손상을 보고했습니다. 네임노드는 데이터노드에서 체크섬을 확인하는 대신 결함이 있는 클라이언트를 맹목적으로 신뢰하고 블록을 영구적으로 손상된 것으로 표시하여 모든 클라이언트에 막대한 데이터 손실을 초래합니다.

소프트웨어 시스템의 신뢰성을 개선했습니다. 실제로 Hadoop 개발자는 시스템을 테스트하기 위해 자체적인 오류 주입 프레임워크를 가지고 있으며[18], 우리가 연구한 생산 실패는 이러한 도구에서 놓친 것일 가능성이 높습니다. 그러나 우리의 연구 결과는 이러한 "하향식" 접근 방식이 이러한 나머지 프로덕션 장애를 추가로 발견하는 데 도움이 될 수 있음을 시사합니다. 이러한 접근 방식은 입력된 오류로 인해 서비스가 중단되기 전에 먼저 시스템을 드문 상태로 만들기 위해 드문 입력 이벤트 순서를 필요로 합니다. 또한 장애의 38%는 장기간 운영되는 시스템에서만 발생합니다. 따라서 가능한 공간은 입력 이벤트는 단순히 추출할 수 없습니다.

복잡한 버그: 나머지 치명적인 장애의 34%는 오류 처리 로직의 복잡한 버그와 관련이 있습니다. 이는 개발자가 특정 오류 시나리오를 예상하지 못한 경우입니다. 예를 들어 그림 11에 표시된 장애를 생각해 보겠습니다. 이 처리 로직은 대부분의 체크섬 오류에는 적합하지만, 단일 클라이언트가 매우 짧은 시간 내에 대량의 손상(RAM 손상으로 인한)을 보고하는 시나리오를 고려하지 않았습니다. 거의 비잔틴에 가까운 이러한 유형의 오류는 실제로 테스트하기 가장 어렵습니다. 이러한 오류를 탐지하려면 시스템 작동 방식을 이해하고 실제 오류 모드를 모두 예측해야 합니다. 이번 연구에서 이러한 버그를 식별하는 방법에 대한 건설적인 제안을 제공할 수는 없지만, 치명적인 장애의 1/3만 차지하는 것으로 나타났습니다.

4.3 토론

거의 모든 치명적인 장애가 잘못된 오류 처리의 결과임을 보여주었지만, 실제 분산 시스템에서 대부분의 코드가 오류 처리 블록에서 도달할 수 있으므로 대부분의 버그는 "잘못된 오류 처리"라고 주장할 수 있습니다. 하지만 이번 연구 결과에 따르면 예외 처리기 블록(예: Java의 `catch` 블록)만 검사해도 많은 버그를 발견할 수 있습니다. 표 9에서 볼 수 있듯이 이러한 시스템에서 캐치 블록의 수는 다음과 같습니다.

상대적으로 작습니다. 예를 들어, HDFS에서는 캐치 블록이 2652개에 불과합니다. 특히 Finding 11에서 '사소한 실수' 범주에 속하는 버그는 이러한 캐치 블록만 검사해도 쉽게 발견할 수 있습니다.

흥미로운 질문은 대형 인터넷 소프트웨어 공급업체의 서비스 중단이 잘못된 오류 처리의 결과인지 여부입니다. 내부 장애 데이터베이스에 액세스하지 않고서는 이 질문에 명확하게 답할 수는 없지만, 가장 눈에 띄는 일부 장애에 대한 사후 분석이 공개되어 있습니다. 흥미롭게도 일부 일화적인 서비스 중단은 잘못된 오류 처리의 결과입니다. 예를 들어, 약 2시간 30분 동안 facebook.com이 다운된 장애는 당시 "페이스북이 4년 만에 겪은 최악의 장애"였으며, "장애를 심각하게 만든 핵심 결함은 오류 상황을 잘못 처리한 것"이었습니다[17]. 2011년에 발생한 아마존 웹 서비스 중단[59]으로 인해 Reddit, Quora, FourSquare, 뉴욕 타임즈 웹사이트 일부 및 기타 약 70개 사이트가 다운되었을 때, 초기 원인은 과중한 워크로드를 위한 것이 아닌 보조 네트워크로 프로덕션 트래픽을 잘못 라우팅하는 구성 변경이 원인이었습니다. 결과적으로 노드에 장애가 발생하기 시작했습니다. 이것이 서비스 수준 장애로 이어진 것은 노드 장애에 대한 잘못된 처리 방식, 즉 "새로운 노드를 찾지 못한 노드가 공간을 찾을 수 없을 때 충분히 공격적으로 물러나지 않고 계속해서 재차 검색"이었기 때문입니다. 이로 인해 네트워크 트래픽이 더 많이 발생했고 결국 서비스 수준 장애로 이어졌습니다.

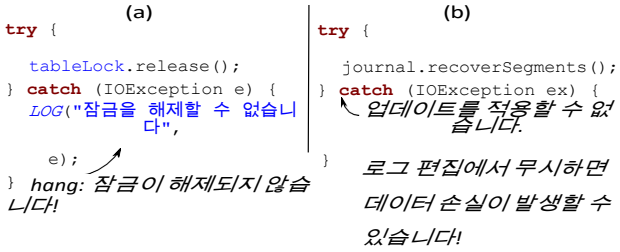
5 흡인기: 간단한 검사기

섹션 4.1에서 대부분의 카타스 트로피 장애는 (i) 오류 처리기가 비어 있음 등 세 가지 간단한 범주에 속하는 사소한 실수로 인해 발생한다는 사실을 확인했습니다;
(ii) 오류 처리기가 예외 및 중단을 과도하게 포착하는 경우.
(iii) 오류 처리기에는 "TODO" 및 "FIXME"와 같은 문구가 포함되어 있습니다. 이러한 간단한 규칙의 적용 가능성을 측정하기 위해 이러한 버그 패턴을 찾을 수 있는 규칙 기반 정적 검사기인 Aspirator를 구축했습니다. 다음에서는 Aspirator의 구현 방법과 여러 시스템에 적용한 결과에 대해 설명합니다.

5.1 흡인기 구현

Java 바이트코드에서 Chord 정적 분석 프레임워크[11]를 사용하여 Aspirator를 구현했습니다. 애스피레이터는 다음과 같이 작동합니다. Java 바이트코드를 인스트럭션별로 스캔합니다. 명령어가 예외 *e를 발생시킬* 수 있는 경우, Aspirator는 해당 예외에 해당하는 캐치 블록을 식별하고 기록합니다. 캐치 블록이 비어 있거나 로그 인쇄문만 포함된 경우, 또는 캐치 블록에 "TODO" 또는 "FIXME" 컴포넌트가 포함된 경우 Aspirator는 경고를 발신합니다.

멘트를 해당 소스 코드에 추가합니다. 또한
상위 수준의 예외에 대한 캐치 블록이 발생하면 경고가 표시됩니다.
(예: 예외 또는 던지기 가능)가 *여러 개의*
하위 수준 예외를 호출하고 동시에 중단을 호출합니다.
또는 `System.exit()`를 호출할 수 있습니다. 흡인기는 이러한 *초과 어획을* 식별할 수 있는데, 그 이유는 어획에 도달하면



블록을 사용하면 캐치 블록이 처리하는 인-구조에서 어떤 예외가 발생하는지 정확히 알 수 있습니다.

모든 빈 캐치 블록이 반드시 나쁜 관행이나 버그인 것은 아닙니다. 예외가 캐치 블록 외부에서 처리되는 다음 예제를 고려해 보세요:

```
uri = null;
try {
 uri = Util.fileAsURI(새로운 파일(uri));
  } catch (IOException ex) { /* 빈 */ }
if (uri == null) { // 여기서 처리하세요!
```

따라서 흡인기는 비어 있을 때 경고를 발산하지 않습니다
.
캐치 블록을 호출합니다. 다음 조건이 모두 참인 경우:
(i) 해당 시도 블록이 변수 V를 수정하고, (ii) 캐치 블록 뒤의 기본 블록에서 V의 값을 확인합니다. 또한 해당 시도 블록의 마지막 인스트럭션이 반환, 중단 또는 계속이고 캐치 블록 이후의 블록이 비어 있지 않은 경우, 캐치 블록 이후의 모든 로직이 사실상 예외 처리 중이므로 캐치 블록이 비어 있는 경우 Aspirator는 경고를 보고하지 않습니다.

또한 프로그래머가 오탐과 오탐 사이의 절충점을 조정할 수 있도록 런타임 구성 옵션을 제공합니다. 이를 통해 프로그래머는 경고가 발생하지 않아야 하는 예외를 지정할 수 있습니다. 테스트에서는 FileNotFound 예외의 대다수가 실제 오류를 나타내지 않는다는 것을 알았기 때문에 모든 예외를 무시했습니다. 또한 프로그래머가 특정 메서드를 분석에서 제외할 수 있습니다. 테스트에서는 이 기능을 사용하여 무시된 예외가 종료, 닫기 또는 정리 메서드에서 발생한 경우 경고를 표시하지 않았는데, 정리 단계의 예외는 어차피 시스템이 다운되는 것이기 때문에 그다지 중요하지 않을 가능성이 높습니다. 이 두 가지 휴리스틱을 사용해도 치명적인 장애로 이어지는 사소한 실수를 탐지하는 Aspirator의 기능에는 영향을 미치지 않았지만 오탐의 수는 현저히 감소했습니다.

제한 사항: 개념 증명으로서 Aspirator는 현재 Java 및 Java 바이트코드와 호환되는 기타 언어(예: Scala)에서만 작동하며, 예외가 있습니다.

는 해당 언어에서 지원되며 명시적으로 포착해야 합니다. Aspirator를 Java 이외의 프로그램으로 확장하는 데 있어 가장 큰 어려움은 오류 조건을 식별하는 것입니다. 그러나 시스템 호출 오류 반환, 스위치 폴스루, `abort()` 호출 등 일부 오류 조건은 여전히 쉽게 식별할 수 있습니다.

또한 흡인기는 발생하는 경고의 심각도를 예측할 수 없습니다. 따라서 모든 경고가 장애로 이어질 수 있는 버그를 식별하는 것은 아닙니다,

그림 12: 흡인기에서 발견된 두 가지 새로운 버그.

일부 오탐이 발생할 수 있습니다. 그러나 Aspirator는 각 경고와 함께 적발된 예외 목록과 해당 예외를 던지는 지침을 제공하기 때문에 대부분의 경우 개발자는 각 경고의 중요도를 신속하게 평가하고 향후 특정 경고를 억제하기 위해 프로그램에 주석을 달 수 있습니다.

마지막으로, 흡인기의 기능은 FindBugs [32]와 같은 기존의 정적 분석 도구에 추가될 수 있으며, 추가되어야 할 가능성이 높습니다.

5.2 실제 시스템 확인

먼저 연구에 사용된 치명적인 장애 세트에 대해 Aspirator를 평가했습니다. Aspirator를 사용하고 캡처된 버그를 수정했다면, 우리가 연구한 Cassandra, HBase, HDFS, MapReduce의 치명적인 장애 중 33%를 예방할 수 있었을 것입니다.

그런 다음 Aspirator를 사용하여 9개의 분산 시스템 또는 분산 시스템 구축에 사용되는 구성 요소(예: Tomcat 웹 서버)의 최신 안정 버전을 확인했습니다. 2.7GHz 인텔 코어 i7 프로세서가 탑재되고 메모리 사용량이 1.2GB 미만인 MacBook Pro 노트북에서 각 시스템에 대해 15초 이내에 Aspirator의 분석이 완료되었습니다. Aspirator가 생성하는 각 경고는 버그, 잘못된 관행, 오탐의 세 가지 범주 중 하나로 분류합니다. 각 경고에 대해 다음과 같은 최선의 노력을 기울입니다. 예외 처리 로직의 결과를 이해해야 합니다. 경고는 예외가 발생하면 처리 로직이 실패로 이어질 수 있다고 확실하게 결론을 내릴 수 있는 경우에만 버그로 분류됩니다. 오류로 이어지지 않을 것이라고 명확히 이해한 경우에는 오탐으로 분류했습니다. 그 외의 모든 경우는 도메인 지식 없이는 예외 처리 로직의 결과를 확실하게 유추할 수 없는 경우입니다. 따라서 우리는 이를 보수적으로 나쁜 관행으로 분류합니다.

표 9는 결과를 보여줍니다. 전체적으로 Aspirator는 500개의 새로운 버그 및 잘못된 관행과 115개의 오탐지를 탐지했습니다. 이러한 시스템 대부분은 이미 코드 체크인 시마다 400개 이상의 규칙을 검사하는 FindBugs[32]와 같은 최첨단 정적 검사기를 실행하고 있다는 점에 유의하세요. 하지만 Aspirator는 이들 모두에서 새로운 버그를 발견했습니다.

버그: 흡인기에서 탐지된 많은 버그는 실제로 치명적인 장애로 이어질 수 있습니다. 예를 들어, 오버캐치 중단 검사기에서 발견된 4개의 버그는 모두 다음과 같은 결과를 가져올 수 있습니다.

시스템	핸들러 블록	버그					나쁜 관행					False pos.
		합계	확인	무시	중단	할일	합계	확인됨	무시	중단	할일	
카산드라	4,365	2	2	2	-	-	2	2	2	-	-	9
클라우드 스택	6,786	27	24	25	-	-	185	21	182	1	2	20
HDFS	2,652	24	9	23	-	1	32	5	32	-	-	16
HBase	4,995	16	16	11	3	2	43	6	35	5	3	20
하이브	9,948	25	15	23	-	2	54	14	52	-	2	8
Tomcat	5,257	7	4	6	1	-	23	3	17	4	2	30
Spark	396	2	2	-	-	2	1	1	1	-	-	2
YARN/MR2	1,069	13	8	6	-	7	15	3	10	4	1	1
사육사	1,277	5	5	5	-	-	24	3	23	-	1	9
합계	36,745	121	85	101	4	16	379	58	354	14	11	115

표 9: 9개의 분산 시스템에 흡인기 적용 결과. 케이스가 여러 범주에 속하는 경우(예: 빈 핸들러에 "TODO" 주석이 포함될 수 있음) 무시된 예외로 한 번만 계산합니다. "핸들러 블록" 열에는 Aspirator가 발견하고 분석한 예외 처리 블록의 수가 표시됩니다. "-"는 흡인기가 0개의 경고를 보고했음을 나타냅니다.

그림 8과 유사한 방식으로 예기치 않은 예외로 클러스터를 다운시킬 수 있습니다. 4개 모두 수정되었습니다. 일부 버그로 인해 클러스터가 중단될 수도 있습니다. Aspirator는 그림 12 (a)에 표시된 것과 유사한 패턴을 가진 5개의 버그를 HBase 및 Hive에서 감지했습니다. 이 예제에서는 `tableLock`을 해제할 수 없는 경우 HBase는 오류 메시지만 출력하고 실행을 계속하여 테이블에 액세스하는 모든 서버를 교착 상태로 만들 수 있습니다. 개발자는 즉시 상태를 정리하고 문제가 있는 서버를 중단하여 이 버그를 수정했습니다[31].

그림 12 (b)는 데이터 손실로 이어질 수 있는 버그를 보여줍니다. HDFS가 편집 로그에서 업데이트를 재생하여 사용자 데이터를 복구할 때 IOException이 발생할 수 있습니다. 이를 무시하면 조용한 데이터 손실이 발생할 수 있습니다.

잘못된 관행: 잘못된 관행 사례에는 해당 버그를 명확하게 확인할 수 없는 잠재적 버그가 포함됩니다. 결과를 초래할 수 있습니다. 예를 들어, 임시 파일을 삭제할 때 예외가 발생하고 이후 무시되는 경우라면 별 문제가 아닐 수 있습니다. 하지만 파일 시스템에 더 심각한 문제가 있음을 나타낼 수 있기 때문에 나쁜 관행으로 간주됩니다. 이러한 사례 중 일부는 오탐일 수도 있습니다. 얼마나 많은 사례가 오탐인지 확인할 수는 없지만, 처음에 '나쁜 관행'으로 분류한 사례 중 87건을 개발자에게 보고했습니다. 이 중 58건은 확인 또는 수정되었지만 17건은 거부되었습니다. 거부된 17건은 이후 '거짓'으로 분류되었습니다. 양성" 표 9에서 확인할 수 있습니다.

오탐: Aspi- rator가 보고한 경고 중 19%는 오탐입니다. 대부분은 As-pirator가 절차 간 분석을 수행하지 않기 때문입니다. 단점 메서드 호출의 반환값을 테스트하여 예외를 처리하는 다음 예제를 살펴보세요:

```
try {
   set_A();
} catch (SomeException e) { /* 빈 */ }
if (A_is_not_set()) {/* 여기서 처리하세요! */}
```

파일 찾을 수 없음 및 다음의 예외 외에도 종료, 닫기 및 청소, 흡인기에는 다음이 있어야 합니다.

은 다른 예외에 대한 경고를 제외하도록 추가로 구성되었습니다. 예를 들어, 많은 오탐은 NoSuchFieldException과 같은 Java의 리플렉션 관련 예외의 빈 핸들러로 인해 발생합니다. 프로그래머가 Aspirator의 분석에서 예외를 제외했어야 한다는 사실을 알게 되면, 이 예외를 Aspirator의 구성 파일에 추가하기만 하면 됩니다.

5.3 경험

개발자와의 상호작용: 공식 버그 추적 웹사이트를 통해 171건의 버그와 잘못된 관행을 개발자에게 보고했습니다. 현재까지 143건이 이미
(73건은 수정 완료, 나머지 70건은 수정 완료되었으나 아직 수정되지 않음), 17건은 거부되었으며, 나머지는 응답을 받지 못했습니다.

개발자들로부터 다양한 피드백을 받았습니다. 한편으로는 다음과 같은 긍정적인 의견도 받았습니다: "*예외를 제대로 사용하고 예외를 무시하지 않기를 바라기 때문에 이 라인의 문제를 수정하고 싶습니다.*", "*아무도 이 숨겨진 기능을 보지 않았을 것입니다. 예외를 무시하는 것은 바로 이런 이유로 나쁩니다.*", "*Throwable(즉, 예외 오버캐치)을 잡는 것은 나쁘므로 이를 수정해야 합니다.*" 등의 긍정적인 의견도 있었습니다. 반면에 다음과 같은 부정적인 의견도 있었습니다: "*모든 예외를 처리해야 할 이유를 모르겠다*" 등의 부정적인 의견도 있었습니다.

개발자가 오류 처리를 소홀히 하는 데에는 몇 가지 이유가 있습니다. 첫째, 이렇게 무시되는 오류는 적절히 처리할 만큼 중요하지 않은 것으로 간주될 수 있습니다. 시스템에 심각한 장애가 발생한 후에야 개발자는 오류 처리의 중요성을 깨닫게 되는 경우가 많습니다. 오늘날 가장 치명적인 장애의 대부분은 오류 처리 로직의 정확성을 간과하는 데서 비롯된다는 점을 보여줌으로써 개발자의 인식을 제고하고자 합니다.

둘째, 개발자는 오류가 발생하지 않거나 매우 드물게 발생할 것이라고 생각할 수 있습니다. HBase에서 Aspirator가 감지한 다음 코드 스니펫을 고려해 보십시오:

```
try {
  t = 새로운 TimeRange(timestamp,
  timestamp+1);
} catch (IOException e) {
  // 절대 일어나지 않음
}
```

이 경우 개발자는 생성자가 예외를 던질 수 없다고 생각하여 코드의 주석에 따라 예외를 무시했습니다. 저희가 확인한 여러 시스템에서 유사한 주석이 포함된 빈 오류 처리기가 많이 발견되었습니다. 저희는 "절대 발생할 수 없는" 오류는 전파를 방지하기 위해 방어적으로 처리해야 한다고 주장합니다. 개발자의 판단이 틀릴 수도 있고, 나중에 코드가 발전하면서 오류가 발생할 수도 있으며, 이러한 예기치 않은 오류가 전파되도록 허용하면 치명적일 수 있기 때문입니다. 위의 HBase 예제에서 개발자의 판단은 실제로 잘못되었습니다. 생성자는 다음과 같이 구현됩니다:

```
공용 TimeRange (긴 최소, 긴 최대)
IOException을 던집니다 {
  if (최대 < 최소)
    새로운 IOException("최대 < 최소")을 던집니다;
}
```

인테그레이션 오버플로우가 발생하면 IOException이 발생할 수 있으며, 이 예외를 삼키면 데이터가 손실될 수 있었습니다. 개발자들은 나중에 IOException을 올바르게 처리하여 이 문제를 해결했습니다.

셋째, 오류를 적절히 처리하는 것이 어려울 수 있습니다. 시스템의 정상적인 실행 경로보다 비정상적인 실행 경로의 정확성을 추론하는 것이 훨씬 더 어려운 경우가 많습니다. 많은 예외가 적절한 문서가 없는 타사 컴포넌트에 의해 발생한다는 현실 때문에 문제는 더욱 악화됩니다. 많은 경우 개발자조차도 예외의 발생 원인이나 잠재적인 단점을 완전히 이해하지 못하는 경우가 많을 것으로 예상됩니다. 이는 CloudStack의 아래 코드 스니펫을 보면 알 수 있습니다:

```
} catch (NoTransitionException ne) {
  /* 왜 이런 일이 일어날 수 있을까요? 내가 아니라
  신께 물어보세요. */
}
```

다른 시스템에서도 빈 예외 처리기에서 유사한 코멘트가 관찰되었습니다.

마지막으로, 실제로 릴리스 마감일이 다가오면 기능 개발이 예외 처리보다 우선시되는 경우가 많습니다. 당황스럽게도 Aspirator의 코드에서 이 문제를 직접 경험한 적이 있습니다. 5개의 빈 예외 처리기가 발견되었는데, 모두 기본 라이브러리에서 던져진 예외를 잡기 위한 용도로 코드가 컴파일되도록 하기 위해 배치된 것이었습니다.

Cassandra의 모범 사례: 저희가 확인한 9개 시스템 중 Cassandra의 버그 대 처리기 차단 비율이 가장 낮았는데, 이는 Cassandra 개발자가 예외 처리에 있어 모범 프로그래밍 사례를 신중하게 따르고 있음을 나타냅니다. 특히, 대부분의 예외는 재귀적으로 예외를

호출자이며 호출 그래프에서 최상위 메서드에 의해 처리됩니다. 흥미롭게도, 우리가 연구한 5개 시스템 중 무작위로 샘플링한 장애 세트에서 치명적인 장애 발생률이 가장 낮은 시스템도 Cassandra였습니다(표 1 참조).

6 관련 작업

많은 연구에서 분산 시스템의 장애를 특성화하여 이러한 장애를 훨씬 더 깊이 이해하고 신뢰성을 개선했습니다. 저희의 연구는 이러한 장애의 종단 간 발현 순서를 이해한 (저희가 아는 한) 최초의 분석입니다. 수동 분석을 통해 가장 치명적인 장애의 발현 순서에서 가장 약한 연결 고리, 즉 잘못된 오류 처리를 찾을 수 있었습니다. 오류 처리가 많은 오류의 원인이라는 것은 잘 알려져 있지만, 오류 처리 코드의 이러한 버그는 대부분 매우 단순하며 오늘날 치명적인 장애의 주요 원인이라는 것을 발견했습니다.

다음으로 특성화 연구, 오류 처리 코드 연구, 분산 시스템 테스트 등 세 가지 범주의 관련 작업에 대해 설명합니다.

장애 특성화 연구 오펜하이머 *등은* 11년 전에 배포된 인터넷 서비스에서 100건이 넘는 장애 보고서를 연구했습니다[50]. 이들은 이러한 장애의 근본 원인, 복구 시간, 완화 전략에 대해 논의하고 일련의 흥미로운 발견을 요약했습니다(예: 운영자 실수가 가장 큰 원인). 오픈소스 프로젝트를 통해 소스 코드, 로그, 개발자의 토론 내용 등 기존 연구에서는 얻을 수 없었던 더 풍부한 데이터 소스를 조사할 수 있었기 때문에 이번 연구는 상호 보완적인 측면이 많습니다. 실제로 저자들이 인정했듯이, "시스템 로그와 버그 추적 데이터베이스를 조사할 수 있었다면 자세한 원인에 대해 더 많이 알 수 있었을 것"이라고 합니다.

Rabkin과 Katz[52]는 클라우드 시대의 프로덕션 하둡 클러스터의 보고서를 분석했습니다. 그들의 연구는 장애의 근본 원인을 분류하는 데 중점을 두었습니다.

Li *등*[38]은 SCOPE로 작성된 Microsoft Bing의 데이터 분석 작업의 버그를 연구했습니다. 그들은 대부분의 버그가 데이터 처리 로직에 있으며 테이블 스키마의 잦은 변경으로 인해 발생하는 경우가 많다는 것을 발견했습니다.

다른 연구자들은 비분산 시스템의 버그를 연구했습니다. 1985년 Gray는 Tan- dem[22] 운영 체제에서 100건이 넘는 장애를 조사한 결과 운영자 실수와 소프트웨어 버그가 두 가지 주요 원인이라는 것을 발견했습니다. Chou *등*[12]은 OS 버그를 연구한 결과 장치 드라이버에 버그가 가장 많다는 것을 관찰했습니다. 이 발견은 디바이스 드라이버 품질을 개선하기 위한 많은 시스템과 도구로 이어졌고, 10년 후의 연구[51]에 따르면 디바이스 드라이버의 품질이 실제로 크게 개선되었다고 합니다. Lu *등*[42]은 서버 프로그램의 통화 버그를 연구한 결과, 서버 프로그램 간에 많은 버그가 있음을 발견했습니다.

예를 들어, 거의 모든 동시성 버그가 2개의 스레드를 사용하여 트리거될 수 있다는 사실을 발견했습니다.

오류 처리 코드에 대한 연구 많은 연구에 따르면 오류 처리 코드에 버그가 있는 경우가 많습니다[24, 44, 55, 58]. 구나위 *등은*정적 검사기를 사용하여 파일 시스템과 저장 장치 드라이버가 오류 코드를 올바르게 전파하지 않는 경우가 많다는 사실을 발견했습니다[24]. 푸와 라이더도 많은 자바 프로그램에서 상당수의 캐치 블록이 비어 있다는 사실을 발견했습니다[20]. 하지만 이들이 오류를 유발했는지는 연구하지 않았습니다. 설리반 *등은* 1986년과 1989년 사이에 IBM의 MVS 운영 체제를 사용한 현장 장애에 대한 연구에서 잘못된 오류 복구가 장애의 21%, 영향이 큰 장애의 36%의 원인이라는 것을 발견했습니다[58]. 이에 비해 우리가 연구한 분산 시스템에서는 잘못된 오류 처리가 치명적이지 않은 장애의 25%, 치명적인 장애의 92%를 발생시킨 것으로 나타났습니다.

많은 테스트 도구가 잘못된 정보를 효과적으로 노출할 수 있습니다.
에러 주입을 통한 에러 처리 [18, 23, 44]. Fate&Destini [23]는 여러 오류의 고유한 조합을 지능적으로 주입할 수 있으며, LFI [44]는 프로그램/라이브러리 경계에서 오류를 선택적으로 주입하고 중복되는 오류 주입을 방지합니다. 이러한 도구는 많은 잘못된 오류 처리 버그를 노출하는 데 효과적일 수 있지만, 모두 "하향식" 접근 방식을 사용하며 사용자/테스터가 시스템을 구동하기 위해 워크로드를 제공해야 합니다. 이번 연구에서는 하향식 접근 방식으로는 트리거하기 어려운 오류 상태로 시스템을 구동하려면 입력 이벤트의 조합이 필요하다는 사실을 발견했습니다. 연구 결과에 따르면 오류 처리 로직에서 테스트 케이스를 재구성하는 '상향식' 접근 방식이 치명적인 오류로 이어지는 대부분의 결함을 효과적으로 방지할 수 있습니다.

다른 도구는 오류 한에서 버그를 식별할 수 있습니다.
정적 분석[24, 55, 67]을 통해 코드를 탐지합니다. EIO [24]는 정적 분석을 사용하여 확인되지 않거나 더 이상 전파되지 않는 오류 코드를 탐지합니다. Errlog [67]는 *로깅되지* 않은 오류 처리 코드를 보고합니다. 이에 비해 저희의 간단한 검사기는 보완적입니다. 이 검사기는 로깅 여부에 관계없이 검사되었지만 잘못 처리된 예외를 감지합니다.

분산 시스템 테스트 모델 검사 [25, 34, 37, 65] 도구를 사용하여 다양한 이벤트의 대규모 조합을 체계적으로 탐색할 수 있습니다. 예를 들어, SAMC[37]는 여러 오류를 지능적으로 주입하여 대상 시스템을 코너 케이스로 몰아넣을 수 있습니다. 이 연구는 사용자가 이러한 도구를 사용할 때 정보에 입각한 결정을 내리는 데 도움이 됩니다(예: 사용자는 3개 이하의 노드만 확인해야 함).

7 결론

이 백서에서는 널리 사용되는 데이터 집약적인 5가지 장애에 대해 사용자가 보고한 198건의 장애를 심층적으로 분석했습니다.

12가지 결과를 도출했습니다. 장애로 이어지는 오류 발현 순서는 비교적 복잡한 것으로 나타났습니다. 그러나 가장 치명적인 장애의 경우 거의 모든 장애가 잘못된 오류 처리로 인해 발생하며, 그 중 58%는 사소한 실수이거나 문 커버리지 테스트를 통해 노출될 수 있다는 사실도 발견했습니다.

기존의 테스트 기법으로 이러한 오류 처리 버그를 성공적으로 발견할 수 있을지는 의문입니다. 이러한 기법들은 모두 "하향식" 접근 방식을 사용합니다. 일반적인 입력이나 모델 검사를 사용하여 시스템을 시작하고[65, 23], 여러 단계에서 오류를 적극적으로 주입하는 방식입니다[9, 18, 44]. 하지만 입력 및 상태 공간의 크기와 장기간 실행되는 시스템에서만 상당한 수의 오류가 발생한다는 사실 때문에 이러한 버그를 노출하는 문제는 해결하기가 어렵습니다. 예를 들어, 하둡에는 시스템을 테스트하기 위한 자체 오류 주입 프레임워크가 있지만[18], 우리가 연구한 프로덕션 장애는 이러한 도구가 놓친 것일 가능성이 높습니다.

대신, 우리는 이러한 버그를 발견하기 위해 세 가지 접근 방식을 제안합니다. (1) 여러 가지 사소한 버그를 식별할 수 있는 Aspi- rator와 유사한 도구를 사용하고, (2) 오류 처리 로직이 단순히 잘못된 경우가 많으므로 오류 처리 코드에 대한 코드 리뷰를 시행하고, (3) 확장된 기호 실행 기법[8, 10]을 사용하여 각 오류 처리 코드 블록에 도달할 수 있는 실행 경로를 의도적으로 재구성하는 것입니다. 장애에 대한 자세한 분석과 Aspirator의 소스 코드는 http://www.eecg.toronto.edu/failureAnalysis/ 에서 공개적으로 확인할 수 있습니다.

감사

익명의 검토자, 목자 Jason Flinn, 레오니드 리지크가 생생한 피드백을 제공해 주셔서 대단히 감사합니다. 이 연구는 NSERC 디스커버리 보조금, NetApp 교수 펠로우십, Con- naught 신진 연구자상으로 지원되었습니다.

참조

[1] Amazon의 클라우드 타이타닉이 다운된 이유: //money.cnn.com/2011/04/22/technology/a mazon ec2 cloud outage/index.htm.

[2] Apache Cassandra. http://cassandra.apache.o rg.

[3] Apache HBase. http://hbase.apache.org.

[4] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. dOS의 억제 프로세스 그룹. *운영 체제 설계 및 구현에 관한 제 9회 USENIX 컨퍼런스*, OSDI'10, 2010.

[5] I. Beschastnikh, Y. Brun, , J. Abrahamson, M. D. Ernst, 그리고 A. 크리슈나무르티. 선언적 사양을 통한 FSM 추론 알고리즘 통합. In *Proceedings*

국제 소프트웨어 엔지니어 컨퍼런스, ICSE'13, 2013.

[6] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, 및 T. E. Anderson. 부분적으로 정렬된 로그에서 시간 불변량 마이닝. *시스템 로그 분석 및 기계 학습 기법 적용을 통한 대규모 시스템 관리*, SLAML'11, 3:1-3:10, 2011.

[7] J. Bloch. *효과적인 자바 (2 판)*. Prentice Hall, 2008.

[8] C. 캐더, D. 던바, D. 엥글러. Klee: 복잡한 시스템 프로그램에 대한 높은 커버리지 테스트의 비보조 및 자동 생성. *제8회 USENIX 논문집에서 운영 체제 설계 및 구현 컨퍼런스*, OSDI'08, 209-224 페이지, 2008.

[9] 카오스 원숭이. https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey.

[10] V. 치푸노프, V. 쿠즈네초프, G. 칸데아. S2E 플랫폼: 설계, 구현 및 애플리케이션. *ACM Trans. Comput. Syst.*, 30(1):2:1-2:49, Feb.

[11] Chord: Java용 프로그램 분석 플랫폼. http://pag.gatech.edu/chord.

[12] A. Chou, J. Yang, B. Chelf, S. Hallem 및 D. Engler. 운영 체제 오류에 대한 경험적 연구. *제18회 운영 체제 원칙에 관한 ACM 심포지엄*, SOSP '01, 73-88페이지, 2001.

[13] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. 양, G. A. 깁슨, 및 R. E. 브라이언트. Parrot: 결정론적이고 안정적이며 신뢰할 수 있는 스레드를 위한 실용적인 런타임. *제24회 운영 체제 원칙에 관한 ACM 심포지엄 논문집*, SOSP '13, 388-405쪽, 2013.

[14] J. Dean. Google의 내부: 현재 시스템과 향후 방향. *Google I/O*, 2008.

[15] G. W. 던랩, S. T. 킹, S. 시나르, M. A. 바스라이, 및 P. M. Chen. ReVirt: 가상 머신 로깅 및 재생을 통한 침입 분석 활성화. *운영 체제 설계 및 구현에 관한 제5회 심포지엄*, OSDI'02, 2002.

[16] D. Engler, B. Chelf, A. Chou, 및 S. Hallem. 시스템별 프로그래머 작성 컴파일러 확장을 사용하여 시스템 규칙 확인. *제4회 컨퍼런스의 절차에서 운영 체제 설계 및 구현에 관한 심포지엄*, OSDI'00, 1-16페이지, 2000.

[17] Facebook: 오늘 서비스 중단에 대한 자세한 내용. https://www.facebook.com/note.php?note_id=431441338919&id=9445547199&ref=mf.

[18] Hadoop 팀. 결함 주입 프레임워크: 사용 방법, 인공 결함을 사용하여 테스트하고 새로운 결함을 개발하는 방법. http://wiki.apache.org/hadoop/HowToUseInjectionFramework.

[19] N. 페스터스터와 H. 발라크리쉬난. 정적 분석을 통한 BGP 구성 결함 탐지. *네트워크 시스템 디버깅에 관한 제2회 USENIX 심포지엄 논문집. 서명 및 구현*, NSDI'05, 2005.

[20] C. 푸와 G. B. 라이더. 예외 체인 분석: 자바 서버 응용 프로그램에서 예외 처리 아키텍처 재검토. *제29회 국제 소프트웨어 공학 컨퍼런스*, ICSE'07, 230-239 페이지, 2007.

[21] 구글 서비스 중단으로 인해 전 세계 트래픽이 크게 감소한 것으로 알려졌습니다. http://www.cnet.com/news/google-중단으로 인한 글로벌 트래픽 급감/.

[22] J. Gray. 컴퓨터가 중지되는 이유와 수행 할 수있는 작업 그것에 대해? *분산 소프트웨어 및 데이터베이스 시스템의 신뢰성에 관한 심포지엄 논문집*, 1986.

[23] H. S. 구나위, T. 도, P. 조시, P. 알바로, J. M. 헬러스타인, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, 및 D. Borthakur. FATE와 DESTINI: 클라우드 복구 테스트를 위한 프레임워크. *네트워크 시스템 설계 및 구현에 관한 제8회 USENIX 컨퍼런스*, NSDI'11, 2011.

[24] H. S. 구나위, C. 루비오-곤자레즈, A. C. 아르파치-두소, R. H. 아르파치-두시 및 B. 리블리트. EIO: 오류 처리는 때때로 정확합니다. *파일 및 스토리지 기술에 관한 제6회 USENIX 컨퍼런스*, FAST'08, 2008.

[25] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang. 동적 인터페이스 축소를 통한 실용적인 소프트웨어 모델 검사. *운영 체제 원칙에 관한 제23회 ACM 심포지엄 논문집*, 265-278페이지, 2011년 10월.

[26] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. 카쇼크, Z. 장. R2: 기록 및 재생을 위한 애플리케이션 수준 커널. *제8회 운영 체제 설계 및 구현에 관한 USENIX 컨퍼런스*, OSDI'08, 193-208페이지, 미국 버클리, 캘리포니아, 2008. USENIX 협회.

[27] Hadoop 분산 파일 시스템(HDFS) 아키텍처 가이드. http://hadoop.apache.org/docs/stable/hdfs_design.html.

[28] Hadoop MapReduce. http://hadoop.apache.org/docs/stable/mapred_tutorial.html.

[29] 2018년 하둡 시장 규모는 전 세계적으로 209억 달러에 달할 것으로 예상됩니다. http://www.prnewswire.com/news-releases/hadoop-market-is-expected-in-2018-transparency-market-research-217735621.html.

전 세계적으로 미화 209억 달러에 도달했습니다.

[30] 와이드 컬럼 저장소의 DB 엔진 순위. http://db-engines.com/en/ranking/wide+column+store.

[31] HBase 버그 보고서 10452 - 예외 한들러의 버그 수정. https://issues.apache.org/jira/browse/HBASE-10452.

[32] D. 호브마이어와 W. 퓨. 버그를 찾는 것은 쉽습니다. *SIGPLAN 공지*, 39(12):92-106, 2004.

[33] S. 칸둘라, R. 마하잔, P. 베르카익, S. 아가르왈, J. 패드혜, P. 발. 엔터프라이즈 네트워크에서 상세한 진단. *ACM SIGCOMM 2009 학술대회 논문집*, SIGCOMM '09, 243-254페이지, 2009.

[34] C. Killian, J. W. Anderson, R. Jhala, 및 A. Vahdat. 삶, 죽음, 그리고 중요한 전환: 시스템 코드에서 생명력 있는 버그 찾기. *네트워크 시스템 설계 및 구현에 관한 제4회 심포지엄 자료집*, 243-256페이지, 2007년 4월.

[35] O. 라단, N. 비에노트, J. 니에. 상품에 대한 투명하고 가벼운 애플리케이션 실행 리플레이

멀티프로세서 운영 체제. *컴퓨터 시스템의 보증 및 모델링에 관한 ACM SIGMETRICS 국제 컨퍼런스*, SIGMET- RICS '10, 155-166페이지, 2010.

[36] J.-C. Laprie. 신뢰할 수 있는 컴퓨팅: 개념, 한계, 도전 과제. *제25회 내결함성 컴퓨팅에 관한 국제 컨퍼런스*, FTCS'95, 42-54페이지, 1995.

[37] T. 리사타폰윙사, M. 하오, P. 조시, J. F. 루크만, H. S. 구나위. Samc: 클라우드 시스템에서 심층 버그를 빠르게 발견하기 위한 시맨틱 인식 모델 검사. In *제11회 운영 시스템 설계 및 구현에 관한 USENIX 심포지엄*, OSDI'14, 2014.

[38] S. Li, T. Xiao, H. Zhou, H. Lin, H. Lin, W. Lin, 및 T. Xie. 프로덕션 분산 데이터 병렬 프로그램의 장애에 대한 특성 연구. *국제 소프트웨어 공학 컨퍼런스 (ICSE 2013), 소프트웨어 공학 실무(SEIP) 트랙*, 2013년 5월.

[39] Z. 리, S. 루, S. 미그마르, 및 Y. 저우. Cp-miner: A 운영 체제 코드에서 복사-붙여넣기 및 관련 버그를 찾기 위한 도구. *제6회 운영 체제 설계 및 구현에 관한 심포지엄*, OSDI'04 *컨퍼런스 자료집*, 2004.

[40] G. C. 로렌조 켈러, 프라상 우파디야야. ConfErr: A 인적 구성 오류에 대한 복원력을 평가하기 위한 도구. *Proceedings International Conference on De- pendable Systems and Networks*, DSN'08, 2008.

[41] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, 및 S. Lu. 리눅스 파일 시스템 진화에 대한 연구. *파일 및 스토리지 기술에 관한 제11회 USENIX 컨퍼런스*, FAST'13, 31-44페이지, 2013.

[42] S. Lu, S. Park, E. Seo, Y. Zhou. 실수로부터 배우기: 실제 동시성 버그 특성에 대한 포괄적인 연구. *제13회 국제 동시성 버그 학회 논문집*. *프로그래밍 언어 및 운영 체제를 위한 아키텍처 지원에 관한 국제 컨퍼런스*, ASPLOS'08, 329-339페이지, 2008.

[43] R. Mahajan, D. Wetherall, 및 T. Anderson. BGP 구성 오류 이해하기. *ACM SIGCOMM 2002 컨퍼런스 논문집*, SIGCOMM '02, 3페이지-. 16, 2002.

[44] P. D. Marinescu, R. Banabic, 및 G. Candea. 복구 코드의 고정밀 테스트를 위한 텐서블 기법. *2010 USENIX 연례 기술 컨퍼런스*, USENIX ATC'10, 2010의 *프로시딩에서*.

[45] P. D. Marinescu와 G. Candea. 결함 주입을 사용한 재 커버리지 코드의 효율적인 테스트. *ACM Trans. Comput. 시스템*, 29(4):11:1-11:38, Dec.

[46] 누락된 점으로 스웨덴이 인터넷에서 사라집니다. `http://www.networkworld.com/communi TY/NODE/46115`.

[47] P. 몬테시노스, L. 세제, 및 J. 토렐라스. De- lorean: 공유 메모리 멀티프로세서 실행을 효율적으로 기록하고 결정론적으로 재생. *제35회 연례 국제 심포지엄 진행*중. *컴퓨터 아키텍처*, ISCA '08, 289-300페이지, 미국 워싱턴 DC, 2008. IEEE 컴퓨터 학회.

[48] K. 나가라즈, C. 킬리안, 및 J. 네빌. 성능 진단을 위한 시스템 로그의 구조화된 비교 분석 문제. *네트워크 시스템 설계 및 구현에 관한 제9회 USENIX 컨퍼런스의 프로시딩*, NSDI'12, 2012.

[49] K. 나가라자, F. 올리베이라, R. 비안치니, R. P. 마틴, 및 T. D. 응우옌. 인터넷 서비스에서 운영자의 실수에 대한 이해와 대처. *제6회 Opearting 시스템 설계 및 구현 심포지엄*, OSDI'04, 2004 *회의 자료집*.

[50] D. 오펜하이머, A. 가나파티, 및 D. A. 패터슨. 인터넷 서비스가 실패하는 이유는 무엇이며, 이를 어떻게 해결할 수 있을까요? *인터넷 기술 및 시스템에 관한 USENIX 심포지엄 제4회 컨퍼런스*, USITS'03, 1-15페이지, 2003.

[51] N. 팔릭스, G. 토마스, S. 사하, C. 칼브스, J. 로올, 및 G. 뮬러. Linux의 결함 : 10 년 후. *제16회 국제 아키텍텍 국제 컨퍼런스 진행 중*. *프로그래밍 언어 및 운영 체제에 대한 교육 지원*, ASPLOS '11, 305-318페이지, 2011.

[52] A. Rabkin과 R. Katz. 하둡 클러스터가 깨지는 방법. *Soft-웨어, IEEE*, 30(4):88-94, 2013.

[53] 주요 가치 스토어에 대한 DB-Engines 순위. `http://db- engines.com/ko/ranking/key-value+store`.

[54] Redis: 오픈 소스 고급 키-값 저장소. `http://redis.io/`.

[55] C. Rubio-Gonza´lez, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, 및 A. C. Arpaci-Dusseau. 파일 시스템에 대한 오류 프로포절 분석. *2009 ACM SIGPLAN 프로그래밍 언어 설계 및 구현 컨퍼런스*, PLDI '09, 270-280 페이지, 2009.

[56] B. 슈뢰더와 G. 깁슨. 고성능 컴퓨팅 시스템의 장애에 대한 대규모 연구. *IEEE 트랜잭션: 신뢰할 수 있고 안전한 컴퓨팅*, 7(4):337-350, 2010.

[57] C. Spatz. 기본 통계, 1981.

[58] M. 설리반과 R. 칠라레지. 소프트웨어 결함과 시스템 가용성에 미치는 영향 - 운영 체제의 현장 장애에 대한 연구. *제21회 국제 내결함성 컴퓨팅 심포지엄*, FTCS'91, 2-9페이지, 1991.

[59] Amazon EC2 및 RDS 서비스 중단에 대한 요약. http://aws.`amazon.com/message /65648/`.

[60] 삼킨 예외의 저주 `http://mi chaelscharf.blogspot.ca/2006/09/dont-` 삼키기 중단 예외 호출`.html`.

[61] K. 비라라가반, D. 리, B. 웨스터, J. 오우양, P. 엠 첸, J. 플린, S. 나라야나사미. DoublePlay: 순차 로깅과 리플레이의 병렬화. *제16회 국제 건축 슈퍼컴퓨팅 학술대회 논문집*. *프로그래밍 언어 및 운영 체제용 포트*, ASPLOS '11, 2011.

[62] K. V. 비슈와나트 및 N. 나가판. 클라우드 특성화 컴퓨팅 하드웨어 신뢰성. *클라우드 컴퓨팅에 관한 제1회 ACM 심포지엄*, SoCC '10, 193-204페이지, 미국 뉴욕, 2010. ACM.

[63] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, 및 S. 파수파시. 잘못된 구성에 대해 사용자를 탓하지 마세요. *제24회 ACM 심포지엄*논문집에서

운영 체제 원칙에 관한 sium, SOSP '13, 2013.

[64] W. Xu, L. Huang, A. Fox, D. Patterson, 및 M. I. Jordan. 콘솔 로그 마이닝을 통한 대규모 시스템 문제 탐지. *ACM SIGOPS 제22회 운영 체제 원칙에 관한 심포지엄*, SOSP '09, 117-132쪽, 2009, *Proceedings of the ACM SIGOPS 22nd Sympo- sium on Operating Systems Principles.*

[65] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, 및 L. Zhou. MODIST: 수정되지 않은 분산 시스템의 투명 모델 검사. *제6회 네트워크 시스템 설계 및 구현 심포지엄(NSDI '09)*, 213~228페이지, 2009년 4월.

[66] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, 및 S. Pasupathy. 상용 및 오픈 소스 시스템의 구성 오류에 대한 경험적 연구. *운영 체제 원칙에 관한 제23회 ACM 심포지엄*, SOSP '11, 159-172페이지, 2011.

[67] D. Yuan, S. Park, P. Huang, Y. Liu, M. Lee, Y. Zhou, 및 S. Savage. 보수적이어야 합니다: 사전 예방적 로깅으로 장애 감지 능력 향상. *제10회 운영 체제 설계 및 구현에 관한 USENIX 심포지엄*, OSDI'12, 293-306페이지, 2012.