AWS 백서

가용성과 그 이상: AWS 분산 시스템의 복원력에 대한 이해 및 개선

가용성과 그 이상: AWS 분산 시스템의 복원력에 대한 이해 및 개선: AWS 백서

Table of Contents

가용성과 그 이상: AWS 분산 시스템의 복원력에 대한 이해 및 개선

발행일: 2021년 11월 12일(문서 이력)

오늘날 기업들은 클라우드와 온프레미스 모두에서 복잡한 분산 시스템을 운영합니다. 그들은 이러한 워크로드가 탄력적이어서 고객에게 서비스를 제공하고 비즈니스 성과를 달성하기를 원합니다. 이 백서에서는 복원력의 척도로서 가용성에 대한 일반적인 이해를 설명하고, 고가용성 워크로드를 구축하기 위한 규칙을 설정하고, 워크로드 가용성을 개선하는 방법에 대한 지침을 제공합니다.

소개

가용성이 높은 워크로드를 구축한다는 것은 무엇을 의미할까요? 가용성을 어떻게 측정합니까? 워크로드의 가용성을 높이려면 어떻게 해야 하나요? 이 문서는 이러한 질문에 답하는 데 도움이 됩니다. 이 문서는 세 개의 주요 항목으로 구성되어 있습니다. 첫 번째 항목인 가용성에 대한 이해는 대체로 이론적인 내용입니다. 이를 통해 가용성의 정의와 가용성에 영향을 미치는 요인을 기초적으로 이해할 수 있습니다. 두 번째 항목인 가용성 측정에서는 워크로드의 가용성을 경험적으로 측정하는 방법에 대한 지침을 제공합니다. 세 번째 항목인 AWS 기반 고가용성 분산 시스템 설계는 첫 번째 항목에서 제시한 아이디어를 실제로 적용한 것입니다. 또한 이 백서는 항목 전반에 걸쳐 이 백서에서는 복원력이 뛰어난 워크로드를 구축하기 위한 자격 증명 규칙을 설명합니다. 이 문서는 AWS Well-Architected 신뢰성 요소에 제시된 지침과 모범 사례를 지원하기 위한 것입니다.

이 백서에서 여러분은 대수 수학을 많이 접하게 될 것입니다. 중요한 점은 수학 자체가 아니라 그 수학이 뒷받침하는 개념입니다. 이 백서의 또 다른 목적은 독자에게 과제를 제시하는 것입니다. 가용성이 높은 워크로드를 운영할 때는 구축한 것이 의도한 바를 달성하고 있음을 수학적으로 증명할 수 있어야 합니다. 좋은 의도에 기반한 최상의 설계라도 원하는 결과를 일관되게 달성하지 못할 수 있습니다. 즉, 솔루션의 효과를 측정하는 메커니즘이 필요하므로 복원력이 뛰어나고 가용성이 높은 분산 시스템을 구축하고 운영하려면 일정 수준의 수학이 필요합니다.

가용성에 대한 이해

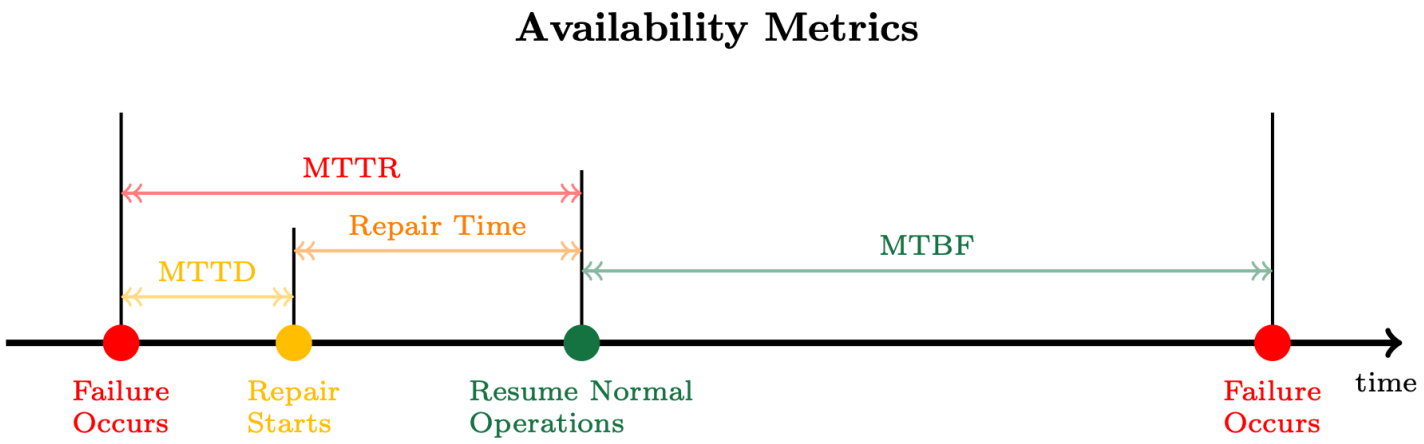
가용성은 복원력을 정량적으로 측정할 수 있는 주요 방법 중 하나입니다. A로 표시하는 가용성은 워크로드를 사용할 수 있는 시간의 백분율로 정의합니다. 가용성은 측정되는 총 시간(예상 “가동 시간”과 예상 “가동 중지 시간”)에 대한 예상 “가동 시간"(사용 가능)의 비율입니다.

$$A = \frac{uptime}{uptime + downtime}$$

수식 1: 가용성

이 공식을 더 잘 이해하기 위해 가동 시간과 가동 중지 시간을 측정하는 방법을 살펴보겠습니다. 먼저 워크로드가 고장 없이 얼마나 오래 지속되는지 알고 싶습니다. 이를 평균 고장 간격(MTBF)이라고 하며, 워크로드가 정상적으로 작동하기 시작한 시점과 다음 고장 시점 사이의 평균 시간을 의미합니다. 그런 다음 고장이 발생한 후 복구하는 데 시간이 얼마나 걸릴지 알고 싶습니다.

이를 평균 수리(또는 복구) 시간(MTTR)이라고 하며, 고장이 발생한 하위 시스템을 수리하거나 서비스 상태로 복원하는 동안 워크로드를 사용할 수 없는 기간을 MTTR(평균 복구 시간)이라고 합니다. MTTR에서 중요한 기간은 평균 탐지 시간(MTTD), 즉 고장 발생 시점과 수리 작업 시작 시점 사이의 시간입니다. 다음 다이어그램은 이러한 모든 지표가 어떻게 관련되어 있는지를 보여줍니다.



MTTD, MTTR 및 MTBF 간의 관계

따라서 워크로드가 가동되는 시간인 MTBF와 워크로드가 중단된 시간인 MTTR을 사용하여 가용성 A를 표현할 수 있습니다.

$$A = \frac{MTBF}{MTBF + MTTR}$$

수식 2: MTBF와 MTTR 간의 관계

그리고 워크로드가 “다운(사용할 수 없음)”될 확률은 고장 확률 F로 표시합니다.

$$F = 1 - A$$

수식 3: 고장 확률

신뢰성이란 요청 시 지정된 응답 시간 내에 워크로드가 올바른 작업을 수행할 수 있는 능력을 말합니다. 이것이 가용성을 측정하는 척도입니다. 워크로드 고장 빈도를 줄이거나(MTBF 연장) 복구 시간이 짧을수록(MTTR 단축) 가용성이 향상됩니다.

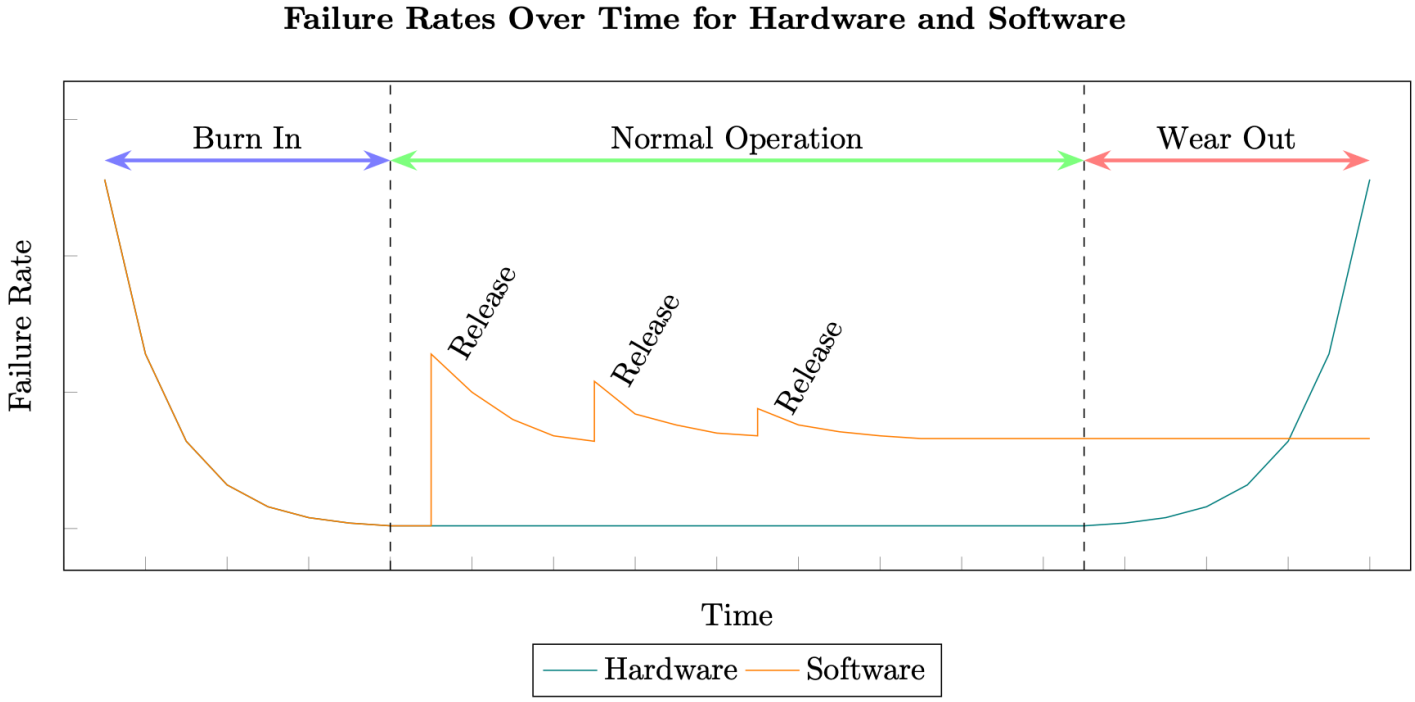
> 규칙 1
>
> 고장 빈도 감소(MTBF 연장), 고장 감지 시간 단축(MTTD 단축), 수리 시간 단축(MTTR 단축)은 분산 시스템에서 가용성을 개선하는 데 사용되는 세 가지 요소입니다.

주제

분산 시스템 가용성

분산 시스템은 소프트웨어 구성 요소와 하드웨어 구성 요소로 구성됩니다. 일부 소프트웨어 구성 요소는 그 자체가 또 다른 분산 시스템일 수 있습니다. 기본 하드웨어와 소프트웨어 구성 요소 모두의 가용성은 결과적으로 워크로드의 가용성에 영향을 미칩니다.

MTBF와 MTTR을 사용한 가용성 계산은 하드웨어 시스템에 그 뿌리를 두고 있습니다. 하지만 분산 시스템이 실패하는 이유는 하드웨어와는 매우 다릅니다. 제조업체가 하드웨어 구성 요소가 마모되기까지 걸리는 평균 시간을 일관되게 계산할 수 있는 경우 분산 시스템의 소프트웨어 구성 요소에는 동일한 테스트를 적용할 수 없습니다. 하드웨어는 일반적으로 고장률의 "욕조" 곡선을 따르는 반면, 소프트웨어는 새 릴리즈마다 발생하는 추가 결함으로 인해 생기는 엇갈린 곡선을 따릅니다(소프트웨어 신뢰성 참조).



하드웨어 및 소프트웨어 고장률

또한 분산 시스템의 소프트웨어는 일반적으로 하드웨어보다 기하급수적으로 높은 속도로 변경됩니다. 예를 들어, 표준 마그네틱 하드 드라이브의 연간 평균 고장률(AFR)은 0.93%일 수 있으며, 이는 HDD의 실제 수명이 마모 기간에 도달하기까지 최소 3~5년이 걸릴 수 있으며, 이는 더 길어질 수 있습니다(Backblaze 하드 드라이브 데이터 및 통계, 2020 참조). 예를 들어 Amazon은 3~5년 내에 소프트웨어 시스템에 4억 5천만~7억 5천만 개 이상의 변경 사항을 배포할 수 있는 수명 기간 동안 하드 드라이브는 크게 변경되지 않습니다. (Amazon Builders' Library: 안전한 수동 배포 자동화를 참조하세요.)

또한 하드웨어에는 계획된 노후화 개념이 적용됩니다. 즉, 수명이 내장되어 있으며 일정 기간이 지나면 교체해야 합니다. (위대한 전구 음모 참조) 이론적으로 소프트웨어는 이러한 제약의 적용을 받지 않으며, 마모 기간이 없으며 무기한으로 작동할 수 있습니다.

즉, 하드웨어에서 MTBF 및 MTTR 번호를 생성하는 데 사용하는 것과 동일한 테스트 및 예측 모델이 소프트웨어에는 적용되지 않습니다. 1970년대 이후 이 문제를 해결하기 위해 모델을 구축하려는 시도가

수백 번 있었지만, 모두 일반적으로 예측 모델링과 추정 모델링이라는 두 가지 범주로 나뉩니다. (소프트웨어 신뢰성 모델 목록 참조)

따라서 분산 시스템에 대한 미래 예측 MTBF 및 MTTR 계산, 따라서 미래 예측 가용성은 항상 특정 유형의 예측 또는 예보에서 도출됩니다. 예측 모델링, 확률적 시뮬레이션, 과거 분석 또는 엄격한 테스트를 통해 생성될 수 있지만 이러한 계산이 가동 시간이나 가동 중지 시간을 보장하지는 않습니다.

과거에 분산 시스템이 고장 났던 이유는 다시 발생하지 않을 수 있습니다. 미래에 고장 나는 이유는 다를 수 있으며 아마도 알 수 없을 것입니다. 향후 고장 발생 시 필요한 복구 메커니즘은 과거에 사용된 메커니즘과 다를 수 있으며 소요 시간도 크게 다를 수 있습니다.

또한 MTBF와 MTTR은 평균값입니다. 평균값과 실제 값 사이에 약간의 차이가 있을 수 있습니다(이 변동을 측정한 표준 편차 γ는 이 변동을 측정합니다). 따라서 실제 프로덕션 사용에서는 워크로드에 고장이 발생한 후 복구 시간이 걸리는 시간이 짧거나 길어질 수 있습니다.

그렇긴 하지만 분산 시스템을 구성하는 소프트웨어 구성 요소의 가용성은 여전히 중요합니다. 소프트웨어는 여러 가지 이유로 장애가 발생할 수 있으며(다음 항목에서 자세히 설명) 워크로드의 가용성에 영향을 미칠 수 있습니다. 따라서 가용성이 높은 분산 시스템의 경우 하드웨어 및 외부 소프트웨어 하위 시스템과 마찬가지로 소프트웨어 구성 요소의 가용성을 계산, 측정 및 개선하는 데 중점을 두어야 합니다.

> 규칙 2
>
> 워크로드의 소프트웨어 가용성은 워크로드의 전체 가용성을 결정하는 중요한 요소이므로 다른 구성 요소와 마찬가지로 중점을 두어야 합니다.

분산 시스템에서 MTBF와 MTTR은 예측하기 어렵지만 여전히 가용성 개선 방법에 대한 주요 통찰력을 제공한다는 점에 유의해야 합니다. 고장 빈도를 줄이면(MTBF가 높음), 고장 발생 후 복구 시간을 줄이면(MTTR이 낮아짐) 모두 경험적 가용성을 높일 수 있습니다.

분산 시스템의 고장 유형

분산 시스템에는 일반적으로 보어버그와 하이젠버그라는 애칭으로 불리는 두 종류의 버그가 있습니다 (“브루스 린지와의 대화”, ACM Queue vol. 2, No. 8 - 2004년 11월 참조).

보어버그는 반복해서 작동하는 소프트웨어 문제입니다. 동일한 입력이 주어지면 버그는 일관되게 동일한 잘못된 출력을 생성합니다(예: 확실하고 쉽게 감지할 수 있는 결정론적 보어 원자 모델). 워크로드가 프로덕션에 들어갈 때쯤이면 이런 유형의 버그는 거의 발생하지 않습니다.

하이젠버그는 일시적인 버그입니다. 즉, 특정하고 흔하지 않은 상황에서만 발생합니다. 이러한 조건은 일반적으로 하드웨어(예: 일시적인 장치 장애 또는 레지스터 크기와 같은 하드웨어 구현 세부 사항), 컴파일러 최적화 및 언어 구현, 제한 조건(예: 일시적으로 스토리지 부족) 또는 경쟁 조건(예: 다중 스레드 작업에 세마포어를 사용하지 않음)와 관련이 있습니다.

하이젠버그는 프로덕션 환경에서 발생하는 버그의 대부분을 차지하며 관찰하거나 디버깅하려고 할 때 동작이 바뀌거나 사라지는 것처럼 보이기 때문에 찾기가 어렵습니다. 그러나 프로그램을 다시 시작하면 운영 환경이 약간 달라서 하이젠버그를 발생시킨 조건이 사라지기 때문에 고장 난 작업이 성공할 가능성이 높습니다.

따라서 대부분의 프로덕션 실패는 일시적이며 작업을 다시 시도해도 다시 고장 날 가능성은 거의 없습니다. 복원력을 갖추려면 분산 시스템이 하이젠버그에 대한 내결함성을 갖추어야 합니다. 분산 시스템 MTBF 늘리기 항목에서 이를 실현하는 방법을 살펴보겠습니다.

종속성이 있는 가용성

이전 항목에서는 하드웨어, 소프트웨어 및 기타 분산 시스템이 모두 워크로드의 구성 요소라고 언급했습니다. 이러한 구성 요소를 종속성이라고 하며, 이러한 구성 요소는 워크로드가 기능을 제공하기 위해 의존하는 요소입니다. 여기에는 워크로드가 제대로 작동하지 않는 하드 종속성과 일정 기간 동안 가용성이 눈에 띄지 않거나 용인될 수 있는 소프트 종속성이 있습니다. 하드 종속성은 워크로드의 가용성에 직접적인 영향을 미칩니다.

워크로드의 이론상 최대 가용성을 계산해 보고 싶을 수도 있습니다. 이는 소프트웨어 자체를 포함한 모든 종속성(α_n는 단일 하위 시스템의 가용성)의 곱입니다. 각 종속성이 작동해야 하기 때문입니다.

$$A = \alpha_1 \times \alpha_2 \times \ldots \times \alpha_n$$

수식 4: 이론적 최대 가용성

이러한 계산에 사용되는 가용성 수치는 일반적으로 SLA 또는 서비스 수준 목표(SLO)와 같은 항목과 연관됩니다. SLA는 고객이 받게 될 예상 서비스 수준, 서비스를 평가하는 기준, 서비스 수준을 달성하지 못할 경우 조치 또는 벌금(일반적으로 금전적)을 정의합니다.

위 공식을 사용하면 순전히 수학적으로 볼 때 워크로드는 종속 항목보다 가용성이 높을 수 없다는 결론을 내릴 수 있습니다. 하지만 현실에서는 일반적으로 그렇지 않다는 것을 알 수 있습니다. 99.99% 가용성 SLA와 함께 2~3개의 종속성을 사용하여 구축된 워크로드는 여전히 자체적으로 99.99% 또는 그 이상의 가용성을 달성할 수 있습니다.

이것은 이전 항목에서 설명했듯이 이러한 가용성 수치는 추정치이기 때문입니다. 그것은 고장이 발생하는 빈도와 수리 속도를 추정하거나 예측합니다. 그렇다고 가동 중지를 보장하지는 않습니다. 종속성은 명시된 가용성 SLA 또는 SLO를 초과하는 경우가 많습니다.

또한 종속성은 공개 SLA에 제공된 수치보다 성능을 목표로 하는 내부 가용성 목표가 더 높을 수 있습니다. 이를 통해 미지 또는 불가지의 상황이 발생했을 때 SLA를 준수하는 데 따르는 위험을 일정 수준 완화할 수 있습니다.

마지막으로, 워크로드에는 SLA를 알 수 없거나 SLA 또는 SLO를 제공하지 않는 종속성이 있을 수 있습니다. 예를 들어 전 세계 인터넷 라우팅은 많은 워크로드의 공통적인 종속성이지만 글로벌 트래픽이 어떤 인터넷 서비스 공급자를 사용하고 있는지, SLA가 있는지, 공급업체 간에 얼마나 일관성이 있는지 파악하기가 어렵습니다.

이 모든 것이 시사하는 바는 이론상의 최대 가용성을 계산하는 것은 대략적인 규모만 계산할 수 있을 뿐 그 자체로는 정확하지 않거나 의미 있는 통찰력을 제공하지 못할 가능성이 높다는 것입니다. 계산을 통해 알 수 있는 것은 워크로드가 의존하는 대상이 적을수록 전반적인 실패 가능성이 줄어든다는 것입니다. 1보다 작은 수의 숫자를 적게 곱할수록 결과는 더 커집니다.

> 규칙 3
>
> 종속성을 줄이면 가용성에 긍정적인 영향을 미칠 수 있습니다.

수학은 종속성 선택 프로세스에 정보를 제공하는 데도 도움이 됩니다. 선택 프로세스는 워크로드를 설계하는 방법, 종속성의 중복성을 활용하여 가용성을 개선하는 방법, 이러한 종속성을 소프트 종속성으로 간주할지 하드 종속성으로 간주할지 여부에 영향을 줍니다. 워크로드에 영향을 미칠 수 있는 종속성은 신중하게 선택해야 합니다. 다음 규칙은 이 작업을 수행하는 방법에 대한 지침을 제공합니다.

> 규칙 4
>
> 일반적으로 가용성 목표가 워크로드 목표와 같거나 더 큰 종속성을 선택하세요.

중복성이 있는 가용성

워크로드가 여러 독립적이고 중복된 하위 시스템을 사용하는 경우 단일 하위 시스템을 사용하는 경우보다 이론적으로 더 높은 수준의 가용성을 달성할 수 있습니다. 예를 들어 두 개의 동일한 하위 시스템으로 구성된 워크로드를 고려하세요. 하위 시스템 1이나 하위 시스템 2가 작동 중이면 완전히 작동할 수 있습니다. 전체 시스템이 다운되려면 두 하위 시스템이 동시에 다운되어야 합니다.

한 하위 시스템의 고장 확률이 1 - α인 경우 두 개의 중복 하위 시스템이 동시에 다운될 확률은 각 하위 시스템의 고장 확률의 곱, $F = (1-\alpha_1) \times (1- \alpha_2)$입니다. 두 개의 중복 하위 시스템이 있는 워크로드의 경우 방정식 (3)을 사용하면 다음과 같이 정의된 가용성이 제공됩니다.

$$A = 1 - F$$
$$F = (1 - \alpha_1) \times (1 - \alpha_2)$$
$$A = 1 - (1 - \alpha)^2$$

방정식 5

따라서 가용성이 99%인 두 하위 시스템의 경우 한 하위 시스템에 고장이 발생할 확률은 1%이고 둘 다 고장 날 확률은 (1-99%) × (1-99%) = 0.01%입니다. 따라서 두 개의 중복 하위 시스템을 사용할 경우 가용성이 99.99% 향상됩니다.

이를 일반화하여 추가 중복 스페어, s,도 통합할 수 있습니다. 수식 (5)에서는 스페어를 하나만 가정했지만, 워크로드에는 스페어가 2개, 3개 또는 그 이상 있을 수 있으므로 여러 하위 시스템이 동시에 손실되어도 가용성에 영향을 주지 않고 버틸 수 있습니다. 워크로드에 세 개의 하위 시스템이 있고 두 개가 스페어 시스템인 경우 세 개의 하위 시스템이 모두 동시에 실패할 확률은 (1−α) × (1−α) × (1−α) or $(1-\alpha)^3$입니다. 일반적으로 s개의 스페어가 있는 워크로드는 s + 1개의 하위 시스템에 고장이 발생하는 경우에만 고장 납니다.

n개의 하위 시스템과 s개의 스페어가 있는 워크로드의 경우 f는 고장 모드의 수 또는 s + 1개의 하위 시스템이 n개 중 고장을 일으킬 수 있는 방법을 나타냅니다.

이것은 사실상 이항 정리로, n개의 집합에서 k개의 요소를 선택하거나 “n은 k를 선택한다”는 조합 수학입니다. 이 경우 k는 s + 1입니다.

$$k = s + 1$$

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

$$\binom{n}{s+1} = \frac{n!}{(s+1)!\,\big(n-(s+1)\big)!}$$

$$f = \frac{n!}{(s+1)!\,(n-s-1)!}$$

방정식 6

그런 다음 고장 모드와 스페어링 수를 포함하는 일반화된 가용성 근사치를 산출할 수 있습니다. (근사치로 이러한 이유를 이해하려면 Highleyman 등의 부록 2를 참조하세요. 가용성 장벽을 허물다.)

$$s = \ Number\ of\ spares$$
$$\alpha = Availability\ of\ subcomponent$$
$$f = Number\ of\ failure\ modes$$
$$A = 1 - F \approx 1 - f(1-\alpha)^{s+1}$$

방정식 7

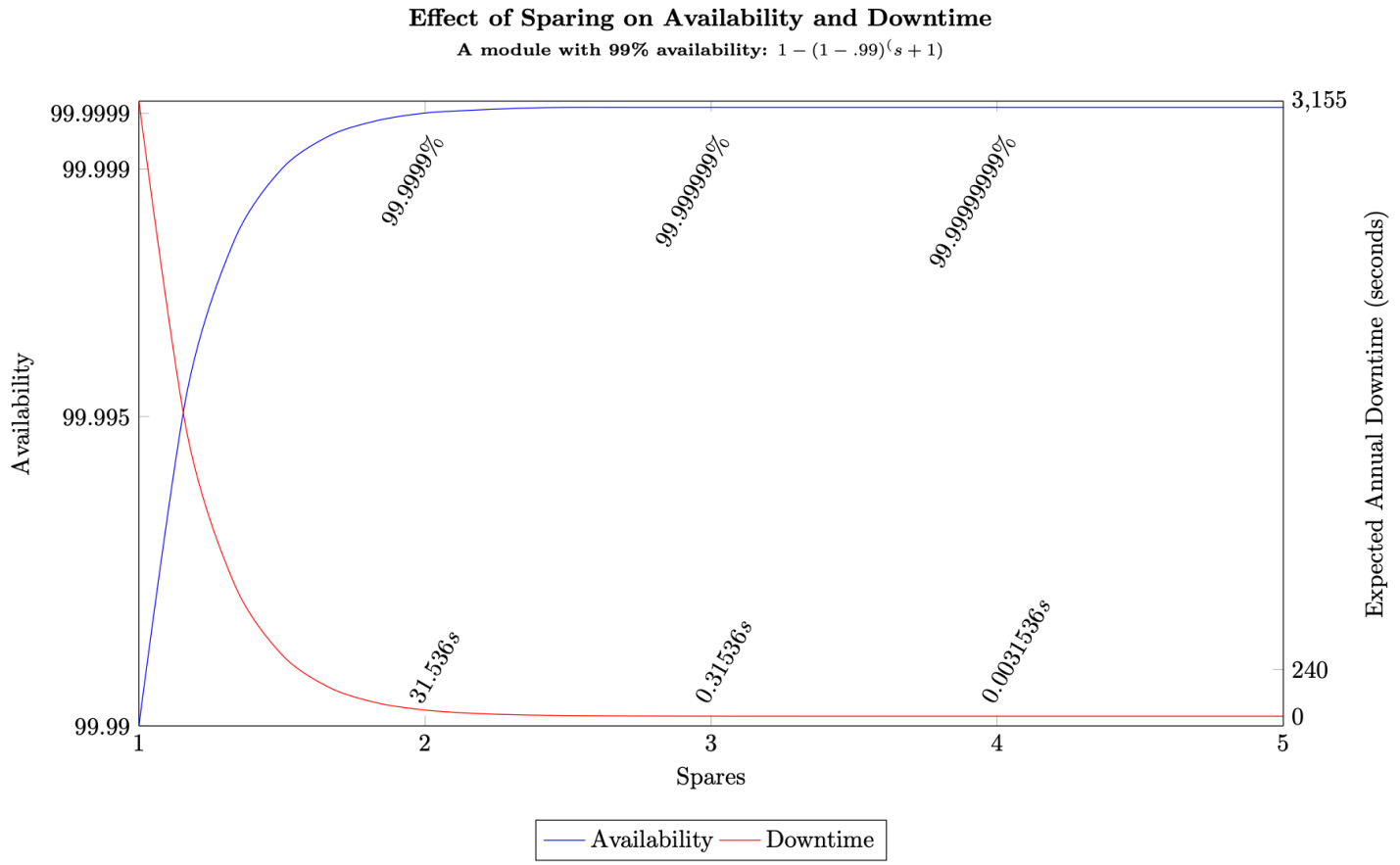
스페어링은 독립적으로 고장이 발생하는 리소스를 제공하는 모든 종속성에 적용할 수 있습니다. 그 예로는 여러 AZ에 있는 Amazon EC2 인스턴스, 다른 AWS 리전에 있는 Amazon S3 버킷이 포함됩니다. 스페어를 사용하면 종속 항목이 총 가용성을 높여 워크로드의 가용성 목표를 지원하는 데 도움이 됩니다.

> 규칙 5
>
> 스페어링을 사용하여 워크로드의 종속성 가용성을 높이세요.

하지만 스페어링에는 대가가 따릅니다. 예비 부품을 추가할 때마다 원래 모듈과 비용이 동일하므로 비용이 최소한 선형적으로 늘어납니다. 스페어를 사용할 수 있는 워크로드를 구축하면 복잡성도 증가합니다. 종속성 고장을 식별하고, 적절한 리소스로 작업량을 줄이고, 워크로드의 전체 용량을 관리하는 방법을 알아야 합니다.

중복성은 최적화 문제입니다. 스페어가 너무 적으면 워크로드가 원하는 것보다 더 자주 고장 나고, 스페어가 너무 많으면 워크로드 실행 비용이 너무 많이 들 수 있습니다. 스페어를 더 추가할 경우 보증된 추가 가용성보다 비용이 더 많이 드는 임계값이 있습니다.

스페어 사용 일반 가용성 공식인 수식 (7) 을 사용하면 가용성이 99.5% 이고 스페어가 두 개 있는 서브시스템의 경우 워크로드 가용성은 $A \approx 1 - (1)(1-.995)^3 = 99.9999875\%$(연간 약 3.94초의 가동 중지)고, 스페어 10개를 사용할 경우 $A \approx 1 - (1)(1-.995)^{11} = 25.5$ 9′s가 됩니다(대략적인 가동 중지는 연간 1.26252×10^{-15}ms이며, 사실상 0입니다). 이 두 워크로드를 비교한 결과 스페어링 비용이 5배 증가하여 연간 가동 중지를 4초 줄였습니다. 대부분의 워크로드에서 이러한 비용 증가는 가용성 증가에 비해 부당합니다. 다음 그림은 이 관계를 보여줍니다.



스페어링 증가로 인한 수익 감소

예비 부품이 3개 이상이면 연간 예상 가동 중지 시간의 1분의 1분의 1에 불과합니다. 즉, 이 시점 이후에는 수익이 감소하는 영역에 도달하게 됩니다. 더 높은 수준의 가용성을 달성하기 위해 "그냥 추가하고 싶다"는 충동이 들 수도 있지만 실제로는 비용상의 이점이 매우 빠르게 사라집니다. 예비 부품을 3개 이상 사용한다고 해서 실질적인 효과를 얻을 수 있는 것은 아닙니다. 하위 시스템 자체의 가용성이 99% 이상이면 거의 모든 워크로드에서 눈에 띄는 이득을 얻을 수 있습니다.

> **규칙 6**
>
> 스페어링의 비용 효율성에는 상한선이 있습니다. 필요한 가용성을 달성하는 데 필요한 최소한의 스페어를 활용하세요.

올바른 스페어 수를 선택할 때는 고장 단위를 고려해야 합니다. 예를 들어, 최대 용량을 처리하기 위해 EC2 인스턴스 10개가 필요하고 단일 AZ에 배포된 워크로드를 살펴보겠습니다.

AZ는 고장 격리 경계를 설정하도록 설계되었으므로 단일 EC2 인스턴스에만 고장이 발생하는 것은 아닙니다. AZ 전체에 해당하는 EC2 인스턴스가 함께 고장이 발생할 수 있기 때문입니다. 이 경우 다른 AZ와 중복성을 추가하여 AZ 고장 발생 시 부하를 처리할 EC2 인스턴스 10개를 추가로 배포하여 총 20개의 EC2 인스턴스(정적 안정성 패턴 적용)를 구축하는 것이 좋습니다.

이는 10개의 예비 EC2 인스턴스인 것처럼 보이지만 실제로는 단일 스페어 AZ에 불과하므로 수익이 감소하는 지점을 넘지 않았습니다. 하지만 AZ를 3개 활용하고 AZ당 EC2 인스턴스 5개를 배포하면 가용성을 높이는 동시에 비용 효율성을 높일 수 있습니다.

이렇게 하면 1개의 스페어 AZ에 총 15개의 EC2 인스턴스가 제공되지만(20개의 인스턴스가 있는 두 개의 AZ에 비해), 단일 AZ에 영향을 미치는 이벤트 발생 시 최대 용량을 제공하는 데 필요한 총 10개의 인스턴스가 계속 제공됩니다. 따라서 워크로드가 사용하는 모든 장애 격리 경계(인스턴스, 셀, AZ, 지역)에서 내결함성을 갖추도록 스페어링을 구축해야 합니다.

CAP 정리

가용성에 대해 생각할 수 있는 또 다른 방법은 CAP 정리를 이용하는 것입니다. 이론에 따르면 데이터를 저장하는 여러 노드로 구성된 분산 시스템은 다음 세 가지 보장 중 두 가지까지만 동시에 제공할 있습니다.

- 일관성, C: 모든 읽기 요청은 가장 최근의 쓰기 요청을 수신하거나 일관성을 보장할 수 없는 경우 오류가 발생합니다.
- 가용성, A: 모든 요청은 노드가 다운되거나 사용할 수 없는 경우에도 오류 없는 응답을 받습니다.
- 파티션 내성, P: 노드 간에 임의의 수의 메시지가 손실되더라도 시스템은 계속 작동합니다.

(자세한 내용은 Seth Gilbert와 Nancy Lynch, 브루어의 추측과 일관되고 사용 가능하며 파티션을 견딜 수 있는 웹 서비스의 실현 가능성, ACM SIGACT 뉴스, 33권 2호(2002), 51~59페이지를 참조하세요.)

대부분의 분산 시스템은 네트워크 고장을 견뎌야 하므로 네트워크 파티셔닝이 허용되어야 합니다. 즉, 네트워크 파티션이 발생할 경우 이러한 워크로드는 일관성과 가용성 중 하나를 선택해야 합니다. 워크로드가 가용성을 선택하면 항상 응답을 반환하지만 데이터가 일치하지 않을 수 있습니다. 일관성을 선택하면 워크로드가 데이터의 일관성을 확신할 수 없으므로 네트워크 파티션 중에 오류가 반환됩니다.

더 높은 수준의 가용성을 제공하는 것이 목표인 워크로드의 경우 네트워크 파티션 중에 오류(사용할 수 없음)가 반환되는 것을 방지하기 위해 가용성 및 파티션 내성(AP)을 선택할 수 있습니다. 따라서 최종 일관성 또는 단조로운 일관성과 같은 더욱 완화된 정합성 모델이 필요합니다.

내결함성 및 장애 격리

이 두 가지 개념은 가용성을 고려할 때 아주 중요합니다. 내결함성이란 하위 시스템 고장을 견디고 가용성을 유지하는 능력(설정된 SLA 내에서 올바른 작업 수행)을 말합니다. 내결함성을 구현하기 위해 워크로드는 스페어(또는 중복) 하위 시스템을 사용합니다. 중복 세트의 하위 시스템 중 하나에 고장이 발생하면 일반적으로 거의 원활하게 다른 하위 시스템이 작업을 시작합니다. 이 경우 스페어는 진정한 스페어 용량이므로 고장이 발생한 하위 시스템의 작업을 100% 떠맡을 수 있습니다. 진정한 스페어가 있으면 여러 개의 하위 시스템 고장이 발생해야만 워크로드에 부정적인 영향이 미칩니다.

고장 격리는 고장 발생 시 영향 범위를 최소화합니다. 이는 일반적으로 모듈화를 통해 구현됩니다. 워크로드는 고장이 한꺼번에 발생하지 않고, 따라서 개별적으로 복구할 수 있는 작은 하위 시스템들로 나누어집니다. 모듈의 고장은 모듈 외부로 전파되지 않습니다. 이 아이디어는 수직적으로는 워크로드의 다양한 기능에 걸쳐 적용되고 수평적으로는 동일한 기능을 제공하는 여러 하위 시스템에 걸쳐 적용됩니다. 이러한 모듈은 이벤트 중에 미치는 영향 범위를 제한하는 장애 컨테이너 역할을 합니다.

컨트롤 플레인, 데이터 영역 및 정적 안정성의 아키텍처 패턴은 내결함성 및 장애 격리 구현을 직접적으로 지원합니다. Amazon Builders' Library 문서 가용 영역을 사용한 정적 안정성에서는 이러한 용어에 대한 올바른 정의와 복원력 있고 가용성이 높은 워크로드를 구축하는 데 적용하는 방법을 제공합니다. 이 백서에서는 AWS에서 고가용성 분산 시스템 설계 항목에서 이러한 패턴을 사용하며 여기에 해당 정의도 요약되어 있습니다.

- 컨트롤 플레인: 리소스 추가, 리소스 삭제, 리소스 수정, 해당 변경 사항을 필요한 곳으로 전파하는 등 변경과 관련된 워크로드의 일부입니다. 컨트롤 플레인은 일반적으로 데이터 영역보다 더 복잡하고 움직이는 부분이 많기 때문에 통계적으로 실패 가능성이 더 높고 가용성이 낮습니다.
- 데이터 영역: 일상적인 비즈니스 기능을 제공하는 워크로드의 일부입니다. 데이터 영역은 컨트롤 플레인보다 단순하고 대량으로 작동하는 경향이 있어 가용성이 높아집니다.
- 정적 안정성: 종속성 손상에도 불구하고 워크로드가 올바른 작동을 계속할 수 있는 능력입니다. 구현 방법 중 하나는 데이터 영역에서 컨트롤 플레인 종속성을 제거하는 것입니다. 또 다른 방법은 워크로드 종속성을 느슨하게 결합하는 것입니다. 종속 항목이 전달해야 하는 업데이트된 정보(예: 새 항목, 삭제된 항목, 수정된 항목)가 워크로드에 표시되지 않을 수 있습니다. 하지만 종속성이 손상되기 전에 수행했던 모든 작업은 계속 작동합니다.

워크로드 장애를 생각할 때 복구를 위해 고려할 수 있는 두 가지 상위 수준의 접근 방식이 있습니다. 첫 번째 방법은 장애가 발생한 후 이에 대응하는 것입니다. AWS Auto Scaling로 새 용량을 추가하는 방법이 있겠지요. 두 번째 방법은 이러한 장애가 발생하기 전에 이에 대비하는 것입니다. 예를 들어 워크로드의 인프라를 오버프로비저닝하여 추가 리소스 없이도 올바르게 운영될 수 있도록 하는 것입니다.

정적으로 안정적인 시스템은 후자의 접근 방식을 사용합니다. 고장 발생 시 사용할 수 있도록 예비 용량을 미리 프로비저닝합니다. 이 방법을 사용하면 고장 복구를 위한 새 용량을 프로비저닝하기 위해 워크로드의 복구 경로에 컨트롤 플레인에 종속성을 생성하지 않아도 됩니다. 또한 다양한 리소스에 새 용량을 프로비저닝하려면 시간이 걸립니다. 새 용량을 기다리는 동안 기존 수요로 인해 워크로드에 과부하가 걸리고 추가 성능 저하가 발생하여 가용성이 완전히 손실될 수 있습니다. 하지만 사전 프로비저닝된 용량을 가용성 목표에 맞게 활용할 경우 비용에 미치는 영향도 고려해야 합니다.

정적 안정성은 고가용성 워크로드에 대한 다음 두 가지 규칙을 제공합니다.

> **규칙 7**
>
> 특히 복구 중에는 데이터 영역의 컨트롤 플레인에 대한 종속성을 고려하지 마세요.

> **규칙 8**
>
> 가능한 경우 종속성이 손상되더라도 워크로드가 올바르게 작동할 수 있도록 종속성을 느슨하게 결합하세요.

가용성 측정

앞서 살펴본 것처럼 분산 시스템을 위한 미래 지향적인 가용성 모델을 만드는 것은 어려운 일이며 원하는 통찰력을 제공하지 못할 수도 있습니다. 워크로드의 가용성을 측정하는 일관된 방법을 개발하는 것이 유용성을 더할 수 있습니다.

가용성을 가동 시간과 가동 중지로 정의하면 워크로드가 증가하든 그렇지 않든 고장을 이분법적으로 나타냅니다.

하지만 이런 경우는 거의 없습니다. 고장은 상당한 영향을 미치며 워크로드의 일부 하위 집합에서 발생하는 경우가 많으며, 이는 사용자 또는 요청의 비율, 위치의 백분율 또는 지연 시간의 백분위수에 영향을 미칩니다. 이는 모두 부분적 고장 모드입니다.

MTTR과 MTBF는 결과적으로 시스템의 가용성을 좌우하는 요소와 이를 개선하는 방법을 이해하는 데 유용하지만 가용성을 실증적으로 측정할 수 있는 척도로는 유용하지 않습니다. 또한 워크로드는 여러 구성 요소로 구성되어 있습니다. 예를 들어 결제 처리 시스템과 같은 워크로드는 여러 애플리케이션 프로그래밍 인터페이스(API)와 하위 시스템으로 구성됩니다. 따라서 “전체 워크로드의 가용성은 어떻게 됩니까?”와 같은 질문은 사실 복잡하고 미묘한 질문입니다.

이 항목에서는 가용성을 경험적으로 측정할 수 있는 세 가지 방법, 즉 서버 측 요청 성공률, 클라이언트 측 요청 성공률, 연간 가동 중지를 살펴보겠습니다.

서버 측 및 클라이언트 측 요청 성공률

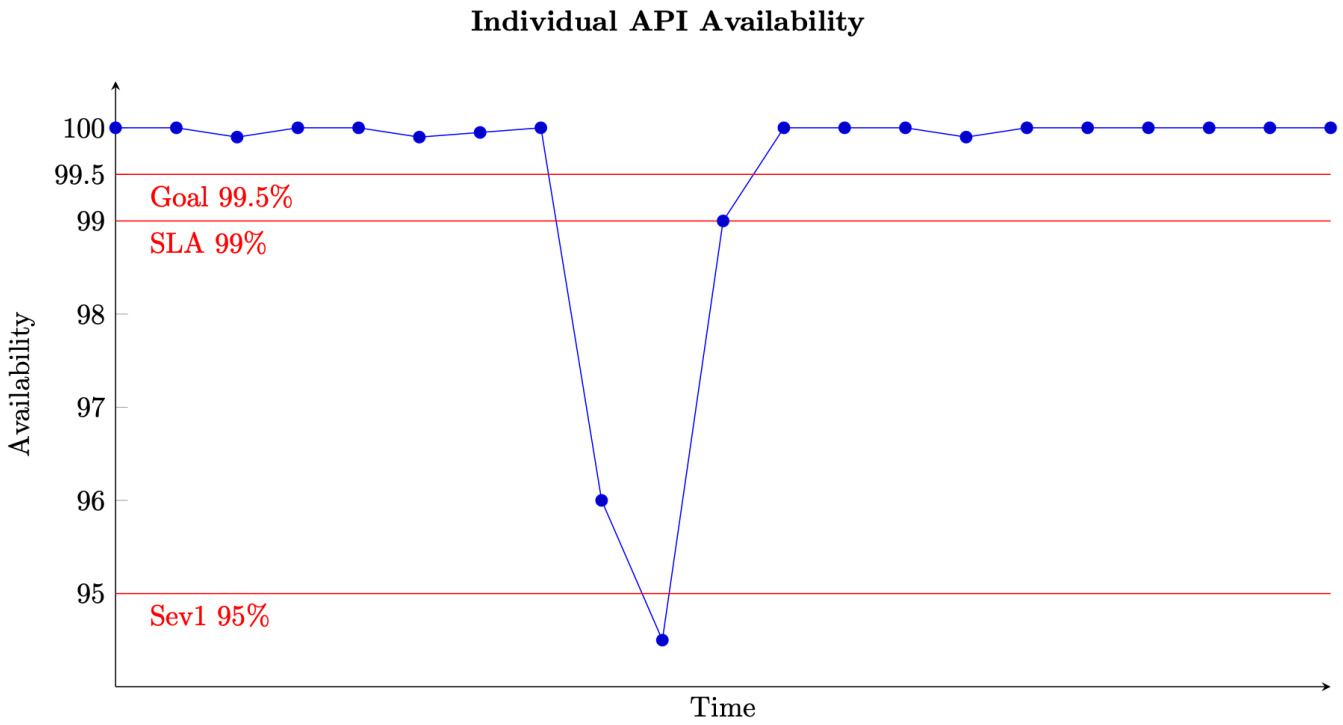
처음 두 가지 방법은 매우 유사하지만 측정이 취해지는 관점에서만 다릅니다. 서버 측 지표는 서비스의 계측에서 수집할 수 있습니다. 하지만 아직 완전하지는 않습니다. 클라이언트가 서비스에 도달할 수 없는 경우 해당 지표를 수집할 수 없습니다. 클라이언트 경험을 이해하려면 실패한 요청에 대한 클라이언트의 원격 분석에 의존하는 대신 정기적으로 서비스를 조사하고 지표를 기록하는 소프트웨어인 canary를 사용하여 고객 트래픽을 시뮬레이션하는 것이 클라이언트 측 지표를 수집하는 더 쉬운 방법입니다.

이 두 방법은 서비스가 수신한 총 유효 작업 단위와 성공적으로 처리한 작업 단위의 비율로 가용성을 계산합니다. 이렇게 하면 404 오류가 발생하는 HTTP 요청과 같은 잘못된 작업 단위는 무시됩니다.

$$A = \frac{Successfully\ Processed\ Units\ of\ Work}{Total\ Valid\ Units\ of\ Work\ Received}$$

방정식 8

요청 기반 서비스의 경우 작업 단위는 HTTP 요청과 마찬가지로 요청입니다. 이벤트 기반 또는 작업 기반 서비스의 경우 작업 단위는 대기열에서 메시지를 처리하는 것과 같은 이벤트 또는 작업입니다. 이 가용성 측정은 1분 또는 5분과 같은 짧은 시간 간격에서 의미가 있습니다. 또한 요청 기반 서비스의 API 수준과 같이 세분화된 관점에서도 가장 적합합니다. 다음 그림은 이러한 방식으로 계산했을 때 시간 경과에 따른 가용성을 보여줍니다. 그래프의 각 데이터 포인트는 방정식 (8)을 사용하여 5분 동안 계산됩니다(1분 또는 10분 간격과 같은 다른 시간 차원을 선택할 수 있음). 예를 들어, 데이터 포인트 10은 94.5%의 가용성을 보여줍니다. 즉, 서비스가 1,000개의 요청을 받은 경우 t+45에서 t+50분 동안 이 중 945개만 성공적으로 처리된 것입니다.



단일 API의 시간 경과에 따른 가용성 측정 예제

그래프에는 API의 가용성 목표, 99.5% 가용성, 고객에게 제공하는 서비스 수준에 관한 계약(SLA), 99% 가용성, 심각도가 높은 경보의 임계값인 95%도 표시됩니다. 이러한 다양한 임계값을 고려하지 않으면 가용성 그래프로는 서비스 운영 방식에 대한 중요한 통찰력을 제공하지 못할 수 있습니다.

목표는 컨트롤 플레인 같은 대규모 하위 시스템이나 전체 서비스의 가용성을 추적하고 설명하는 것입니다. 이를 위한 한 가지 방법은 각 하위 시스템에 대해 각 5분 데이터 포인트의 평균을 구하는 것입니다. 그래프는 이전 그래프와 비슷해 보이지만 더 큰 입력 집합을 나타냅니다. 또한 서비스를 구성하는 모든 하위 시스템에 동일한 가중치를 부여합니다. 또 다른 방법은 서비스의 모든 API에서 수신되고 성공적으로 처리된 모든 요청을 합산하여 5분 간격으로 가용성을 계산하는 것입니다.

하지만 이 후자의 방법을 사용하면 처리량이 낮고 가용성이 좋지 않은 개별 API를 숨길 수 있습니다. 간단한 예로 두 개의 API가 있는 서비스를 생각해 봅시다.

첫 번째 API는 5분 내에 1,000,000개의 요청을 수신하고 그 중 999,000개를 성공적으로 처리하여 99.9%의 가용성을 제공합니다. 두 번째 API는 동일한 5분 내에 100개의 요청을 수신하고 그 중 50개만 성공적으로 처리하여 50%의 가용성을 제공합니다.

각 API의 요청을 합산하면 총 1,000,100개의 유효한 요청이 있고 그 중 999,050개가 성공적으로 처리되어 전체 서비스에 대해 99.895%의 가용성이 제공됩니다. 그러나 전자의 방법인 두 API의 가용성을 평균하면 74.95%의 가용성이 나오는데, 이는 실제 경험을 더 잘 설명해 줄 수 있습니다.

두 접근 방식 모두 잘못된 것은 아니지만 가용성 지표가 무엇을 알려주는지 이해하는 것이 중요하다는 것을 알 수 있습니다. 워크로드가 각 하위 시스템에서 비슷한 요청을 받는 경우 모든 하위 시스템에 대해 요청을 합산하는 것을 선호할 수 있습니다. 이 접근 방식은 가용성과 고객 경험의 척도로서 "요청"과 성공에 초점을 맞춥니다. 또는 요청량 차이에도 불구하고 하위 시스템의 중요도를 동일하게 나타내기 위해 평균 하위 시스템 가용성을 선택할 수도 있습니다. 이 접근 방식은 하위 시스템과 고객 경험을 위한 프록시 역할을 하는 각 하위 시스템의 기능에 초점을 맞춥니다.

연간 가동 중지

세 번째 접근 방식은 연간 가동 중지를 계산하는 것입니다. 이러한 형태의 가용성 지표는 장기 목표 설정 및 검토에 더 적합합니다. 가동 중지가 워크로드에 미치는 영향을 정의해야 합니다. 그런 다음 해당 워크로드가 "중단" 상태가 아닌 시간(분)을 해당 기간의 총 시간(분)과 비교하여 가용성을 측정할 수 있습니다.

일부 워크로드는 1분 또는 5분 간격으로 단일 API 또는 워크로드 함수의 가용성이 95% 미만으로 떨어지는 것과 같이 가동 중지를 정의할 수 있습니다(이전 가용성 그래프에서 발생). 또한 가동 중지는 중요한 데이터 영역 작업의 일부에 적용되므로 가동 중지만 고려할 수도 있습니다. 예를 들어, SQS 가용성에 대한 Amazon 메시징(SQS, SNS) 서비스 수준에 관한 계약은 SQS 전송, 수신 및 삭제 API에 적용됩니다.

더 크고 복잡한 워크로드에는 시스템 전반의 가용성 지표를 정의해야 할 수 있습니다. 대규모 전자 상거래 사이트의 경우 시스템 전반의 지표는 고객 주문률과 같을 수 있습니다. 여기서 5분 동안 예상 수량에 비해 주문이 10% 이상 감소하면 가동 중지가 발생할 수 있습니다.

어느 방법을 사용하든 운영 중단 기간을 모두 합산하여 연간 가용성을 계산할 수 있습니다. 예를 들어, 한 해 동안 모든 데이터 영역 API의 가용성이 95% 미만으로 떨어지는 것으로 정의되는 5분 가동 중지가 27번 발생한 경우 전체 가동 중지는 135분(일부 5분 기간은 연속적이었을 수도 있고 다른 기간은 분리되었을 수도 있음)이었으며, 이는 연간 가용성이 99.97%에 달합니다.

가용성을 측정하는 이 추가 방법을 사용하면 클라이언트 측 및 서버 측 지표에서 누락된 데이터와 통찰력을 얻을 수 있습니다. 예를 들어, 워크로드가 손상되어 오류율이 크게 높아진 경우를 생각해 보세요. 이 워크로드의 고객은 해당 서비스에 대한 호출을 완전히 중단할 수 있습니다. 회로 차단기를 활성화했거나 재해 복구 계획에 따라 다른 리전에서 서비스를 사용했을 수 있습니다. 실패한 응답만 측정했다면 장애가 발생한 동안에도 실제로 워크로드의 가용성이 증가할 수 있습니다. 고장이 개선되거나 사라지기 때문이 아니라 고객이 사용을 중단하기 때문입니다.

지연 시간

마지막으로, 워크로드 내에서 작업 단위 처리의 지연 시간을 측정하는 것도 중요합니다. 가용성 정의의 일부는 설정된 SLA 내에서 작업을 수행하는 것입니다. 응답을 반환하는 데 클라이언트 제한 시간보다 오래 걸리는 경우 클라이언트는 요청이 실패하고 워크로드를 사용할 수 없다고 인식합니다. 하지만 서버 측에서는 요청이 성공적으로 처리된 것처럼 보일 수 있습니다.

지연 시간 측정은 가용성을 평가할 수 있는 또 다른 관점을 제공합니다. 이 측정에는 백분위수와 절삭 평균을 사용하는 것이 좋습니다. 일반적으로 50번째 백분위수(P50 및 TM50)와 99번째 백분위수(P99 및 TM99)로 측정됩니다. 클라이언트 경험을 나타내는 canary와 서버 측 지표를 사용하여 지연 시간을 측정해야 합니다. P99 또는 TM99.9와 같은 일부 백분위수 지연 시간의 평균이 목표 SLA를 초과할 때마다 해당 가동 중지 시간을 고려할 수 있으며, 이는 연간 가동 중지 시간 계산에 반영됩니다.

AWS에서 고가용성 분산 시스템 설계

이전 항목에서는 주로 워크로드의 이론적 가용성과 이를 통해 달성할 수 있는 것을 다루었습니다. 그것은 분산 시스템을 구축할 때 염두에 두어야 할 중요한 개념입니다. 그것을 통해 종속성 선택 프로세스와 중복성 구현 방법을 알 수 있습니다.

또한 MTTD, MTTR, MTBF와 가용성의 관계에 대해서도 살펴보았습니다. 이 항목은 이전 이론에 기반한 실용적인 지침을 소개합니다. 간단히 말해, 고가용성을 위한 엔지니어링 워크로드는 MTBF를 늘리고 MTTR과 MTTD를 줄이는 것을 목표로 합니다.

모든 고장을 제거하는 것이 이상적이기는 하지만 현실적이지는 않습니다. 종속성이 깊이 쌓여 있는 대규모 분산 시스템에서는 고장이 발생하기 마련입니다. “모든 것은 항상 실패합니다”(Werner Vogels, CTO, Amazon.com, CTO, Amazon Web Services 10년간 얻은 10가지 교훈 참조) “실패를 방지하는 법은 제정할 수 없으므로 빠른 탐지와 대응에 집중하세요” (Chris Pinkham, Amazon EC2 팀 창립 멤버, ARC335, 실패를 대비한 설계: AWS에서 복원력이 뛰어난 시스템 설계 기반)

이것이 의미하는 바는 고장 발생 여부를 제어할 수 없는 경우가 많다는 것입니다. 제어할 수 있는 것은 고장을 얼마나 빨리 감지하고 조치를 취하느냐입니다. 따라서 MTBF를 늘리는 것은 여전히 고가용성의 중요한 구성 요소이지만 고객이 통제할 수 있는 가장 중요한 변화는 MTTD와 MTTR을 줄이는 것입니다.

주제

- MTTD 감소
- MTTR 감소
- MTBF 증가

MTTD 감소

고장의 MTTD를 줄인다는 것은 고장을 최대한 빨리 발견하는 것을 의미합니다. MTTD 단축은 관찰 가능성, 즉 워크로드를 계측하여 상태를 파악한 방법을 기반으로 합니다. 고객은 문제 발생 시점을 사전에 식별할 수 있는 방법으로 워크로드의 중요 하위 시스템에서 고객 경험 지표를 모니터링해야 합니다(이러한 지표에 대한 자세한 내용은 부록 1 - MTTD 및 MTTR 중요 지표 참조). 고객은 Amazon CloudWatch Synthetics를 사용하여 API와 콘솔을 모니터링하는 canary를 생성하여 사용자 경험을 사전에 측정할 수 있습니다. Elastic Load Balancing(ELB) 상태 확인, Amazon Route 53 상태 확인 등과 같이 MTTD를 최소화하는 데 사용할 수 있는 다른 상태 점검 메커니즘도 많이 있습니다. (Amazon Builders' Library - 상태 확인 구현을 참조하세요.)

또한 모니터링을 통해 시스템 전체와 개별 하위 시스템 모두의 부분적 고장을 감지할 수 있어야 합니다. 가용성, 고장 및 지연 시간 지표는 고장 격리 경계의 차원을 CloudWatch 지표 차원으로 사용해야 합니다. 예를 들어 us-east-1 리전의 use1-az1 AZ에 있는 셀 기반 아키텍처의 일부인 단일 EC2 인스턴스가 컨트롤 플레인 하위 시스템의 일부인 워크로드 업데이트 API의 일부라고 가정해 보겠습니다. 서버는 지표를 푸시할 때 인스턴스 ID, AZ, 리전, API 이름 및 하위 시스템 이름을 차원으로 사용할 수 있습니다. 이를 통해 관찰성을 확보하고 각 차원에 경보를 설정하여 고장을 감지할 수 있습니다.

MTTR 감소

고장이 발견된 후 남은 MTTR 시간은 실제 수리 또는 영향 완화에 사용됩니다. 고장을 복구하거나 완화하려면 무엇이 잘못되었는지 알아야 합니다. 이 단계에서 통찰력을 제공하는 두 가지 주요 지표 그룹이 있습니다. 바로 1/영향 평가 지표와 2/운영 상태 지표입니다. 첫 번째 그룹은 영향을 받은 고객, 리소스 또는 워크로드의 수 또는 비율을 측정하여 고장 발생 시 영향의 범위를 알려줍니다. 두 번째 그룹은 영향이 생긴 이유를 식별하는 데 도움이 됩니다. 이유가 밝혀지면 운영자와 자동화가 고장에 대응하고 문제를 해결할 수 있습니다. 이러한 지표에 대한 자세한 내용은 부록 1 - MTTD 및 MTTR 중요 지표를 참조하세요.

> 규칙 9
>
> 관찰성과 계측은 MTTD 및 MTTR을 줄이는 데 매우 중요합니다.

고장 우회 경로

영향을 완화하는 가장 빠른 방법은 고장을 우회하는 빠른 실패 하위 시스템을 이용하는 것입니다. 이 접근 방식은 중복성을 사용하여 고장이 발생한 하위 시스템의 작업을 스페어 시스템으로 신속하게 전환하여 MTTR을 줄입니다. 중복성은 소프트웨어 프로세스에서 EC2 인스턴스, 여러 AZ, 여러 리전에 이르기까지 다양합니다.

스페어 하위 시스템을 사용하면 MTTR을 거의 0으로 줄일 수 있습니다. 복구 시간은 작업을 대기 스페어 부품으로 다시 라우팅하는 데 걸리는 시간에 불과합니다. 이는 대개 지연 시간을 최소화하면서 발생하며 정의된 SLA 내에서 작업을 완료하여 시스템 가용성을 유지할 수 있습니다. 이로 인해 MTTR을 사용할 수 없는 기간이 길어지는 것이 아니라 미미하거나 심지어 감지할 수 없을 정도로 지연되는 경우가 발생합니다.

예를 들어, 서비스가 Application Load Balancer(ALB) 기반의 EC2 인스턴스를 사용하는 경우 최소 5초 간격으로 상태 확인을 구성하고, 실패한 상태 확인을 두 번만으로 대상이 비정상으로 표시되게 할 수

있습니다. 즉, 10초 이내에 고장을 감지하고 비정상 호스트로의 트래픽 전송을 중단할 수 있습니다. 이 경우 MTTR은 고장이 감지되는 즉시 완화되므로 MTTD와 사실상 동일합니다.

이것이 바로 고가용성 또는 지속적인 가용성 워크로드가 달성하고자 하는 것입니다. 고장이 발생한 하위 시스템을 신속하게 탐지하여 고장이 발생한 것으로 표시하고, 트래픽 전송을 중단하고, 대신 중복된 하위 시스템으로 트래픽을 전송함으로써 워크로드의 고장을 신속하게 우회하고자 합니다.

이러한 종류의 빠른 실패 메커니즘을 사용하면 워크로드가 일시적인 오류에 매우 민감하게 반응한다는 점에 유의하세요. 제공된 예제에서 로드 밸런서 상태 확인이 종속성이나 워크플로를 테스트하지 않고(깊은 상태 확인이라고도 함) 인스턴스에 대해서만 얕은 상태 또는 활성 및 로컬 상태 확인을 수행하는지 확인하세요. 이렇게 하면 워크로드에 영향을 미치는 일시적 오류가 발생하는 동안 인스턴스를 불필요하게 교체하는 것을 방지할 수 있습니다.

고장 우회 라우팅이 성공하려면 하위 시스템의 고장 감지 기능과 관찰성이 매우 중요합니다. 영향을 받는 리소스를 비정상 또는 고장으로 표시하고 서비스를 중단하여 다른 곳으로 라우팅할 수 있도록 하려면 영향의 범위를 알아야 합니다. 예를 들어 단일 AZ에 부분적인 서비스 장애가 발생하는 경우, 계측기는 복구될 때까지 해당 AZ의 모든 리소스를 라우팅하기 위해 AZ로 지역화된 문제가 있음을 식별할 수 있어야 합니다.

고장을 우회하려면 환경에 따라 추가 도구가 필요할 수도 있습니다. ALB를 기반으로 하는 EC2 인스턴스를 사용하는 이전 예제를 사용하여 한 AZ의 인스턴스가 로컬 상태 확인을 통과했지만 격리된 AZ 장애로 인해 다른 AZ에 있는 데이터베이스에 연결하지 못한다고 가정해 보겠습니다. 이 경우 로드 밸런서 상태 확인으로 해당 인스턴스의 서비스가 중단되지는 않습니다. 로드 밸런서에서 AZ를 제거하거나 인스턴스가 상태 확인에 실패하도록 하려면 다른 자동 메커니즘이 필요합니다. 이는 영향 범위가 AZ인지 확인하는 데 따라 달라집니다. 로드 밸런서를 사용하지 않는 워크로드의 경우 특정 AZ의 리소스가 작업 단위를 수락하지 못하도록 하거나 AZ에서 용량을 완전히 제거하는 유사한 방법이 필요합니다.

중복된 하위 시스템으로의 작업 이동을 자동화할 수 없는 경우도 있습니다. 예를 들어 기술 자체에서 리더를 선택할 수 없는 기본 데이터베이스에서 보조 데이터베이스로의 장애 조치도 마찬가지입니다. 이는 AWS 다중 리전 아키텍처의 일반적인 시나리오입니다. 이러한 유형의 장애 조치를 수행하려면 어느 정도의 가동 중지가 필요하고, 즉시 되돌릴 수 없으며, 일정 기간 동안 워크로드를 중복되지 않은 상태로 둘 수 있기 때문에 의사 결정 프로세스에 인력을 두는 것이 중요합니다.

덜 엄격한 정합성 모델을 적용할 수 있는 워크로드는 다중 리전 장애 조치 자동화를 사용하여 고장을 우회함으로써 MTTR을 단축할 수 있습니다. Amazon S3 교차 리전 복제 또는 Amazon DynamoDB 글로벌 테이블과 같은 특성은 최종적으로 일관된 복제를 통해 다중 리전 기능을 제공합니다. 또한 CAP 정리를 고려할 때는 완화된 정합성 모델을 사용하는 것이 좋습니다. 상태 저장 하위 시스템에 대한 연결에 영향을 미치는 네트워크 고장 발생 시 워크로드가 일관성보다 가용성을 선택하더라도 오류 없는 응답을 제공할 수 있는데, 이는 고장을 우회하는 또 다른 방법입니다.

두 가지 전략을 사용하여 고장 우회 라우팅을 구현할 수 있습니다. 첫 번째 전략은 고장이 발생한 하위 시스템의 전체 부하를 처리할 수 있는 충분한 리소스를 사전에 프로비저닝하여 정적 안정성을 구현하는 것입니다. 이는 단일 EC2 인스턴스일 수도 있고 전체 AZ 용량일 수도 있습니다. 고장 발생 시 새 리소스를 프로비저닝하려고 하면 MTTR이 증가하고 복구 경로의 컨트롤 플레인에 종속성이 추가됩니다. 하지만 추가 비용이 발생합니다.

두 번째 전략은 장애가 발생한 하위 시스템의 일부 트래픽을 다른 하위 시스템으로 라우팅하고 남은 용량으로는 처리할 수 없는 초과 트래픽이 주는 부하를 제거하는 것입니다. 이 성능 저하 기간 동안에는 새 리소스를 확장하여 고장이 발생한 용량을 대체할 수 있습니다. 이 접근 방식은 MTTR이 더 길고 컨트롤 플레인에 대한 종속성을 생성하지만 대기 스페어 용량의 경우 비용이 더 적게 듭니다.

정상 작동이 확인된 상태로 돌아가기

복구 중에 이를 완화하는 또 다른 일반적인 방법은 워크로드를 이전의 알려진 정상 상태로 되돌리는 것입니다. 최근 변경으로 인해 고장이 발생했을 수 있는 경우 해당 변경 사항을 롤백하는 것이 이전 상태로 돌아갈 수 있는 한 가지 방법입니다.

다른 경우에는 일시적인 상황으로 인해 고장이 발생했을 수 있으며, 이 경우 워크로드를 다시 시작하여 영향을 완화할 수 있습니다. 이 두 경우를 모두 살펴보겠습니다.

배포 중에 MTTD 및 MTTR을 최소화하려면 관찰성과 자동화가 관건입니다. 배포 프로세스에서는 오류율 증가, 지연 시간 증가 또는 이상 현상이 발생하지 않는지 지속적으로 워크로드를 주시해야 합니다. 이러한 문제가 인식되면 배포 프로세스를 중단해야 합니다.

인플레이스 배포, 블루/그린 배포, 롤링 배포와 같은 다양한 배포 전략이 있습니다. 이들 각각은 정상 작동이 확인된 상태로 되돌리기 위해 서로 다른 메커니즘을 사용할 수 있습니다. 자동으로 이전 상태로 롤백하거나, 트래픽을 다시 블루 환경으로 전환하거나, 수동 개입이 필요할 수 있습니다.

CloudFormation은 스택 생성 및 업데이트 작업의 일부로 자동 롤백하는 기능을 제공하며, AWS CodeDeploy 항목도 마찬가지입니다. CodeDeploy는 블루/그린 및 롤링 배포도 지원합니다.

이러한 기능을 활용하고 MTTR을 최소화하려면 이러한 서비스를 통해 모든 인프라 및 코드 배포를 자동화하는 것을 고려해 보세요. 이러한 서비스를 사용할 수 없는 경우에서는 고장 난 배포를 롤백하기 위해 AWS Step Functions 항목으로 saga 패턴을 구현하는 것을 고려해 보세요.

재시작을 고려할 때는 여러 가지 접근 방식이 있습니다. 여기에는 가장 긴 작업인 서버 재부팅부터 가장 짧은 작업인 스레드 재시작까지 다양합니다. 다음은 몇 가지 재시작 접근 방식과 완료에 소요되는 대략적인 시간을 요약한 표입니다(몇 분의 큰 차이를 나타내며 정확하지는 않음).

장애 복구 메커니즘	예상 MTTR
새 가상 서버 시작 및 구성	15분
소프트웨어 재배포	10분
서버 재부팅	5분
컨테이너 재시작 또는 실행	2초
새 서버리스 함수 간접 호출	100ms
프로세스 다시 시작	10ms
스레드 다시 시작	10 마이크로초

표를 검토해 보면 컨테이너 및 서버리스 함수(예: AWS Lambda)를 사용할 때 MTTR의 몇 가지 분명한 이점이 있습니다. MTTR은 가상 시스템을 재시작하거나 새 가상 시스템을 시작하는 것보다 몇 배 더 빠릅니다. 하지만 소프트웨어 모듈화를 통해 장애를 격리하는 것도 유용합니다. 단일 프로세스나 스레드에 고장을 포함할 수 있는 경우 해당 고장을 복구하는 것이 컨테이너나 서버를 다시 시작하는 것보다 훨씬 빠릅니다.

일반적인 복구 접근 방식으로 1/재시작, 2/재부팅, 3/이미지 재생성/재배포, 4/바꾸기를 뒤에서부터 앞으로 시도할 수 있습니다. 하지만 재부팅 단계로 넘어가면 일반적으로 고장을 우회하는 것이 더 빠릅니다(보통 최대 3~4분 소요). 따라서 재시작 시도 후 영향을 가장 빠르게 완화하려면 고장을 우회한 다음 백그라운드에서 복구 프로세스를 계속하여 워크로드에 용량을 반환하세요.

> 규칙 10
>
> 문제 해결이 아닌 영향 완화에 집중하세요. 정상 작동 상태로 돌아가는 가장 빠른 길을 택하세요.

고장 진단

진단 기간은 감지 후 수리 프로세스의 일부입니다. 이 기간은 작업자가 무엇이 잘못되었는지 판단하는 기간입니다. 이 프로세스에는 로그 쿼리, 작업 상태 지표 검토 또는 문제 해결을 위한 호스트 로그인이 포함될 수 있습니다. 이러한 모든 작업에는 시간이 필요하므로 이러한 조치를 신속하게 처리하는 도구와 런북을 만들면 MTTR을 줄이는 데도 도움이 될 수 있습니다.

런북 및 자동화

마찬가지로, 무엇이 잘못되었고 어떤 조치를 취해야 워크로드가 복구될 것인지 판단한 후 작업자는 일반적으로 이를 위해 몇 가지 단계를 수행해야 합니다. 예를 들어 고장이 발생한 후 워크로드를 복구하는 가장 빠른 방법은 워크로드를 다시 시작하는 것일 수 있으며, 이 경우 순서가 정해진 여러 단계를 따라야 할 수 있습니다. 이러한 단계를 자동화하거나 운영자에게 구체적인 지침을 제공하는 런북을 활용하면 프로세스를 가속화하고 의도하지 않은 조치로 인한 위험을 줄일 수 있습니다.

MTBF 증가

가용성을 개선하기 위한 마지막 요소는 MTBF를 늘리는 것입니다. 이는 소프트웨어와 이를 실행하는 데 사용된 AWS 서비스 모두에 적용될 수 있습니다.

분산 시스템 MTBF 증가

MTBF를 높이는 한 가지 방법은 소프트웨어의 결함을 줄이는 것입니다. 이를 달성하는 데는 몇 가지 방법이 있습니다. 고객은 Amazon CodeGuru Reviewer 같은 도구를 사용하여 일반적인 오류를 찾아 수정할 수 있습니다. 또한 소프트웨어를 프로덕션에 배포하기 전에 포괄적인 피어 코드 검토, 유닛 테스트, 통합 테스트, 회귀 테스트, 소프트웨어 부하 테스트 등을 수행해야 합니다. 테스트에서 코드 적용 범위를 늘리면 흔하지 않은 코드 실행 경로도 테스트할 수 있습니다.

작은 변경 내용을 배포하는 것도 변경의 복잡성을 줄여 예상치 못한 결과를 방지하는 데 도움이 될 수 있습니다. 각 활동을 통해 문제가 발생하기 전에 결함을 식별하고 수정할 수 있습니다.

고장을 예방하는 또 다른 방법은 정기적인 테스트입니다. 카오스 엔지니어링 프로그램을 구현하면 워크로드 고장 방식을 테스트하고, 복구 절차를 검증하고, 프로덕션 환경에서 고장이 발생하기 전에 고장 모드를 찾아 수정하는 데 도움이 될 수 있습니다. 고객은 카오스 엔지니어링 실험 도구 세트의 일부로 AWS Fault Injection Simulator를 사용할 수 있습니다.

내결함성은 분산 시스템에서 고장을 방지하는 또 다른 방법입니다. 빠른 실패 모듈, 지수 백오프 및 지터가 발생하는 재시도, 트랜잭션, 멱등성 등은 모두 워크로드의 내결함성을 높이는 데 도움이 되는 기법입니다.

트랜잭션은 ACID 속성을 준수하는 작업 그룹입니다. 그 속성이란 다음과 같습니다.

- 원자성: 모든 행동이 일어나거나 전혀 일어나지 않을 것입니다.
- 일관성: 각 트랜잭션은 워크로드를 유효한 상태로 놔둡니다.
- 격리: 동시에 수행된 트랜잭션은 워크로드를 순차적으로 수행된 것과 동일한 상태로 놔둡니다.
- 내구성: 트랜잭션이 커밋되면 워크로드에 고장이 발생한 경우에도 모든 영향이 보존됩니다.

지수 백오프 및 지터가 있는 재시도를 통해 하이젠버그, 오버로드 또는 기타 조건으로 인한 일시적 고장을 극복할 수 있습니다. 트랜잭션이 멱등인 경우 부작용 없이 여러 번 재시도할 수 있습니다.

하이젠버그가 내결함성 하드웨어 구성에 미치는 영향을 고려한다면, 하이젠버그가 기본 하위 시스템과 중복 하위 시스템 모두에 나타날 확률은 극히 적기 때문에 크게 걱정하지 않아도 될 것입니다. (Jim Gray, "컴퓨터가 멈추는 이유와 이에 대해 취할 수 있는 조치", 1985년 6월, 탠덤 기술 보고서 85.7. 참조) 분산 시스템에서 우리는 소프트웨어를 사용하여 동일한 결과를 달성하고자 합니다.

하이젠버그가 간접 호출되면 소프트웨어가 잘못된 작업을 빠르게 감지하고 실패하여 다시 시도하는 게 중요합니다. 이는 방어 프로그래밍과 입력, 중간 결과 및 출력의 검증을 통해 달성됩니다. 또한 프로세스는 격리되며 다른 프로세스와 상태를 공유하지 않습니다.

이 모듈식 접근 방식을 통해 고장 발생 시 영향의 범위를 제한할 수 있습니다. 프로세스는 독립적으로 고장 납니다. 프로세스가 실패하면 소프트웨어가 "프로세스 쌍"을 사용하여 작업을 재시도해야 합니다. 즉, 새 프로세스가 실패한 프로세스를 맡을 수 있습니다. 워크로드의 신뢰성과 무결성을 유지하려면 각 작업을 ACID 트랜잭션으로 처리해야 합니다.

이렇게 하면 트랜잭션을 중단하고 변경 내용을 롤백하여 워크로드 상태를 손상시키지 않고 프로세스가 실패할 수 있습니다. 이렇게 하면 복구 프로세스가 정상 작동이 확인된 상태에서 트랜잭션을 재시도하고 정상적으로 재시작할 수 있습니다. 이렇게 하면 소프트웨어가 하이젠버그에 대한 내결함성을 유지할 수 있습니다.

하지만 보어버그를 방지하는 소프트웨어를 만드는 것을 목표로 삼아서는 안 됩니다. 어떤 수준의 중복성으로도 올바른 결과를 얻을 수 없으므로 워크로드가 프로덕션에 들어가기 전에 이러한 결함을 찾아 제거해야 합니다. (Jim Gray, "컴퓨터가 멈추는 이유와 이에 대해 취할 수 있는 조치", 1985년 6월, 탠덤 기술 보고서 85.7. 참조)

MTBF를 늘리는 마지막 방법은 고장으로 인한 영향 범위를 줄이는 것입니다. 모듈화를 통한 장애 격리를 사용하여 장애 컨테이너를 생성하는 것은 앞서 내결함성 및 장애 격리에서 설명한 바와 같이 이를 위한 기본 방법입니다. 고장률을 줄이면 가용성이 향상됩니다. AWS는 서비스를 컨트롤 플레인 및 데이터 영역으로 분할, 가용 영역 독립(AZI), 리전 격리, 셀 기반 아키텍처, 셔플 샤딩과 같은 기술을 사용하여 고장을 격리합니다. 이는 AWS 고객이 사용할 수 있는 패턴이기도 합니다.

예를 들어, 전체 고객 중 최대 5%에 해당하는 서비스를 제공하는 워크로드가 자신의 인프라 안의 여러 장애 컨테이너에 고객을 배치한 경우를 살펴보겠습니다. 이러한 장애 컨테이너 중 하나에서 요청의 10%에 대해 클라이언트 제한 시간을 초과하여 지연 시간이 증가하는 이벤트가 발생합니다. 이 이벤트 기간 동안 95%의 고객이 서비스를 100% 이용할 수 있었습니다. 나머지 5%의 경우 서비스를 90% 이용할 수 있는 것으로 나타났습니다. 따라서 100%의 고객에게 요청의 10%가 실패하는 대신 1 − (5% of customers×10% of their requests) = 99.5%의 가용성이 확보됩니다(결과적으로 90%의 가용성이 보장됨).

규칙 11

고장 격리는 전체 고장률을 줄여 영향 범위를 줄이고 워크로드의 MTBF를 증가시킵니다.

종속성 MTBF 증가

AWS 종속성 MTBF를 높이는 첫 번째 방법은 장애 격리를 사용하는 것입니다. 많은 AWS 서비스가 AZ에서 격리 수준을 제공합니다. 즉, 한 AZ에서 고장이 발생해도 다른 AZ의 서비스에는 영향을 주지 않습니다.

여러 AZ에서 중복 EC2 인스턴스를 사용하면 하위 시스템 가용성이 향상됩니다. AZI는 단일 리전 내에서 스페어링 기능을 제공하므로 AZI 서비스의 가용성을 높일 수 있습니다.

하지만 모든 AWS 서비스가 AZ 수준에서 운영되는 것은 아닙니다. 리전별 격리를 제공하는 곳도 많습니다. 이 경우, 리전 서비스에 맞게 설계된 가용성이 워크로드에 필요한 전체 가용성을 지원하지 않는 경우 다중 리전 접근 방식을 고려할 수 있습니다. 각 리전은 스페어링에 해당하는 격리된 서비스 인스턴스화를 제공합니다.

다중 리전 서비스를 더욱 쉽게 구축하는 데 도움이 되는 다양한 서비스가 있습니다. 예:

- Amazon Aurora Global Database
- Amazon DynamoDB 글로벌 테이블
- Amazon ElastiCache for Redis - 글로벌 데이터 스토어
- AWS Global Accelerator
- Amazon S3 교차 리전 복제
- Amazon Route 53 애플리케이션 복구 컨트롤러

이 문서는 다중 리전 워크로드를 구축하는 전략을 자세히 다루지는 않지만, 사용자는 원하는 가용성 목표를 달성하는 데 필요한 추가 비용, 복잡성 및 운영 관행과 함께 다중 리전 아키텍처의 가용성 이점을 비교해 봐야 합니다.

MTBF의 종속성을 높이는 다른 방법은 워크로드를 정적으로 안정적으로 설계하는 것입니다. 예를 들어, 제품 정보를 제공하는 워크로드가 있다고 가정합니다. 고객이 제품을 요청하면 서비스는 외부 메타데이터 서비스에 요청하여 제품 세부 정보를 검색합니다. 그러면 워크로드가 해당 정보를 모두 고객에게 반환합니다.

하지만 메타데이터 서비스를 사용할 수 없는 경우 고객의 요청은 실패합니다. 대신 요청에 응답하는 데 사용할 메타데이터를 서비스에 로컬로 비동기적으로 가져오거나 푸시할 수 있습니다. 이렇게 하면 중요 경로에서 메타데이터 서비스에 대한 동기 직접 호출이 제거됩니다.

또한 메타데이터 서비스가 없는 경우에도 서비스를 계속 사용할 수 있으므로 가용성 계산 시 종속 항목으로 해당 서비스를 제거할 수 있습니다. 이 예는 메타데이터가 자주 변경되지 않으며 요청이 실패하는 것보다 오래된 메타데이터를 제공하는 것이 낫다는 가정을 기반으로 합니다. 또 다른 유사한 예로는 TTL 만료 이후에도 데이터를 캐시에 보관하여 새로 고쳐진 답변을 쉽게 찾을 수 없을 때 응답에 사용할 수 있는 DNS용 serve-stale을 들 수 있습니다.

종속성 MTBF를 높이는 마지막 방법은 고장으로 인한 영향 범위를 줄이는 것입니다. 앞서 설명했듯이, 고장은 났는가 안 났는가로 나뉘지 않고, 얼마나 심하게 또는 경미하게 났는가로 나뉩니다. 이는 모듈화의 영향입니다. 고장은 해당 컨테이너에서 서비스를 받는 요청이나 사용자에게만 국한됩니다.

따라서 이벤트 중에 발생하는 고장이 줄어들어 영향 범위가 제한되므로 궁극적으로 전체 워크로드의 가용성이 향상됩니다.

일반적인 영향 원인 감소

1985년, Jim Gray는 탠덤 컴퓨터에서 연구를 진행하면서 고장이 주로 소프트웨어와 운영이라는 두 가지 요인에 의해 발생한다는 사실을 발견했습니다. (Jim Gray, "컴퓨터가 멈추는 이유와 이에 대해 취할 수 있는 조치", 1985년 6월, 탠덤 기술 보고서 85.7. 참조) 36년이 지난 지금도 이 사실은 변함이 없습니다. 기술의 발전에도 불구하고 이러한 문제를 쉽게 해결할 수 있는 방법은 없으며 고장의 주요 원인은 변하지 않았습니다. 이 항목의 시작 부분에서 소프트웨어 고장 해결에 대해 설명했으므로 여기서는 운영과 고장 빈도 감소에 중점을 둘 것입니다.

안정성과 특성 비교

the section called "분산 시스템 가용성" 항목의 소프트웨어 및 하드웨어 고장률 그래프를 다시 참조하면 각 소프트웨어 릴리스마다 결함이 추가되고 있음을 알 수 있습니다. 즉, 워크로드가 변경되면 고장 위험이 높아집니다. 이러한 변경은 일반적으로 새 특성과 비슷하며 이에 따른 결과를 제공합니다. 워크로드의 가용성이 높을수록 새 특성보다 안정성이 더 좋습니다. 따라서 가용성을 개선하는 가장 간단한 방법 중 하나는 배포 빈도를 줄이거나 특성 수를 줄이는 것입니다. 배포 빈도가 높은 워크로드는 그렇지 않은 워크로드보다 본질적으로 가용성이 낮습니다. 그러나 특성을 추가하지 못한 워크로드는 고객 수요를 따라가지 못하고 시간이 지남에 따라 유용성이 떨어질 수 있습니다.

그렇다면 계속해서 혁신하고 특성을 안전하게 출시하려면 어떻게 해야 할까요? 답은 표준화입니다. 올바른 배포 방법은 무엇입니까? 배포를 어떻게 주문합니까? 테스트 표준은 무엇입니까? 스테이지 사이에 얼마나 시간을 두겠습니까? 유닛 테스트에서 소프트웨어 코드를 충분히 다루고 있나요? 표준화를

통해 이러한 질문에 대한 답을 찾고 부하 테스트를 하지 않거나, 배포 단계를 건너뛰거나, 너무 많은 호스트에 너무 빨리 배포하는 등의 문제로 인해 발생하는 문제를 예방할 수 있습니다.

표준화를 구현하는 방법은 자동화를 이용하는 것입니다. 이를 통해 사람이 실수할 가능성이 줄어들고 컴퓨터가 잘하는 일을 할 수 있게 됩니다. 즉, 매번 같은 작업을 반복해서 수행하는 거죠. 표준화와 자동화를 함께 유지하는 방법은 목표를 설정하는 것입니다. 수동 변경 없음, 조건부 인증 시스템을 통해서만 호스트 액세스, 모든 API에 대한 부하 테스트 작성 등과 같은 목표가 있습니다. 운영의 우수성은 문화적 규범이며 상당한 변화가 필요할 수 있습니다. 목표 대비 성과를 설정하고 추적하면 워크로드 가용성에 광범위한 영향을 미칠 문화적 변화를 주도하는 데 도움이 됩니다. AWS Well-Architected 운영 우수성 요소는 운영 우수성을 위한 포괄적인 모범 사례를 제공합니다.

운영자 안전

고장을 초래하는 운영 문제의 또 다른 주요 원인은 바로 사람입니다. 사람은 실수를 합니다. 잘못된 자격 증명을 사용하거나, 잘못된 명령을 입력하거나, Enter 키를 너무 빨리 누르거나, 중요한 단계를 놓칠 수 있습니다. 수동 조치를 지속적으로 취하면 오류가 발생하여 고장이 일어납니다.

운영자 오류의 주요 원인 중 하나는 혼란스럽거나 직관적이지 않거나 일관되지 않은 사용자 인터페이스입니다. Jim Gray는 또한 1985년 연구에서 “운영자에게 정보를 요청하거나 일부 기능을 수행하도록 요청하는 인터페이스는 단순하고 일관적이며 운영자 내결함성이 있어야 한다”고 언급했습니다. (Jim Gray, “컴퓨터가 멈추는 이유와 이에 대해 취할 수 있는 조치”, 1985년 6월, 탠덤 기술 보고서 85.7. 참조) 이 통찰은 오늘날에도 여전히 유효합니다. 지난 30년 동안 업계 전반에 걸쳐 혼란스럽거나 복잡한 사용자 인터페이스, 확인이나 지침의 부재, 심지어 비우호적인 인간의 언어 때문에 운영자가 잘못된 일을 하게 된 사례는 수없이 많습니다.

> 규칙 12
>
> 운영자가 올바른 일을 쉽게 할 수 있도록 하세요.

과부하 방지

영향을 미치는 최종 공통 기여자는 워크로드의 실제 사용자인 고객입니다. 성공적인 워크로드는 많이 사용되는 경향이 있지만, 때로는 그 사용량이 워크로드의 확장 능력을 능가하기도 합니다. 디스크가 꽉 차거나, 스레드 풀이 고갈되거나, 네트워크 대역폭이 포화되거나, 데이터베이스 연결 한도에 도달하는 등 많은 일이 발생할 수 있습니다.

이러한 문제를 해결할 수 있는 확실한 방법은 없지만 운영 상태 지표를 통해 용량 및 활용도를 사전에 모니터링하면 이러한 고장이 발생할 수 있는 경우 조기 경고를 제공할 수 있습니다. 부하 제거, 회로 차

단기, 지수 백오프 및 지터를 사용한 재시도와 같은 기법은 영향을 최소화하고 성공률을 높이는 데 도움이 될 수 있지만 이러한 상황은 여전히 고장으로 이어집니다. 운영 상태 지표를 기반으로 하는 자동 규모 조정은 과부하로 인한 고장 빈도를 줄이는 데 도움이 될 수 있지만 사용률 변화에 충분히 신속하게 대응하지 못할 수 있습니다.

고객이 지속적으로 사용할 수 있는 용량을 확보하려면 가용성과 비용을 절충해야 합니다. 용량 부족으로 인한 가용성 중단이 발생하지 않도록 하는 한 가지 방법은 각 고객에게 할당량을 제공하고 할당된 할당량을 100% 제공하도록 워크로드 용량을 규모 조정하는 것입니다. 고객이 할당량을 초과하면 속도 제한이 발생하는데, 이는 고장이 아니며 가용성에 영향을 주지 않습니다. 또한 충분한 용량을 프로비저닝하려면 고객 기반을 면밀히 추적하고 향후 활용도를 예측해야 합니다. 이렇게 하면 고객의 과도한 소비로 인해 워크로드가 고장 시나리오로 이어지지 않도록 할 수 있습니다.

- Amazon Builders' Library - 부하 제거를 사용하여 과부하 방지
- Amazon Builders' Library - 다중 테넌트 시스템의 공정성

예를 들어 스토리지 서비스를 제공하는 워크로드를 살펴보겠습니다. 워크로드의 각 서버는 초당 100개의 다운로드를 지원할 수 있고, 고객에게는 할당량 또는 초당 200개의 다운로드를 지원할 수 있으며 고객은 500명입니다. 이 같은 규모의 고객을 지원하려면 서비스가 초당 100,000건의 다운로드 용량을 제공해야 하며, 이를 위해서는 1,000대의 서버가 필요합니다. 고객이 할당량을 초과하면 속도 제한이 발생하여 다른 모든 고객에게 충분한 용량을 제공할 수 있습니다. 이것은 작업 단위를 거부하지 않고 과부하를 방지하는 한 가지 방법의 간단한 예입니다.

결론

이 문서 전체에 걸쳐 고가용성을 위한 12가지 규칙을 세웠습니다.

- 규칙 1: 고장 빈도 감소(MTBF 연장), 고장 감지 시간 단축(MTTD 단축), 수리 시간 단축(MTTR 단축)은 분산 시스템에서 가용성을 개선하는 데 사용되는 세 가지 요소입니다.
- 규칙 2: 워크로드의 소프트웨어 가용성은 워크로드의 전체 가용성을 결정하는 중요한 요소이므로 다른 구성 요소와 마찬가지로 중점을 두어야 합니다.
- 규칙 3: 종속성을 줄이면 가용성에 긍정적인 영향을 미칠 수 있습니다.
- 규칙 4: 일반적으로 가용성 목표가 워크로드 목표와 같거나 더 큰 종속성을 선택하세요.
- 규칙 5: 스페어링을 사용하여 워크로드의 종속성 가용성을 높이세요.
- 규칙 6: 스페어링의 비용 효율성에는 상한선이 있습니다. 필요한 가용성을 달성하는 데 필요한 최소한의 스페어를 활용하세요.
- 규칙 7: 특히 복구 중에는 데이터 영역의 컨트롤 플레인에 대한 종속성을 고려하지 마세요.
- 규칙 8: 가능한 경우 종속성이 손상되더라도 워크로드가 올바르게 작동할 수 있도록 종속성을 느슨하게 결합하세요.
- 규칙 9: 관찰성과 계측은 MTTD 및 MTTR을 줄이는 데 매우 중요합니다.
- 규칙 10: 문제 해결이 아닌 영향 완화에 집중하세요. 정상 작동 상태로 돌아가는 가장 빠른 길을 택하세요.
- 규칙 11: 고장 격리는 전체 고장률을 줄여 영향 범위를 줄이고 워크로드의 MTBF를 증가시킵니다.
- 규칙 12: 운영자가 올바른 일을 쉽게 할 수 있도록 하세요.

워크로드 가용성 개선은 MTTD 및 MTTR을 줄이고 MTBF를 늘려야 합니다. 요약하자면, 기술, 인력, 프로세스를 아우르는 가용성을 개선하기 위한 다음과 같은 방법을 논의했습니다.

- MTTD
  - 고객 경험 지표의 사전 모니터링을 통해 MTTD를 줄이세요.
  - 세분화된 상태 확인을 활용하여 장애 조치를 신속하게 수행할 수 있습니다.
- MTTR
  - 영향 범위 및 운영 상태 지표를 모니터링합니다.
  - 1/재시작, 2/재부팅, 3/이미지 재생성/재배치, 4/교체를 수행하여 MTTR을 줄이세요.
  - 영향 범위를 이해하여 고장을 우회하세요.

  - 가상 머신이나 물리적 호스트를 통한 컨테이너 및 서버리스 기능과 같이 재시작 시간이 더 빠른 서비스를 활용하세요.
  - 가능한 경우 실패한 배포를 자동으로 롤백합니다.
  - 진단 작업 및 재시작 절차를 위한 런북 및 운영 도구를 마련하세요.
- MTBF
  - 소프트웨어가 프로덕션에 출시되기 전에 엄격한 테스트를 통해 소프트웨어의 버그와 결함을 제거합니다.
  - 카오스 엔지니어링과 오류 주입을 구현하세요.
  - 고장을 견딜 수 있도록 종속성을 적절히 절약하세요.
  - 고장 컨테이너를 통해 고장 발생 시 영향 범위를 최소화합니다.
  - 배포 및 변경에 대한 표준을 구현하세요.
  - 단순하고 직관적이며 일관되고 잘 문서화된 운영자 인터페이스를 설계하세요.
  - 운영 우수성을 위한 목표를 설정하세요.
  - 가용성이 워크로드의 중요한 요소인 경우 새 기능 릴리스보다 안정성을 우선시하세요.
  - 제한이나 부하 제거 또는 두 가지 방법을 모두 사용하여 사용량 할당량을 구현하여 과부하를 방지하세요.

고장 예방에 완전히 성공할 수는 없다는 점을 기억하세요. 영향의 범위와 규모를 제한하는 최상의 고장 격리 기능을 갖춘 소프트웨어 설계에 집중하고, 이상적으로는 그 영향을 “가동 중지” 임계값 이하로 유지하고 매우 빠르고 매우 안정적인 탐지 및 완화에 투자하세요. 현대의 분산 시스템은 여전히 고장을 피할 수 없는 상황으로 받아들이고 고가용성을 위해 모든 수준에서 설계되어야 합니다.

부록 1 - MTTD 및 MTTR 중요 지표

다음은 이벤트 발생 시 MTTD 및 MTTR을 줄이는 데 도움이 될 수 있는 계측 및 관찰성 표준화의 프레임워크입니다.

고객 경험 지표 이러한 지표는 서비스가 응답성이 뛰어나고 고객 요청을 처리할 수 있음을 반영합니다. 컨트롤 플레인 지연 시간을 예로 들 수 있습니다. 이러한 지표는 오류율, 가용성, 지연 시간, 볼륨 및 제한 속도를 측정합니다.

영향 평가 지표 이러한 지표는 이벤트 발생 시 영향의 범위에 대한 통찰력을 제공합니다. 데이터 영역 이벤트의 영향을 받은 고객 수 또는 비율을 예로 들 수 있습니다. 영향을 받는 항목의 수 또는 비율을 측정합니다.

운영 상태 지표 이러한 지표는 서비스가 대응력이 뛰어나고 고객 요청을 처리할 수 있지만 공통 인프라 하위 시스템 및 리소스에 초점을 맞추고 있음을 반영합니다. EC2 플릿의 CPU 사용률 비율을 예로 들 수 있습니다. 이러한 지표는 사용률, 용량, 처리량, 오류율, 가용성 및 지연 시간을 측정해야 합니다.

기여자

다음은 이 문서의 기여자입니다.

- Michael Haken, Principal Solutions Architect, Amazon Web Services

참조 자료

자세한 내용은 다음을 참조하세요.

- Well-Architected 신뢰성 요소
- Well-Architected 운영 우수성 요소
- Amazon Builders' Library - 배포 중 롤백 안전성 보장
- Amazon Builders' Library - 99.999%를 넘어서: 가용성이 가장 높은 데이터 플레인에서 얻은 교훈
- Amazon Builders' Library - 안전한 수동 배포 자동화
- Amazon Builders' Library - 탄력적인 서버리스 시스템을 대규모로 설계 및 운영
- Amazon Builders' Library - 고가용성 배포에 대한 Amazon의 접근 방식
- Amazon Builders' Library - 탄력적인 서비스를 구축하기 위한 Amazon의 접근 방식
- Amazon Builders' Library - 성공적인 실패에 대한 Amazon의 접근 방식
- AWS 아키텍처 센터

문서 이력

이 백서에 대한 업데이트 알림을 받으려면 RSS 피드를 구독하면 됩니다.

변경 사항	설명	날짜
최초 게시	백서가 처음 게시되었습니다.	2021년 11월 12일

> **Note**
> RSS 업데이트를 구독하려면 사용 중인 브라우저에서 RSS 플러그인을 활성화해야 합니다.

고지 사항

고객은 본 문서의 정보를 독립적으로 평가할 책임이 있습니다. 본 문서는 (a) 정보 제공의 목적으로만 제공되고, (b) 사전 통지 없이 변경될 수 있는 현재 AWS 제품 및 관행을 나타내고, (c) AWS 및 그 계열사, 공급업체 또는 라이선스 제공자로부터 어떠한 약속이나 보증도 하지 않습니다. AWS 제품 또는 서비스는 명시적이든 묵시적이든 어떠한 종류의 보증, 진술 또는 조건 없이 '있는 그대로' 제공됩니다. 고객에 대한 AWS의 책임 및 채무는 AWS 계약에 준거합니다. 본 문서는 AWS와 고객 간의 어떠한 계약도 구성하지 않으며 이를 변경하지도 않습니다.

AWS 용어집

최신 AWS 용어는 AWS 용어집 참조서의 AWS 용어집을 참조하세요.