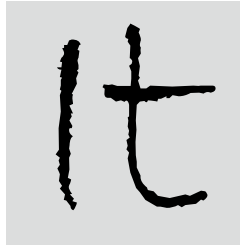


# Release

성공적인 출시를 위한  
소프트웨어 설계와 배치



It

위키북스

# 목 차

역자 서문

저자 · 역자 소개

서문

<b>1 장</b>	<b>소개</b>	<b>1</b>
1.1	올바른 목표를 향해서.....	2
1.2	예지력을 사용하라 .....	3
1.3	삶의 질.....	4
1.4	도전의 범위.....	5
1.5	여기도 백만 달러, 저기도 백만 달러.....	6
1.6	실용주의 아키텍처 .....	7
<b>2 장</b>	<b>사례연구: 항공사를 정지시킨 예외(Exception) 사건</b>	<b>11</b>
2.1	정지 사태 .....	13
2.2	결과.....	17
2.3	사후검토 .....	19
2.4	확실한 증거.....	25
2.5	약간의 예방.....	28
<b>3 장</b>	<b>안정성 소개</b>	<b>31</b>
3.1	안정성이란? .....	33
3.2	고장 유형 .....	36

3.3	크랙 전파 .....	36
3.4	고장의 연쇄(chain of failure) .....	38
3.5	패턴과 안티패턴 .....	40
<b>4 장</b>	<b>안정성 안티패턴</b>	<b>43</b>
4.1	통합지점(integration point) .....	46
	소켓(socket) 기반 프로토콜 .....	46
	새벽 5시 문제 .....	51
	HTTP 프로토콜 .....	58
	벤더 API 라이브러리.....	60
	통합지점 문제에 맞서기 .....	62
4.2	연쇄 반응 .....	64
4.3	연속적인 고장.....	69
4.4	사용자들 .....	72
	트래픽.....	72
	서비스에 비용이 많이 든다.....	76
	바람직하지 않은 사용자들 .....	78
	악의적인 사용자 .....	85
4.5	블록된 스레드.....	88
	블로킹 지점을 찾아라.....	91
	서드파티 라이브러리.....	95
4.6	자기부정 공격.....	98
4.7	확장 효과 .....	102
	일대일 통신.....	102
	공유된 자원.....	104
4.8	불균형 용량.....	108
	테스트하여 문제를 찾아내자.....	111
4.9	느린 응답 .....	113
4.10	SLA 역전 .....	115

4.11	끝이 없는 쿼리 결과.....	120
	검은 월요일.....	120
<b>5 장</b>	<b>안정성 패턴</b>	<b>125</b>
5.1	제한시간을 사용하라.....	126
5.2	차단기.....	131
5.3	칸막이.....	135
5.4	정상 상태.....	140
	데이터 비우기.....	141
	로그 파일들.....	143
	메모리 내(In-Memory) 캐싱.....	146
5.5	빠른 고장.....	148
5.6	핸드셰이킹.....	152
5.7	테스트 하니스.....	155
5.8	분리하는 미들웨어.....	160
<b>6 장</b>	<b>안정성 요약</b>	<b>165</b>
<b>7 장</b>	<b>사례연구: 고객에게 짓밟히다</b>	<b>169</b>
7.1	카운트다운과 런칭.....	169
7.2	QA를 향해.....	171
7.3	부하 테스트.....	175
7.4	균중에 의한 살인.....	179
7.5	테스트와의 차이.....	182
7.6	사고 여파.....	183

<b>8 장</b>	<b>용량 소개하기</b>	<b>187</b>
8.1	용량 정의하기.....	188
8.2	제한조건.....	189
8.3	상관관계.....	192
8.4	확장성.....	192
8.5	용량에 대한 미신.....	194
	CPU는 저렴하다.....	195
	스토리지는 저렴하다.....	197
	대역폭은 저렴하다.....	200
8.6	요약.....	203
<b>9 장</b>	<b>용량 안티패턴</b>	<b>205</b>
9.1	리소스 풀 경쟁.....	206
9.2	지나친 JSP 프라그먼트(fragment).....	210
9.3	AJAX 대량살상.....	212
	상호작용 설계(interaction design).....	213
	요청 시점.....	213
	세션 스래싱(session thrashing).....	214
	응답 포맷.....	214
9.4	너무 오래 머무는 세션.....	216
9.5	HTML 안에 낭비된 공간.....	219
	공백문자(whitespace).....	220
	비싼 스페이서 이미지(spacer image).....	221
	HTML 테이블 과잉.....	223
9.6	새로고침 버튼.....	224
9.7	손으로 만든 SQL.....	227
9.8	데이터베이스 부영양화.....	230
	인덱싱.....	230



16.11	진정국면 .....	318
<b>17 장</b>	<b>투명성</b>	<b>319</b>
17.1	관점 .....	321
	이력정보 .....	322
	미래 예측 .....	323
	현재 상태 .....	325
	악명 높은 계기판 .....	327
	순간 상태 .....	329
17.2	투명성을 위한 설계 .....	331
17.3	권능을 부여하는 기술 .....	332
17.4	로깅(logging) .....	333
	설정 .....	334
	로깅 수준 .....	334
	메시지 목록 .....	336
	인간적 요인 .....	338
	마지막 주의 사항 .....	342
17.5	모니터링 시스템 .....	342
	상용 모니터링 시스템 .....	345
	상용 시스템 간의 차이 .....	345
	모니터링 시스템 설계 .....	347
17.6	명실상부한 표준 .....	349
	단순 네트워크 관리 프로토콜(SNMP) .....	349
	CIM .....	353
	JMX .....	353
	무엇을 노출할 것인가 .....	358
	JMX와 SNMP를 같이 .....	359
17.10	운영 데이터베이스 .....	360
	운영 데이터베이스의 상위 수준 구조 .....	363
	데이터베이스에 입력하기 .....	365

	운영 데이터베이스의 사용 .....	366
17.8	지원 프로세스들 .....	368
	관측의 핵심 .....	370
17.9	요약 .....	372
<b>18 장</b>	<b>적용</b>	<b>375</b>
18.1	시간이 흐르면서 적용하기 .....	376
18.2	적용 가능한 소프트웨어 설계 .....	377
	의존성 주입 .....	378
	객체 설계 .....	379
	XP코딩 프랙티스 .....	383
	애자일 데이터베이스 .....	384
18.3	적용 가능한 엔터프라이즈 아키텍처 .....	386
	시스템 내부의 의존성 .....	389
	시스템 사이의 의존성: 프로토콜 .....	391
	시스템 사이의 의존성: 데이터베이스 .....	394
18.4	릴리스가 고통을 겪어서는 안 된다 .....	395
	배치는 비용이 너무 많이 든다 .....	397
	적당한 시기의 릴리스 .....	398
	무정지 배치 .....	399
	확장하기 .....	400
	배치작업 .....	403
	정리하기(clean up) .....	403
18.5	요약 .....	404
부록	.....	407
참고 문헌	.....	407
베타리더 리뷰 후기	.....	411
찾아보기	.....	417

## 2장



# 사례연구: 항공사를 정지시킨 예외(Exception) 사건

매우 사소한 것에서 시작하여 커다란 이슈로 변지는 사건을 본 적이 있는가? 사소한 프로그래밍 에러가 내리막으로 눈덩이를 굴리기 시작한다. 눈덩이의 운동량이 증가하면서 문제가 점점 커진다. 한 유명한 항공사가 이와 비슷한 사건을 겪었다. 결국 이 사건 때문에 수천 명의 승객이 오도가도 못했으며 회사는 수십 만 달러의 손해를 봤다. 이 사건이 어떻게 일어났는지 살펴보자.

이 사건은 핵심 설비(Core Facilities, CF)<sup>1</sup>에 사용되는 데이터베이스 클러스터의 계획된 페일 오버(failover)<sup>2</sup>에서 시작되었다.<sup>3</sup> 해당 항공사는 재사용을 높이고 개발 시간을 단축하며 운영 비용을 줄인다는 일반적인 목표를 지닌 서비스 지향 아키텍처(service-oriented architecture,

1 이 항공사에서 사용하는 고유한 시스템을 말한다. 즉, 문제를 일으킬 만한 주요 시스템을 가리킨다.

2 페일오버는 1차 시스템이 고장 또는 정기 유지보수 등의 이유로 이용할 수 없는 상태가 되었을 때, 2차 시스템이 즉시 그 임무를 넘겨받아 프로세서, 서버, 네트워크, 데이터베이스 등과 같은 시스템 구성요소의 기능들이 중단 없이 유지될 수 있는 백업 운전 모드를 말한다.

3 언제나 그렇듯이, 관련된 사람들과 회사들의 비밀을 유지하기 위해 모든 이름, 장소, 날짜를 변경했다.

SOA)로 바꾸고 있었다. 이 무렵에, CF가 서비스 지향 아키텍처의 첫 번째 대상이었다. CF팀은 기능에 따라 단계별 공개를 계획하였다. 대부분의 큰 회사는 이 항공사가 진행하는 프로젝트에서 약간 변형된 형태로 프로젝트를 진행하기 때문에, 괜찮은 계획이었으며 익숙하게 들릴 것이다.

CF는 항공사 애플리케이션의 매우 일반적인 서비스인 항공편 검색을 처리했다. 날짜, 시간, 도시, 공항 코드, 항공기 번호나 이런 정보의 조합이 주어지면, CF는 상세한 항공편정보 목록을 찾아서 돌려 준다. 이 사건이 일어났을 때, 셀프 서비스 체크인 키오스크(kiosk)<sup>4</sup>, IVR<sup>5</sup> 그리고 ‘채널 파트너’ 애플리케이션은 CF를 사용하도록 수정되었다. 채널 파트너 애플리케이션은 대형 여행 예약사이트에 데이터 피드(feed)를 생성하였다. IVR과 셀프 서비스 체크인 키오스크는 항공기에 승객을 탑승시키는 데 사용되었다. 즉, 돈을 받는 데 쓰였다. 개발일정에는 탑승구 직원(gate agent) 애플리케이션과 콜 센터 애플리케이션을 새롭게 릴리스하면서 항공편 검색을 위해 CF를 사용하도록 변경되는 계획이 잡혀 있었지만, 새로운 릴리스는 공개되지 않았다. 뒤에서 살펴보겠지만 출시되지 않은 편이 더 나았다.

CF를 설계한 아키텍트는 이 시스템이 얼마나 중요한지 알고 있었다. CF 아키텍트는 높은 가용성을 지니도록 시스템을 설계하였다. CF는 이중화된 오라클 9i 데이터베이스와 J2EE 애플리케이션 클러스터에서 운영되었다. 모든 데이터는 거대한 외부 레이드 어레이(RAID array)에 저장되었는데, 이 레이드 어레이는 매일 두 번씩 외부에 있는 테이프 백업을 실시하였고 보조 새시에 있는 온디스크(on-disk) 복제는 최대로 5분 전 데이터를 보장했다.

오라클 데이터베이스 서버는 어느 한순간에 클러스터에 있는 하나의 노드에서만 실행되었는데, 베리타스 클러스터 서버(Veritas Cluster Server)<sup>6</sup>가 오라클 데이터베이스 서버를

4 키오스크는 무인 자동처리 시스템이다.

5 대화식 음성 응답(Interactive Voice Response, IVR): 끄찍한 전화 메뉴 시스템이다.

6 시만텍의 서버 클러스터링 제품이다.

통제하였으며, 가상IP 주소가 할당되어 레이드 어레이로부터 파일시스템을 올리거나(mount) 내렸다(unmount). 앞단에는, 한 쌍의 이중화된 하드웨어 로드 밸런서(load balancer)가 애플리케이션 서버 가운데 하나로 들어가는 트래픽을 관리하였다. 셀프 서비스 체크인 키오스크나 IVR 시스템과 같은 호출하는 애플리케이션은 앞단에 있는 가상IP 주소에 연결되었다. 여기까지는 괜찮다.

웹사이트나 웹 서비스 작업을 해봤다면, 다음 쪽 그림 2.1은 익숙할 것이다. 매우 일반적인 고가용성의 아키텍처이며, 좋은 아키텍처다. CF는 일반적인 단일지점고장 문제<sup>7</sup>를 겪지 않았다. CPU, 팬, 드라이브, 네트워크 카드, 전원 공급장치, 네트워크 스위치 같은 모든 하드웨어들이 이중화되었기 때문이다. 모든 서버는 하나의 랙(rack)이 고장나거나 훼손되었을 경우를 대비해 서로 다른 랙으로 분리되어 있었다. 또한 30마일 떨어진 두 번째 장소에는 화재, 홍수, 폭격, 운석 충돌의 경우를 대비해 운영을 넘겨받을 준비까지 되어 있었다

## 2.1 정지 사태

내가 말했던 대부분의 대형 고객의 경우처럼, 로컬 팀 엔지니어들은 항공사의 인프라 스트럭처를 운영하는 데 헌신을 다했다. 사실, 로컬 팀은 이 사건이 일어나기 3년 이상 전부터 이 작업을 전적으로 수행해 왔다. 이 일이 벌어진 그 밤에, 로컬 엔지니어는 CF 데이터베이스 1에서 CF 데이터베이스 2로 데이터베이스 페일오버를 수동으로 실행했다(다음 쪽 그림 2.1을 보자). 로컬 엔지니어는 사용하는 데이터베이스를 바꾸려고 베리타스를 사용하였다. 베리타스 덕분에 엔지니어들은 CF 데이터베이스 1 서버에서 정기적인 유지보수 작업을 할 수 있었다. 늘 있는 정기적인 작업이었다. 엔지니어들은 과거에도 수십 번 이 절차를 밟았었다.

7 단일지점고장(single-point-of-failure)은 시스템의 한 부분에 문제가 생겼을 때, 시스템 전체가 마비되는 것을 의미한다.

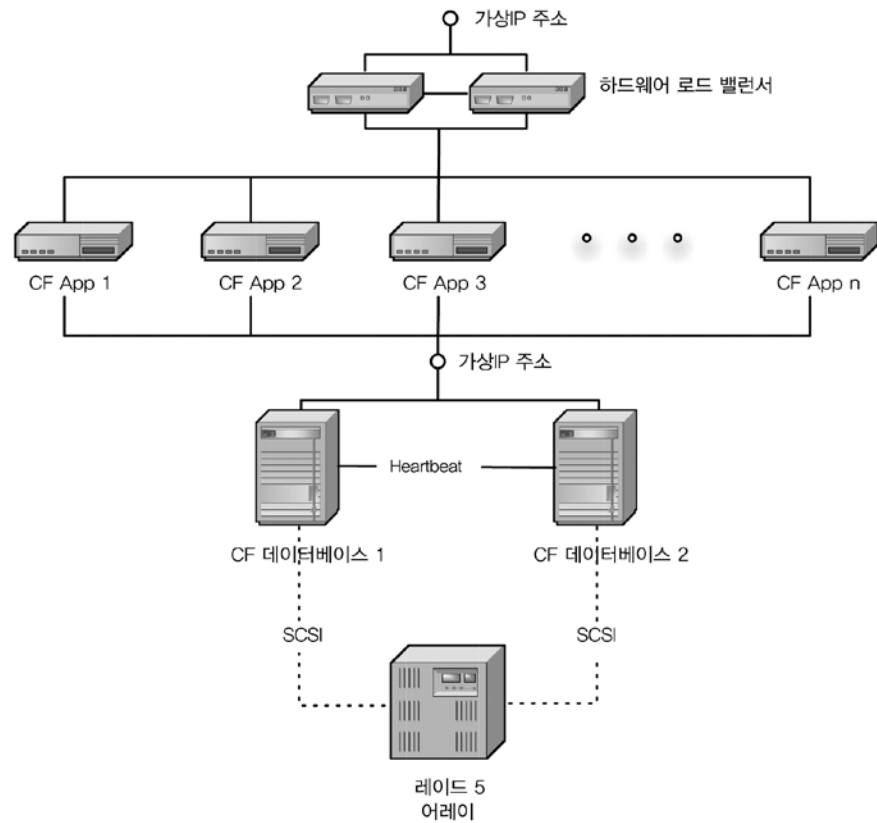


그림 2.1 | CF 배치 아키텍처

베리타스 클러스터 서버는 페일오버를 조정했다. 일부 안에, 베리타스 클러스터 서버는 데이터베이스 1에 있는 오라클 서버를 종료하고, 레이드 어레이에서 파일시스템을 내린(unmount) 다음, 데이터베이스 2에 파일시스템을 올리고(mount), 데이터베이스 2에서 오라클을 시작했으며, 데이터베이스 2에 가상IP 주소를 재할당하였다. 애플리케이션 서버는 가상IP 주소로만 연결하도록 설정되었기 때문에, 애플리케이션 서버는 어떤 것이 바뀌었는지 구별할 수 없었다.

고객은 미국 태평양 시간 기준으로 목요일 밤 11쯤에 이 변경 작업을 잡아두었다.<sup>8</sup> 로컬 팀에 있는 엔지니어 가운데 한 명이 이 변경 작업을 실행하기 위해 운영 센터와 함께 작업했다. 모든 것이 계획한 대로 진행되었다. 이들은 데이터베이스 1에서 데이터베이스 2로 운영 데이터베이스를 옮기고 나서, 데이터베이스 1을 갱신했다. 데이터베이스 1이 올바르게 수정되었다는 것을 두 번 검사한 뒤, 이들은 데이터베이스를 다시 데이터베이스 1로 이관하고 나서 같은 변경을 데이터베이스 2에 적용했다. 작업 시간 내내, 정기적인 사이트 모니터링은 애플리케이션을 계속해서 사용할 수 있다는 것을 보여주었다. 이 변경작업 중에 어떤 서비스 중지도 계획하지 않았고 일어나지도 않았다. 오전 12시 30분쯤, 엔지니어는 변경이 '완료, 성공'했다고 기록한 후 서명했다. 22시간 근무를 마친 후 로컬 엔지니어는 잠을 자러 갔다. 더블 에스프레소에 의존해 겨우 끝낸 긴 작업이었다.

두 시간이 지난 후에야 이상한 일이 일어났다.

오전 2시 30분쯤, 모니터링 콘솔에 모든 체크인 키오스크의 상태에 빨간불이 들어왔다. 즉, 동시에 전국에 있는 모든 키오스크가 요청에 대한 서비스를 중단하였다. 몇 분 지나서, IVR 서버에도 빨간 불이 들어왔다. 정확하게 사용량이 가장 많은 시간은 아니었지만, 이 시간대에 매우 가까웠는데, 왜냐하면 태평양 시간으로 오전 2시 30분은 동부 시간으로 오전 5시 30분이었기 때문이다. 동부 해안에서 통근비행기 이용자들이 체크인하는 주요 시간대였다. 운영 센터는 즉시 '심각도 1'을 발령하고 로컬 팀을 전화회의에 호출했다.

어떤 사태이든 간에, 나의 첫째 우선순위는 항상 서비스를 복구하는 것이다. 서비스 복구는 조사보다 우선한다. 사후에 근본원인 분석(root cause analysis)을 위한 데이터를 모을 수 있다면 다행스럽다. 데이터 수집이 서비스 정지 시간을 늘리지 않는다면 말이다. 큰 소동이 일어나면, 즉흥적인 행동은 도움이 되지 않는다. 다행히도, 팀은 모든 자바 애

<sup>8</sup> 미국은 태평양, 산악, 중부, 동부 시간대를 사용한다. 태평양 시간대가 낮 12시면, 산악 시간대는 아침 11시, 중부는 10시, 동부는 9시가 된다. 미국 동부에 있는 뉴욕과 서부에 있는 LA의 시차는 3시간이다.

플리케이션의 스레드 덤프(thread dump)와 데이터베이스 스냅샷을 저장하는 스크립트를 오래 전에 만들어 두었다. 이런 형태의 자동화된 데이터 수집은 완벽한 균형을 제공한다. 즉, 이것은 즉흥적이지 않고 정지를 길게 끌지도 않으며 사후 분석에 도움을 주기 때문이다. 절차에 따라 운영 센터는 이러한 스크립트를 바로 실행했다. 또 팀은 키오스크 애플리케이션 서버 가운데 하나를 재가동하려고 시도했다.

서비스를 복구하는 비법은 무엇을 대상으로 할지 알아내는 것이다. 계층(layer)별로 모든 서버 각각을 다시 시작하면 항상 '서비스를 다시 시작'할 수 있다. 이 방법은 거의 대부분 효과적이지만, 시간이 오래 걸린다. 대개, 서버를 실제로 멈춘 단 하나의 범인을 찾을 수 있다. 이런 방법은 의사가 병을 진단하는 상황과 비슷하다. 여러분은 모든 알려진 병에 대해서 환자를 치료할 수 있지만, 이것은 고통스럽고, 비싸고, 더디다. 대신에 치료할 병이 무엇인지 정확하게 알아내기 위해 환자가 보이는 증상을 살펴봐야 한다. 문제는 개별적인 증상으로 충분히 명확하지 않을 때의 경우다. 물론 가뭄에 콩 나듯이 어떤 증상이 근본 문제를 직접적으로 알려주기도 하지만, 대개 그렇지 않기 때문이다. 대부분의 경우 여러분은 열이 있다는 증상만으로 아무것도 알 수 없다.

수백 가지의 병이 열을 동반한다. 가능성이 있는 원인을 구별해 내려면 시험과 관찰로 더 많은 정보를 얻어야 한다.

이번 사건에서 팀은 분리된 두 개의 애플리케이션이 전혀 응답이 없는 상황에 직면했다. 이 일은 거의 동시에 일어났는데, 키오스크와 IVR 애플리케이션이 사용하는 별도의 모니터링 도구 사이의 지연시간(latency) 정도로 여길 만큼 가까웠다. 가장 확실한 가정은 두 종류의 애플리케이션이 사용하는 제3의 요소에 문제가 있다는 것이다. 18쪽 그림 2.2에서 볼 수 있는 것처럼 큰 화살이 CF로 향했는데, CF는 키오스크와 IVR 시스템이 서로 공유하는 유일한 공통 모듈이기 때문이다. 이 문제가 생기기 3시간 전에 데이터베이스 패일오버를 CF에 적용했다는 사실 역시 CF를 유력한 용의자로 만들었다. 하지만 모니터링에서는 CF에 문제가 있다고 나타나지 않았다. 모아둔 로그 파일에서 어떤 문제도 나타나지 않았으며, URL 탐침(probing)에서도 문제가 없었다. 나중에 밝

혀진 사실이지만, 모니터링 애플리케이션은 단지 상태 페이지만 갱신했기 때문에, CF 애플리케이션 서버가 실제 정상적으로 작동하는지에 대해 사실 많은 것을 알려주지 못했다. 우리는 나중에 정상적인 경로로 모니터링 애플리케이션 문제를 해결할 수 있도록 기록을 남겼다.

서비스를 복구하는 것이 우선순위가 가장 높다는 것을 명심하자. 이 정지 사태는 한 시간의 SLA<sup>9</sup> 제한에 가까워졌기 때문에, 팀은 CF 애플리케이션 서버 각각을 다시 시작하기로 결정했다. 이들이 첫째 CF 애플리케이션 서버를 다시 시작하자마자, IVR 시스템이 다시 복구되기 시작했다. 모든 CF 서버가 다시 시작되자, IVR 상태는 녹색이 되었지만 키오스크에는 여전히 빨간색이 표시되고 있었다. 수석 엔지니어는 직감적으로 키오스크의 애플리케이션 서버를 다시 시작하기로 결정했다. 이 방법이 적중했다. 키오스크와 IVR 시스템 모두 현황판에 녹색으로 표시되었다.

이 사건에 대한 전체 경과 시간은 3시간 정도였다. 미국 태평양 시간으로 오후 11시 30분부터 오전 2시 30분까지였다.

## 2.2 결과

특별히 몇 개의 역사적인 정지 사태(예를 들어, 1999년에 있었던 eBay의 24시간 정지상태가 떠오른다)와 비교를 한다면, 3시간은 그리 길지 않은 듯하다. 비록 사태의 영향은 단지 3시간 정도 지속되었지만, 항공사에는 구식 시스템을 사용해서 모든 사람을 수속하기에 충분한 탑승구 직원이 없었다. 항공사는 키오스크가 정지하자 비번인 직원들을 불러 들였다. 직원 가운데 일부는 그 주에 40시간이 넘겨 일했기 때문에, 노조 계약 초과근무(고정급의 1.5배)가 발생했다. 비번인 직원도 사람일 뿐인지라, 항공사가 현장 직원을 많이 확보할 때까지 이들은 적체된 수속만을 다룰 수 있었다. 오후 3시

9 서비스수준 협약(Service-level agreement): 서비스 제공자와 고객 사이의 계약이다. 일반적으로 SLA를 위반하면 상당한 위약금이 청구된다.

가 되어서야 적체된 수속을 모두 처리하였다.

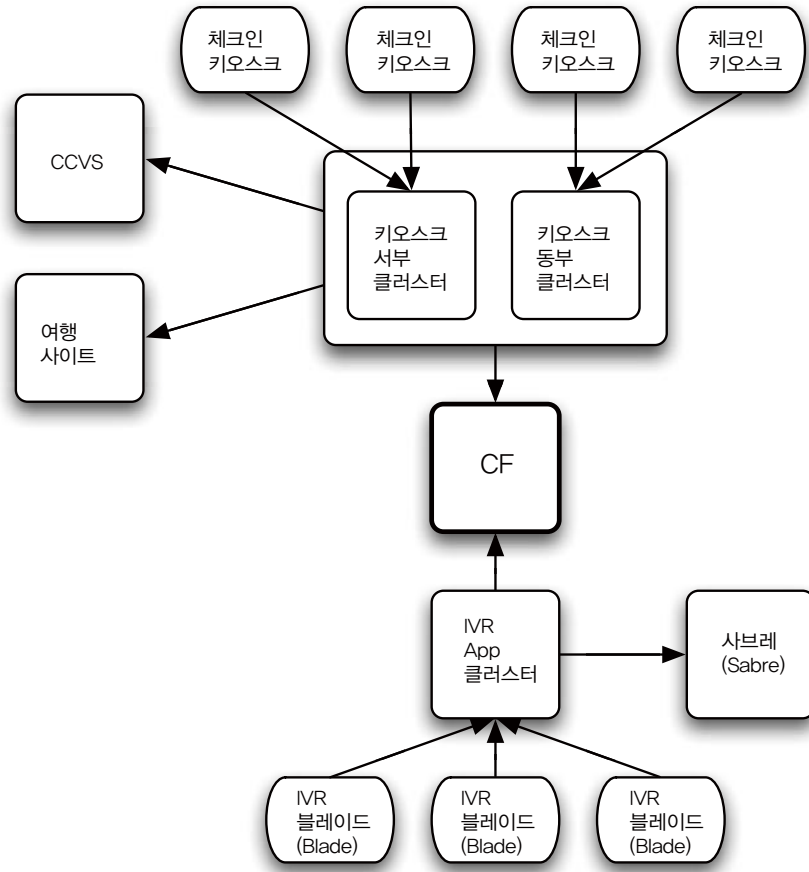


그림 2.2 | 일반적인 의존성

새벽 비행편에서 수속에 너무 오랜 시간이 걸렸지만 그렇다고 탑승구에서 비행기를 그냥 출발시킬 수 없었다. 만일 그랬다면 비행기의 반은 비었을 것이다. 많은 여행객들의 출발이나 도착이 지연되었다. 목요일은 우연히도 '너드버드(nerd-bird)'<sup>10</sup> 항공편이 많은 날과 겹쳤다. 집으로 돌아가는 컨설턴트들이 탑승한 통근 비행기편이 많은 날이었다. 탑승구를 모두 사용하고 있었기 때문에, 들어오는 비행기는 비어 있는 다른 탑승구로 이동해야 했다. 따라서 이미 체크인한 여행객들도 불편을 겪었다. 원래 탑승구에서 변경된 탑승구로 달려가야 했던 것이다.

이 연착사고는 'Good Morning America'<sup>11</sup>(오도가도 못하는 불쌍한 싱글 맘(single mom)과 아기들의 내용을 담은 화면도 보여주었다)와 'the Weather Channel'<sup>12</sup>에서 여행 안내 방송으로도 방영되었다.

연방항공청(Federal Aviation Administration, FAA)은 항공사의 연간 보고서의 일부로 정시 도착과 출발 여부를 측정한다. 이들은 고객이 FAA에 보낸 항공사에 대한 불만도 측정한다.

항공사 CEO의 연봉은 일부분 FAA 연간보고서를 기초로 삼는다.

CEO가 세인트 토마스(St. Thomas)<sup>13</sup>에 있는 자신의 별장 구입비를 날려버린 사람을 찾기 위해, 운영 센터를 뒤지고 다니는 것을 본다면 일정 사나운 날이 될 것이다.

### 2.3 사후검토

태평양 시간 오전 10시 30분, 정지 사태가 시작되고 8시간 후, 고객담당 대표인 톰이 나를 불러서 사후검토를 요청했다. 데이터베이스 페일오버와 유지보수가 끝난 다음 고장이 곧바로 생겼기 때문에, 의심은 자연스럽게 이 작업에 쏠렸다. 운영을 할 때, 대부

<sup>10</sup> 너드(nerd)는 '컴퓨터 마니아'를 뜻한다. '새너제이와 시애틀 사이'나 '새너제이와 오스틴 사이'를 오가는 승객들이 주로 하이테크산업 종사자이기 때문에, 이들이 이용하는 항공편을 너드버드(nerd-bird)라 부른다.

<sup>11</sup> 미국 ABC TV에서 방영하는 아침 정보 프로그램.

<sup>12</sup> 'the Weather Channel'은 북 남미의 약 9천 5백만 케이블 TV 시청 가구에 날씨 정보를 제공한다.

<sup>13</sup> 세인트 토마스(St. Thomas)는 버진 아일랜드에서 가장 개발이 잘된 휴양지다.

본의 경우 “post hoc, ergo propter hoc”<sup>14</sup>은 시작하기 좋은 시작점이다. 이 방법이 항상 옳지는 않지만, 검토를 시작할 만한 지점을 확실히 제공한다. 사실은 톰이 나를 불렀을 때, 그곳으로 날아가서 데이터베이스 페일오버가 왜 정지 사태를 발생시켰는지 알아보라고 요청했다.

나는 비행기에 앉아, 노트북에 있는 문제 접수서와 예비 사건보고서를 살펴보기 시작했다.

안건(agenda)은 간단했다. 사후검토 조사를 실행하여 다음 질문에 답할 것.

- 데이터베이스 페일오버가 정지 사태를 일으켰는가? 그렇지 않다면 무엇이 정지 사태를 일으켰는가?
- 클러스터는 올바르게 설정되었는가?
- 운영 팀은 유지보수를 올바르게 수행했는가?
- 고장이 정지 사태로 번지기 전에 고장을 어떻게 탐지해낼 수 있나?
- 가장 중요한 것으로, 이러한 정지 사태가 결코 다시 생기지 않으려면 어떻게 해야 하나?

물론 내가 그곳에 가는 것은 우리가 이 정지 사태를 심각하게 생각한다는 것을 고객에 보여주는 면도 있었다. 말할 것도 없이, 내 조사는 로컬 팀이 사건을 덮어버릴 걱정을 없애야 했다. 물론 로컬 팀은 이런 짓을 결코 하지 않았겠지만, 주요한 사건 후에 알게 된 내용을 관리하는 것은 사건 그 자체를 관리하는 것만큼 중요하다.

**주요한 사건 후 알게 된 내용을 관리하라. 이것은 사건 그 자체를 관리하는 것만큼 중요하다.**

사후검토는 살인미스터리와 비슷하다. 단서 몇 가지가 있다. 어떤 단서는 정지 사태 당시의 서버 로그파일 복사본처럼 믿을 만하다. 다른 단서는 사람들이 목격한 것에 대한 진술처럼 신뢰할 수 없다. 진짜 살인 사건의 목

14 글자 그대로 보면 “이것 이후에, 따라서 이것 때문에”이란 뜻이지만, 인접한 타이밍이라는 것에 기초해서 인과관계를 부여하는 일반적인 논리상의 오류를 뜻한다. “네가 마지막으로 건드렸잖아.”라는 표현으로도 알려져 있다.

격자들과 마찬가지로, 사람들은 관찰과 추측을 혼동하는 경향이 있다. 사실인 양 가설을 설명한다. 시체가 사라져 버리기 때문에, 진짜로 사후검토는 살인 사건보다 해결하기 더 어렵다. 서버는 정상으로 돌아와서 운영되기 때문에, 부검할 시신은 존재하지 않는다. 고장이 일어났던 당시 서버의 상태가 어떠했든 간에 그 상태는 더 이상 존재하지 않는다. 고장은 사건 당시에 채취해둔 로그파일이나 모니터링 데이터 안에 흔적을 남겨두었을 수도 있으며, 그렇지 않을 수도 있다. 실마리를 찾기란 매우 어려워 보인다.

파일들을 읽으면서 모아 둔 데이터에 대해서 메모를 남겼다. 애플리케이션 서버에서 로그파일, 스프레드 덤프, 설정 파일을 얻어야 했다. 데이터베이스 서버에서는 데이터베이스와 클러스터 서버에 대한 설정파일이 필요했다. 현재 설정 파일과 매일 밤 백업한 파일을 비교하자는 메모도 남겼다. 백업은 정지 사태가 일어나기 전에 실행되었기 때문에, 이 파일들은 백업했을 때와 내가 조사하는 사이에 설정 값들이 변경되었는지를 알려줄 것이다. 다시 말해, 이것들은 누군가가 실수를 은폐하려고 시도했는지를 알려줄 것이다.

내가 호텔에 도착했을 무렵, 시간은 자정을 넘었고 몹시 피곤했다. 내가 간절히 원하는 것은 샤워와 숙면이었지만, 대신 나는 고객담당 책임자와 미팅을 가졌다. 그는 내가 비행기에서 있었던 동안 일어난 조치상황에 대해서 브리핑해주었다. 결국 나의 하루는 새벽 1시쯤 끝났다.

아침에 커피 한 잔으로 기운을 북돋우고, 데이터베이스 클러스터와 레이드 설정을 세심하게 조사했다. 클러스터에 대해서 일반적인 문제를 살펴보았다. 즉, 하트비트(heartbeat)<sup>15</sup>가 충분했는지, 스위치에 들어가는 하트비트가 실제 트래픽을 실었는지, 서버가 가상주소 대신 물리적인 IP 주소를 사용하도록 설정되었는지, 관리되는 패킷(managed packet)사이에서 나쁜 의존성이 있었는지 등을 조사했다. 그 당시에는 체크리스트가 없었는데, 내가 한 번 이상 본 문제들이거나 말로 전해 들었던 것이다. 잘못된 것이 없었다. 엔지니어링 팀은 데이터베이스 클러스터에 대해서 훌륭한 작업을 했었다. 증명된 교과

15 서버가 정상적인지를 나타내는 신호다.

서적인 작업이었다. 사실, 스크립트의 일부는 베리타스 자체의 교재에서 직접 가져온 것처럼 보였다.

다음으로, 애플리케이션 서버의 설정으로 넘어갈 차례였다. 로컬 엔지니어는 키오스크 애플리케이션 서버에서 정지 사태 동안의 모든 로그파일의 복사본을 만들어두었다. 나는 CF 애플리케이션 서버에서 로그파일을 얻을 수도 있었다. CF 애플리케이션 서버에는 정지 사태가 일어난 시간 대의 로그파일이 있었는데, 이 파일은 그 전날 것이었다. 게다가 두 로그파일 안에는 스레드 덤프가 있었다. 오랜 경력의 자바 프로그래머로서, 나는 애플리케이션 행(application hang)<sup>16</sup>을 디버깅하는 데 자바 스레드 덤프를 애용했다.

스레드 덤프로 무장하고 스레드 덤프를 어떻게 읽는지 안다면, 애플리케이션을 분석하는 건 식은 죽 먹기다. 여러분이 소스코드를 한 번도 보지 못한 애플리케이션도 상당 부분 추론할 수 있다. 애플리케이션이 어떤 서드파티 라이브러리(third-party library)를 사용하는지, 어떤 종류의 스레드 풀(thread pool)을 갖고 있는지, 애플리케이션마다 얼마나 많은 스레드가 있는지, 애플리케이션이 사용하는 백그라운드 프로세싱이 무엇인지 알 수 있다. 각 스레드의 스택 트레이스(stack trace) 안에 있는 클래스와 메서드를 살펴봄으로써, 애플리케이션이 무슨 프로토콜을 사용하는지조차 알 수 있다.

CF 안의 문제가 무엇인지 결정하는 데 그리 오래 걸리지 않았다. 키오스크 애플리케이션 서버에 있는 스레드 덤프는 사건이 발생한 동안 내가 관찰한 동작으로부터 예측했던 것을 정확하게 보여주었다. 개별적인 키오스크로부터 온 요청을 처리하는 데 할당된 40개의 스레드 모두가 `SocketInputStream.socketRead0( )` 안에서 블록되었다(`SocketInputStream.socketRead0( )`는 자바 소켓 라이브러리 내부에 있는 네이티브 메서드(native method)다). 스레드는 영원히 돌아오지 않는 응답을 읽으려고 헛된 시도를 하였다.

키오스크 서버의 스레드 덤프는 40개의 스레드 모두가 호출한 클래스와 메서드의 정확한 이름을 알려주었다. 바로 `FlightSearch.lookupByCity( )`였다. 나는 스택 트레이스

<sup>16</sup> 애플리케이션에 행이 걸릴 때

스의 몇 프레임 위에서 RMI와 EJB 메서드의 참조를 확인하고 놀랐다. CF는 항상 ‘웹 서비스(web service)’로 언급되었다. 확실히, 웹 서비스의 정의는 당시에 상당히 느슨하긴 했지만 상태 없는 세션 빈(stateless session bean)을 ‘웹 서비스’라고 부르는 것은 적절하지 못한 것 같다.

원격 메서드 호출(Remote Method Invocation, RMI)은 원격 프로시저 호출과 함께 EJB를 제공한다. EJB 호출은 (disco처럼 사장된) CORBA<sup>17</sup>나 RMI의 두 가지 전달방법 중 하나로 전달된다. 내가 RMI 프로그래밍 모델을 좋아하긴 하지만, 호출에서 제한시간(timeout)을 설정할 수 없기 때문에 RMI는 정말 위험하다. 결과적으로, 호출자는 원격서버에서 일어나는 문제에 취약하다.

### 스레드 덤프 얻기

유닉스에서 자바 애플리케이션에 시그널 3(SIGQUIT)을 보내거나 윈도 시스템에서 Ctrl+Break 키를 누르면, 자바 애플리케이션은 JVM에 있는 모든 스레드의 상태를 덤프한다.

윈도에서 이 기능을 쓰려면, 여러분은 자바 애플리케이션을 실행하는 커맨드 창을 띄워둔 콘솔에 있어야 한다. 이런 이유 때문에, 원격으로 접속하는 경우, VNC나 원격 데스크탑을 사용해야 한다.

유닉스에서, 이 신호를 보내기 위해 다음처럼 kill을 사용할 수 있다.

```
kill -3 18835
```

이런 스레드 덤프를 얻는 한 가지 수단은 다음과 같다. 스레드 덤프는 항상 ‘표준 출력(standard out)’으로 나온다. 그러나 미리 만들어진 대부분의 시작 스크립트는 표준 출력을 캡처하지 않고 `/dev/null`로 보내버린다(예를 들어, 젠투Gentoo 리눅스의 JBoss 패키지는 기본값으로 `JBOSS_CONSOLE`을 `/dev/null`로 설정한다). `Log4J`나 `java.util.logging`으로 만들어지는 로그파일은 스레드 덤프에 나타나지 않는다. 스레드 덤프를 얻으려면 애플리케이션 서버의 시작 스크립트로 실험해 봐야 할 수도 있다.

<sup>17</sup> DISCO(Discovery of Web Services)는 마이크로소프트웨어에서 개발한 웹 서비스관련 기술이다. DISCO는 UDDI로 대체되었다.

## 스레드 덤프 얻기 &lt;계속&gt;

다음은 JBoss 3.2.5에서 얻은 스레드 덤프의 일부다.

```
"http-0.0.0.0-8080-Processor25" daemon prio=1 tid=0x08a593f0 \
  nid=0x57ac runnable [a88f1000..a88f1ccc]
  at java.net.PlainSocketImpl.socketAccept(Native Method)
  at java.net.PlainSocketImpl.accept(PlainSocketImpl.java:353)
  - locked <0xac5d3640> (a java.net.PlainSocketImpl)
  at java.net.ServerSocket.implAccept(ServerSocket.java:448)
  at java.net.ServerSocket.accept(ServerSocket.java:419)
  at org.apache.tomcat.util.net.DefaultServerSocketFactory.\
acceptSocket(DefaultServerSocketFactory.java:60)
  at org.apache.tomcat.util.net.PoolTcpEndpoint.\
acceptSocket(PoolTcpEndpoint.java:368)
  at org.apache.tomcat.util.net.TcpWorkerThread.\
runIt(PoolTcpEndpoint.java:549)
  at org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.\
run(ThreadPool.java:683)
  at java.lang.Thread.run(Thread.java:534)
"http-0.0.0.0-8080-Processor24" daemon prio=1 tid=0x08a57c30 \
  nid=0x57ab in Object.wait() [a8972000..a8972ccc]
  at java.lang.Object.wait(Native Method)
  - waiting on <0xacede700> (a org.apache.tomcat.util.threads.\
ThreadPool$ControlRunnable)
  at java.lang.Object.wait(Object.java:429)
  at org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.\
run(ThreadPool.java:655)
  - locked <0xacede700> (a org.apache.tomcat.util.threads.\
ThreadPool$ControlRunnable)
  at java.lang.Thread.run(Thread.java:534)
```

결과를 상세하게 출력한다.

스레드 덤프 일부는 두 개의 스레드를 보여주는데, 각 스레드는 http-0.0.0.0-8080-ProcessorN 과 같은 이름이 있다. 25번은 실행 가능한 상태인 반면, 24 스레드는 Object.wait() 안에서 블록되었다. 이 트레이스는 이것들이 스레드 풀의 구성원임을 명백하게 알려준다. 스택에 있는 클래스 가운데 일부가 ThreadPool\$ControlRunnable() 라는 것도 이것을 뒷받침한다.

## 2.4 확실한 증거

이 시점에, 사후검토 분석은 정지 사태의 증상과 일치했다. 즉, CF가 IVR과 키오스크 체인에 행(hang)을 일으킨 것처럼 보였다. “CF에서 무슨 일이 생겼는가?”라는 가장 큰 질문이 여전히 남아 있었다.

CF에서 얻은 스레드 덤프를 조사하면서 이 그림은 더욱 선명해졌다. CF의 애플리케이션 서버는 EJB 호출과 HTTP 요청을 다루기 위해 별도의 스레드 풀을 사용하였다. 이런 이유로 정지 사태 중에도, CF는 모니터링 애플리케이션에 항상 응답할 수 있었다. HTTP 스레드는 완전히 유희상태(idle)였는데, 이것은 EJB 서버 입장에서 이치에 맞았다. 한편 EJB 스레드는 오로지 FlightSearch.lookupByCity() 호출을 처리하는 데 사용되었다. 사실 모든 애플리케이션 서버에 있는 모든 스레드는 코드의 정확하게 같은 라인, 즉 리소스 풀(resource pool)에서 데이터베이스 연결을 체크아웃하려는 부분에서 블록되었다.

이것은 정황증거였지, 확실한 증거는 아니었다. 그러나 정지 사태 전의 데이터베이스 페일오버를 고려한다면, 방향은 제대로 잡은 듯했다.

다음 부분이 아슬아슬했다. 나는 소스코드를 조사해야 했지만, 운영 센터는 소스 관리 시스템에 접근권한이 없었다. 바이너리(binary) 파일만 실전환경에 배치되었다. 이렇게 하는 것은 일반적으로 좋은 보안대책이지만, 그 순간에는 다소 불편했다. 고객담당 책임자에게 소스코드에 어떻게 접근할 수 있는지 물어봤을 때, 고객담당 책임자는 그 단계를 진행하길 꺼려했다. 정지 사태의 규모로 볼 때, 누군가 책임지게 하려는 분위기가 만연했음을 짐작할 것이다. 운영센터와 개발조직 사이의 관계가 항상 편한 건 아니었지만, 평소보다 더욱 긴장감이 돌았다. 모든 사람들이 방어적이었으며, 비난의 화살을 자신 쪽으로 돌리려는 어떤 시도도 경계했다.

소스코드에 합법적인 접근권한이 없었기 때문에, 내가 할 수 있는 유일한 일, 즉 실전 서버에서 바이너리 파일을 가져다가 이 파일을 디컴파일(decompile)했다.<sup>18</sup> 의심스러운 EJB에 해당하는 코드를 본 순간, 진짜 확실한 증거를 발견한 것을 알았다. 이 특정 세션 빈은 CF가 현재 구현한 유일한 기능임이 드러났다. 실제 코드는 아래에 실려있다.

처음 보기에, 이 메서드는 잘 만들어진 듯 보인다. try.. finally 블록의 사용은 작성자가 리소스를 해지하려는 의도를 보여준다. 사실 이 리소스 해지블록은 서점에 있는 자바 서적 몇 권에도 실려있다. 매우 유감스럽게도 이 소스코드에는 치명적인 결함이 있다.

`java.sql.Statement.close()`도 `SQLException`을 던질 수 있다. 그러나 거의 발생하지 않는다. 오라클 드라이버는 연결을 종료하려고 시도할 때 `IOException`을 만나는 경우 이 예외를 던진다(예를 들어 데이터베이스 파일오버 다음).

파일오버 전에 JDBC 연결이 생성되었다고 가정해보자. 연결을 생성하기 위해 사용된 IP 주소는 호스트에서 다른 호스트로 변경되지만, TCP 연결의 현재 상태는 보조 데이터베이스 호스트로 넘어가지 않는다. 이 때 소켓에 무언가를 쓰면 `IOException`을 던진다(운영시스템과 네트워크 드라이버가 최종적으로 TCP 연결이 끊어졌다고 결정한 다음에 그렇다). 이 얘기는 리소스 풀에 있는 모든 JDBC 연결 시 사고가 발생한다는 것을 뜻한다.

```
package com.example.cf.flightsearch;

...

public class FlightSearch implements SessionBean {

    private MonitoredDataSource connectionPool;
```

<sup>18</sup> 내가 제일 좋아하는 자바코드 디컴파일 도구는 여전히 JAD다. 비록 자바5 코드에서 사용할 때 문제가 있지만, JAD는 빠르고 정확하다.

```
public List lookupByCity(. . .) throws SQLException, RemoteException {
    Connection conn = null;
    Statement stmt = null;

    try {
        conn = connectionPool.getConnection();
        stmt = conn.createStatement();

        // Do the lookup logic
        // return a list of results
    } finally {
        if (stmt != null) {
            stmt.close();
        }

        if (conn != null) {
            conn.close();
        }
    }
}
```

놀랍게도, JDBC 연결 객체는 여전히 Statement 객체를 생성하기 위해 대기한다. Statement 객체를 생성하기 위해 드라이버의 Connection 객체는 드라이버의 내부 상태를 검사한다.<sup>19</sup> JDBC 연결이 아직 연결되어 있다고 간주하면, JDBC 연결 객체는 Statement 객체를 만들 것이다. 이 Statement 객체를 실행하면 Statement 객체가 네트워크 I/O 작업을 할 때 `SQLException`을 던질 것이다. 그러나 Statement 객체를 닫으려고 할 때도 `SQLException`을 던지는데, 드라이버가 이 Statement 객체와 관련된 리소스를 해지하라고 데이터베이스 서버에게 알려주기 때문이다.

간단히 말해, 드라이버는 사용할 수 없는 Statement 객체를 생성하려고 했다. 여러분은 이것을 버그라고 생각할지 모른다. 항공사에 있는 많은 개발자들은 이러한 비난거리를 만들었다. 여기에서 얻는 핵심 교훈은 JDBC 명세가 `java.sql.Statement.close()`

<sup>19</sup> 이것은 오라클 JDBC 드라이버에 있는 이상한 특징인지도 모르겠다. 나는 오라클 JDBC 드라이버만 디컴파일 해봤기 때문이다.

)가 `SQLException`을 던지도록 허용했다는 것이다. 따라서 여러분의 코드는 이 예외를 처리해야 한다.

앞에서 본 문제가 약간 존재하는 코드에서, `statement`를 닫으려는 코드가 예외를 던지고 나서 연결이 닫히지 않았기에 리소스 누수가 생겼다. 이러한 호출이 40번 일어난 후, 리소스 풀은 바닥이 났으며 모든 호출은 `connectionPool.getConnection()`에서 블록되었다. 이것은 정확히 내가 CF에서 얻은 스레드 덤프에서 보았던 것이다.

항공기 수백 대와 직원 수만 명을 거느린 전세계적인 수십억 달러 가치의 항공사는 프로그래머의 초보적인 실수, 즉 처리되지 않은 `SQLException` 때문에 정지되었다.

## 2.5 약간의 예방

어마어마한 비용이 이처럼 작은 에러에서 발생할 때, 자연스런 반응은 다음처럼 말하는 것이다. “다시는 일어나서는 안돼.” 그러나 어떻게 막을 수 있을까? 코드리뷰가 이 버그를 잡아낼 수 있었을까? 검토자 가운데 한 사람이 오라클 JDBC 드라이버의 내부를 알았거나 검토 팀이 메서드마다 몇 시간을 보내기만 했다면 그랬을지도 모른다. 더 많은 테스트가 이 버그를 예방했을까? 아마 그럴지도 모른다. 문제가 일단 파악되고, 같은 에러를 보여주는 스트레스 테스트 환경에서 팀이 테스트했다면 예방했을 것이다. 평범한 테스트 프로파일은 이 버그를 보여줄 정도로 이 메서드를 테스트하지 못했다. 다른 말로 하자면, 어디를 살펴봐야 하는지 알기만 한다면 버그를 찾는 테스트를 만들기란 쉽다.

궁극적으로, 이처럼 모든 버그를 몰아내길 기대하는 것은 단지 환상이다. 버그란 생기기 마련이다. 버그들은 제거할 수 없기 때문에, 버그들은 반드시 살아남는다.

여기에 든 사례에서 최악의 문제는 하나의 시스템에 있는 버그가 영향을 미치는 다른 모든 시스템으로 전달되었다는 사실이다. 의문을 던질 만한 더 좋은 질문은 “한 시스템에 있는 버그들이 다른 시스템에 영향을 끼치는 것을 어떻게 막을까?”이다. 오늘날의

모든 회사는 서로 연결되고, 서로 관련된 시스템들의 집합이다. 이러한 시스템들은 버그로 인해 연쇄적으로 고장이 발생하도록 허용할 수도 없으며, 허용해서도 안 된다. 따라서 이러한 종류의 문제가 퍼지는 상황을 막을 수 있는 디자인 패턴을 살펴볼 것이다.

## 3장



## 안정성 소개

새로운 소프트웨어는 낙관적인 열정으로 가득 찬 대학 졸업생처럼 세상에 등장했다가 갑자기 실험실 밖 세상의 험한 현실을 마주하게 된다. 실험실에서는 보통 생기지 않는 일들이 실제 세상에서 생기게 마련인데, 대개는 나쁜 일들이다. 실험실에서 이뤄지는 모든 실험은 어떤 결과가 나올지 아는 사람들에게 의해 시도된다. 그러나 실제 세상에서, 실험은 답을 얻도록 설계된 것이 아니다. 때로 실험은 소프트웨어가 실패하는 것을 보기 위해 만들어 놓았을 뿐이다.

엔터프라이즈 소프트웨어는 냉소적이어야 한다. 냉소적인 소프트웨어는 나쁜 일이 생길 거라 예상하고 그런 일이 생겼을 때 전혀 놀라지 않는다. 냉소적인 소프트웨어는 자신조차 믿지 않기 때문에, 고장으로부터 자신을 보호하려고 내부 장벽을 세운다. 냉소적인 소프트웨어는 다른 시스템과 지나치게 친밀하길 거부하는데, 다치게 될까 염려하기 때문이다.

이전 장에서 논의한 항공사 CF 프로젝트는 충분히 냉소적이지 못했다. 매우 자주 일어나는 일이지만, 그 프로젝트를 구현한 팀은 신기술과 진보된 아키텍처에 대한 흥분에 사로잡혀 있었다. 이 사건은 지렛대 효과와 상승효과(synergy)에 대해 많은 애깃거리를 제공한다. 돈의 유혹에 현혹되어 멈춤 신호를 보지 못했고 악화일로로 걸었다.

낮은 안정성은 실질적으로 심각한 비용을 수반한다. 이 가운데 명백한 비용은 수입의 손실이다. 1쪽 1장 ‘소개’에서 논의한 소매상은 가동이 한 시간 중단될 때마다 10만 달러를 잃는데, 그것도 제철이 아닌 때를 기준으로 한 것이다. 거래 시스템(trading system)이라면 단 한 번의 누락된 트랜잭션으로 그 만한 비용을 잃을 수도 있다.

일반적인 경험으로는 온라인 소매상이 고객을 확보하는데 한 명당 25달러에서 50달러가 든다. 시간 당 5,000명의 고유 방문자가 있고 이 예비 고객의 10퍼센트가 영원히 떠난다고 가정해 보자. 이것은 1만 2천 500달러에서 2만 5천 달러가 고객 확보 비용으로 낭비됨을 의미한다.<sup>1</sup>

실감이 덜 나지만, 이와 동일한 정도로 고통스러운 일은 평판을 잃는 것이다. 바로 드러나는 면에서 브랜드에 손상이 가는 것은 고객을 잃는 것보다 덜하지만, 비즈니스 위크(Business-Week)지에 연휴 운영 중 발생한 문제가 보도된다고 생각해 보라. 기업 이미지 광고에 든 수백만 달러가(온라인 고객 서비스를 열렬히 선전하면서) 불량 하드드라이브 때문에 몇 시간 안에 무용지물이 될 것이다.

**매우 안정적인 설계라도 구현할 때는 대개 불안정한 설계를 구현할 때와 동일한 비용이 들어간다.**

안정성이 좋다고 꼭 비용이 많이 드는 것은 아니다. 아키텍처와 설계, 심지어 저수준의 시스템을 구현할 때 시스템의 궁극적 안정성에 큰 영향을 미치는 결정 요소들이 많다. 이런 지렛대 효과에 영향을 주는 요소들과 마주칠 때, 안정적 설계와 불안정한 설계 모두는 (QA를

목표로 한) 기능적 요구를 만족시킬 수 있다. 하지만 불안정한 설계는 매년 수 시간의

작동 중단 시간을 초래할 것이지만 안정적인 설계는 그렇지 않다. 놀라운 일은 매우 안정적인 설계나 불안정한 설계나 구현할 때 드는 비용이 같다는 것이다.

### 3.1 안정성이란?

안정성에 대해 말하려면 몇 가지 용어를 정의할 필요가 있다. 트랜잭션(transaction)이란 시스템이 처리하는 추상적인 작업단위다. 이것은 데이터베이스 트랜잭션과 다르다. 하나의 작업 단위는 많은 데이터베이스 트랜잭션을 포함할 수 있다. 예를 들면, 온라인거래(ecommerce) 사이트에서 트랜잭션의 일반적인 형태는 ‘고객이 제품을 주문하는 것’이다. 이 트랜잭션은 여러 페이지에 걸쳐 이뤄지고, 신용카드 인증과 같은 외부 시스템과의 통합도 흔히 포함한다. 트랜잭션은 시스템이 존재하는 이유다. 시스템이 단지 한 형태의 트랜잭션만을 처리하면 전용시스템(dedicated system)이 된다. 복합 작업(mixed workload)은 하나의 시스템이 처리하는 다양한 트랜잭션 형태의 조합이다.

내가 시스템이라는 용어를 쓸 때, 사용자를 위해 트랜잭션을 처리하는데 필요한 하드웨어, 애플리케이션, 서비스 등의 완전하고, 상호의존적인 집합을 의미한다. 하나의 시스템은 단일 애플리케이션처럼 작을 수도 있고, 애플리케이션과 서버가 전개된 다계층(multitier) 네트워크일 수도 있다.

나는 중단에서 다른 중단까지 트랜잭션을 처리하는 호스트, 애플리케이션, 네트워크 부분, 전원공급 등의 집합을 뜻할 때 시스템이란 용어를 사용한다.

복원 시스템(resilient system)은 일시적인 충격이나 지속적인 스트레스, 또는 정상 처리를 방해하는 부품 고장이 있을 때라도 트랜잭션을 계속 처리한다. 이것이 바로 사람들이 안정성이란 말을 할 때 의미하는 바다. 그러므로 여러분의 개인 서버나 애플리케이션이 단순히 살아서 실행될 뿐 아니라, 사용자가 작업을 무사히 마칠 수 있어야 한다.

충격(impulse)과 스트레스라는 용어는 기계공학에서 나왔다. 충격이란 시스템에 가해진 급격한 힘이다. 시스템에 가해지는 충격은 그 시스템을 해머와 같은 것으로 세계 내리

<sup>1</sup> <http://retailindustry.about.com/library/weekly/aa122599a.htm> 참조.

칠 때 생긴다. 대조적으로 시스템에 가해지는 스트레스는 일정시간 동안 시스템에 가해진 힘이다.

할인판매 소문 때문에 Xbox 360 제품사이트의 제품설명 페이지에 접속이 몰리는 것은 충격을 일으킨다. 새로운 세션 만 개가 일본 안에 도달하면 매우 견디기 힘들다. 슬래시닷을 경험하는 것도 충격이다. 또 11월 21일 밤 중에 메시지 1,200만 개를 큐로 보내는 것도 충격이다.<sup>2</sup> 이런 것들이 시스템을 눈 깜짝할 사이에 파괴할 수 있다.

반면, 신용카드 처리기에서 겪는 느린 반응은 모든 고객을 감당할 용량이 안 되기 때문인데, 이것이 바로 시스템에 가해지는 스트레스다. 기계 시스템에서 스트레스가 가해지면 물체는 형상이 바뀐다. 이 형상의 변화를 스트레인이라고 한다. 스트레스는 스트레인을 일으킨다. 컴퓨터 시스템에서도 비슷한 일이 생긴다. 신용카드 처리기에서 생긴 스트레스는 시스템의 다른 부분으로 스트레인을 전파하는데, 이것은 이상한 영향을 끼칠 수 있다. 웹 서버의 메모리 사용량이 증가하든지, 데이터베이스 서버의 입출력률(I/O rate)이 과도하게 되든지, 아니면 다른 멀리 떨어진 대상에 영향을 미친다.

**장기 테스트(longevity tests)를  
실행하라. 이것이 장기 버그  
(longevity bugs)를 잡는 유일한  
길이다.**

지속성을 갖는 시스템은 트랜잭션을 오랫동안 계속해서 처리한다. ‘오랫동안’이란 게 무엇일까? 상황에 따라 다르겠지만 코드 배치 사이의 시간을 뜻하는 것이 유용하겠다. 새로운 코드가 매주 실전에 배치된다면, 시스템이 리부팅 없이 2년 동안 실행될 수 있는지 여부는 중요

하지 않다. 반면, 서부 몬타나(Montana) 주의 자료 수집 시스템(data collector)은 일주일에 한 번씩 시스템을 직접 리부팅해야 할 필요가 정말 없어야 한다(여러분이 서부 몬타나 주에 살고 싶지 않다면 말이다).<sup>3</sup>

2 11월 21일은 세계 인사의 날로서 문안인사를 평화를 기원하기 위해 10사람에게 문안인사를 한다. 그러나 문맥 상 Xbox 360의 미국 발매일인 2005년 11월 22일 전날 밤에 소비자들이 사이트에 엄청나게 접속한 상황을 말하는 것으로 보인다.

3 몬타나 주는 스페인어 ‘산’의 뜻에서 유래했다. 몬타나 주의 서부에는 주 면적의 40퍼센트를 차지하는 록시산맥이 뻗어 있고 동부의 나머지 60퍼센트가 고지평원이다. 기후는 동서 간에 차이가 있으나 서늘하고 건조한 대륙성 기후를 보이며, 겨울에는 무척이나 춥다.

## 생명 연장하기

시스템의 수명을 위협하는 주요 위험은 메모리 누수(memory leak)와 데이터 성장(data growth)이다. 두 종류의 찌꺼기는 실전 환경에서 시스템을 죽일 것이다. 둘 다 테스트 중에는 잡아내기 어렵다.

테스트는 문제를 눈에 띄게 하여 고칠 수 있게 한다(이 때문에 나는 테스터들이 버그를 발견할 때 항상 고마워한다). 머피의 법칙대로, 테스트해보지 않은 일은 무엇이든 생길 수 있다. 그러므로 자정이 지난 직후에 일어나는 충돌이나 49시간째 가동된 후 애플리케이션에서 발생하는 메모리 부족 에러 같은 것을 테스트하지 않았다면, 이런 일들은 발생하게 된다. 일주일 지나야 나타나는 메모리 누수를 테스트하지 않았다면, 일주일 후에 메모리 누수가 일어난다.

문제는 애플리케이션의 장기 버그를 드러내기에 충분할 만큼 개발 환경에서 애플리케이션을 오래 실행하지 않는다는 것이다. 개발 환경에서 보통 얼마나 오랫동안 애플리케이션 서버를 계속 실행하는가? 장담컨대 티보(Tivo)\*1에 저장된 시트콤의 길이보다 짧을 것이다.\*2 QA기간 동안에는 조금 더 실행할지 모르지만, 최소한 하루에 한 번은(더 자주는 아니라도) 다시 실행할 것이다. 게다가 시스템이 실행되어 운영 중이더라도 부하가 계속 걸리지는 않는다. 이런 환경은 서버를 매일 트래픽 상황에서 한달 동안 실행하는 것과 같이 오랫동안 테스트하는 것과는 맞지 않다.

그럼, 이런 버그를 어떻게 찾을까? 실전 환경에서 이런 버그가 발견되기 전에 잡는 유일한 방법은 여러분이 직접 장기 테스트를 실행하는 것이다. 가능하면, 개발 머신을 마련해서 제이미터(JMeter)나 마라톤(Marathon), 아니면 다른 부하 테스트 도구를 실행하라. 시스템을 심하게 몰아세우지는 말고, 요청을 항상 처리하게 하라(또한, 하루에 몇 시간씩 스크립트가 느려지게 하여 한밤중의 느린 주기를 시뮬레이션하게 하라. 이러면, 연결 풀이나 방화벽 시간제한 등의 문제를 잡을 수 있다).

때로는 돈 때문에 완벽한 환경을 구축할 수 없을 수도 있다. 그런 상황이라면, 나머지 부분을 무시하더라도 적어도 중요한 부분은 테스트하라. 전혀 안 하는 것보다는 낫다.

아무런 테스트도 하지 않는다면, 실전 환경은 자동으로 여러분의 장기 테스트 환경이 된다. 분명히 실전에서 버그를 발견하겠지만, 행복한 생활방식을 위한 처방은 아닌 셈이다.

\*1 TV 프로그램 140시간 분을 녹화할 수 있는 대용량 하드디스크를 탑재한 기기다.

\*2 광고하는 시간과 출연자들 이름이 나오는 시작과 끝 부분의 시간을 제외하면 약 21분이다.

### 3.2 고장 유형

갑작스런 충격과 과도한 스트레인 모두는 최악의 고장을 일으킬 수 있다. 어느 경우든지, 시스템의 어떤 컴포넌트(component)는 다른 부분보다 먼저 고장나기 시작할 것이다. 'Inviting Disaster[Chi01]'에서 제임스 R. 차일즈(James R. Chiles)는 이것을 시스템의 크랙이라고 불렀다. 차일즈는 금속에 미세한 크랙이 생긴 강판(steel plate)과 고장 직전의 복잡한 시스템 사이에서 유사점을 발견했다. 스트레스를 받으면 크랙은 전파되기 시작하며, 점점 빨라진다. 결국 크랙은 소리의 속도보다도 빨리 전파되고, 금속은 폭발음과 함께 파괴된다. 최초로 크랙을 일으키는 것과 시스템의 나머지 부분으로 크랙이 전파되는 방식, 그리고 손상된 결과를 합쳐 고장 유형(failure mode)이라고 부른다.

무엇이 되었든, 시스템에는 다양한 고장 유형이 존재한다. 고장이 불가피하다는 것을 부인하면 그 고장을 통제하고 수용할 능력을 잃어버리게 된다. 고장이 일어난다는 것을 받아들인다면, 특정 고장에 대해 시스템의 반응을 설계할 능력이 생긴다. 자동차 엔지니어가 완충구간(crumple zone, 사고 발생 시 승객을 보호하기 위해 설계된 영역)을 만들듯이, 여러분은 손상을 수용하고 시스템의 나머지를 보호하는 고장 유형을 만들 수 있다. 이런 종류의 자기 보호(self-protection)는 전체 시스템의 복원력을 결정한다.

차일즈는 이런 보호를 크랙차단기(crackstoppers)라고 부른다. 충격을 흡수하고 승객을 안전하게 보호하기 위해 차에 완충구간을 만들듯이, 시스템의 어떤 기능이 필수적인지 결정하여 크랙이 그 기능에 영향을 미치지 못하도록 고장 유형을 만들 수 있다. 고장 유형을 설계하지 않으면, 예측하지 못한(그리고 대개 위험한) 고장은 무엇이든 일어난다.

### 3.3 크랙 전파

내가 전에 조사했던 운항이 정지된 항공사에 이런 크랙차단기가 어떻게 적용되는지 보자. 항공사의 CF 프로젝트는 고장 유형을 설계하지 않았다. 크랙은 SQLException을 부적절하게 처리하는 데서 발생했지만, 다른 많은 지점에서 크랙을 멈출 수 있었다. 낮

은 수준의 상세사항부터 상위 수준의 아키텍처까지 몇 가지 예를 살펴보자.

풀은 자원(resources)을 할당할 수 없을 때 요청하는 스레드를 블록하도록 설정되어 있었기 때문에, 결국 요청을 처리하는 모든 스레드가 중지되었다(이 현상은 애플리케이션 서버의 인스턴스마다 독립적으로 일어났다). 자원이 고갈되었더라도 더 많은 연결을 생성하도록 풀을 구성할 수도 있었다. 또한 모든 연결이 체크아웃(check out)되었을 때 요청자를 영원히 블록하는 대신 제한된 시간 동안 호출자를 블록하도록 설정할 수도 있었다. 둘 중 하나라도 해 놓았다면 크랙이 전파되는 것을 막았을 것이다.

한 수준을 올려 살펴보면, CF에서 하나의 호출과 관련된 문제가 다른 호스트에서 호출하는 애플리케이션을 실패하게 만들었다. CF는 자신의 서비스를 EJB로 노출하기 때문에 RMI를 사용했다. 기본적으로 RMI 호출은 시간제한에 걸리지 않는다. 다시 말해, 호출자는 CF의 EJB로부터 응답을 얻으려고 기다리면서 블록되었다. 각각의 인스턴스를 호출한 처음 스무 개의 애플리케이션은 예외(정확히 말해 RemoteException로 감싸진 InvocationTargetException, 그리고 이 예외로 감싸진 SQLException)를 받았다. 그 후에는 호출이 블록되기 시작했다.

클라이언트에는 RMI 소켓에 타임아웃을 설정할 수도 있었다.<sup>4</sup> 또 어떤 시점에서는 CF를 EJB 대신 HTTP를 사용하는 웹서비스로 만들도록 결정할 수도 있었다. 그렇게 하면, 클라이언트는 HTTP 요청에 제한시간을 설정할 수 있다.<sup>5</sup> 클라이언트는 요청을 처리하는 스레드가 외부에 통합 요청을 하는 대신 블록된 스레드는 버려지도록 호출을 작성할 수도 있었다. 이런 방법 중 아무것도 이뤄지지 않았고 CF로부터 이 CF를 사용한 모든 시스템으로 크랙이 전파되었다.

좀 더 큰 관점에서 봤을 때, CF 서버 자체를 하나 이상의 서비스 그룹으로 나눌 수

<sup>4</sup> 예를 들어 클라이언트가 새로운 소켓을 만들 때마다 Socket.setTimeout()를 호출하는 소켓 팩토리(socket factory)를 설치해서 제한시간을 설정한다.

<sup>5</sup> java.net.URL이나 java.net.URLConnection을 사용하지 않았다면 말이다. JAVA 5까지는 표준 자바 라이브러리로 만들어진 HTTP 호출에 제한시간을 설정하는 것이 불가능했다

있었다. 이 경우, CF의 모든 사용자가 정지하는 대신 서비스 그룹 중 하나만 정지되는 것으로 문제가 한정될 수 있었을 것이다(이런 경우에, 모든 서비스 그룹이 동일한 방식으로 크랙을 경험했을 수도 있지만 항상 그렇게 모든 서비스 그룹에 크랙이 생기는 것은 아니다). 이것은 엔터프라이즈 시스템의 나머지로 크랙이 전파되는 것을 막는 다른 방법이다.

더 큰 아키텍처 이슈를 보면, 요청/응답 메시지 큐(message queues)를 사용하여 CF를 구축할 수도 있었다. 이런 경우라면, 호출자는 응답이 아예 도착하지 못할 수 있음을 안다. 호출자는 프로토콜을 처리하는 기능의 일부로서 이런 경우를 처리해야 한다. 더 근본적으로, 호출자는 검색 조건에 맞는 터플스페이스(tuple space)<sup>6</sup>에서 항목을 찾음으로써 항공편을 찾을 수도 있었다. CF는 항공편 기록을 터플스페이스에 채울 수도 있었다. 아키텍처가 더 밀접히 결합(coupling)될수록, 이런 코딩 에러가 전파될 가능성은 더 커진다. 반대로, 결합이 덜 된 아키텍처가 충격을 흡수해주면, 이런 에러의 영향은 증폭되지 않고 줄어든다.

이런 접근 방식 중 무엇이든 SQLException 문제가 항공사의 다른 부분으로 전파되는 것을 막을 수 있었다. 유감스럽게도, 설계자는 공유 서비스를 만들 때 ‘크랙’의 가능성을 고려하지 않았다.

### 3.4 고장의 연쇄(chain of failure)

모든 시스템 정지의 이면에는 앞에서 살펴본 바와 같은 연속적인 사건이 있다. 사소한 일이 다른 일을 일으키고, 또 일어난 일은 다른 일을 일으킨다. 이러한 사실을 안 이후에 전체 고장의 연쇄 과정을 살펴보면 고장은 불가피해 보인다. 그런 사건의 연쇄가 일어날 가능성을 예측한다면, 아무리 해도 일어나지 않을 것 같아 보인다. 하지만, 각

사건의 확률을 독립적으로 고려할 때만 일어나지 않을 것 같아 보이는 것이다. 동전 던지기의 경우 동전은 기억력이 없기 때문에, 이전의 결과에 상관없이 동전을 던질 때마다 확률은 동일하다. 그러나 고장을 일으키는 사건의 조합은 독립적이지 않다. 한 지점이나 계층(layer)에서 발생한 고장은 다른 고장의 가능성을 실제로 증가시킨다. 데이터베이스가 느려지면 애플리케이션 서버는 메모리가 바닥날 가능성이 커진다. 계층은 결합되었기 때문에, 각 사건들은 독립적이지 않다.

연쇄적인 고장의 각 단계마다, 크랙은 빨라질 수도, 느려질 수도, 혹은 멈출 수도 있다. 복잡할수록 더 다양한 방향으로 크랙이 진행된다.

결합이 긴밀할수록 크랙을 가속한다. 예를 들어 EJB 호출이 밀접하게 결합되어 있으면 CF에서 발생한 자원 고갈 문제는 호출자에게 더 큰 문제를 일으킨다. 이러한 시스템에서 외부 통합 호출에 요청을 처리하는 스레드를 결합하면 원격지 문제가 시스템 정지 문제로 바뀐다.

가능한 모든 고장에 대비하는 한 가지 방법은 모든 외부 호출과, 모든 입출력, 모든 자원 사용, 그리고 모든 예상 결과를 살펴보고, “이런 것들이 잘못될 수 있는 방법에는 어떤 것들이 있을까?” 라고 질문해 보는 것이다. 가해질 수 있는 충격과 스트레스의 다양한 종류에 대해 생각해 보자.

- 초기 접속을 못 하면 어떻게 되는가?
- 접속하는 데 10분이 걸리면 어떻게 되는가?
- 접속된 후 끊어지면 어떻게 되는가?
- 접속된 후 상대방 반대편에서 어떤 응답도 못 받으면 어떻게 되는가?
- 내 쿼리에 응답하는데 2분이 걸리면 어떻게 되는가?
- 동시에 만 개의 요청이 오면 어떻게 되는가?
- 네트워크가 임에 의해서 멈추었기 때문에 발생한 SQLException에 대한 에러 메시지를 남기려 할 때 디스크가 꽉 차면 어떻게 되는가?

<sup>6</sup> 병렬 또는 분산컴퓨팅 시스템에서 공유 메모리 패러다임을 구현한 것을 말한다. 터플은 특정한 형의 객체가 연속된 것을 말하는데, 터플 스페이스는 동시에 접근할 수 있는 터플의 저장소를 제공한다. 칠판 은유(Blackboard Metaphor)라고도 불린다.

나열하려니 벌써 피곤해지는데, 잘못될 수 있는 모든 것을 나열하는 일을 이제 시작했을 뿐이다. 그러므로 이와 같은 무지마지한(brute-force) 접근방식은 화성 탐사기 같이 꼭 살아 있어야 하는 시스템을 제외하면 비현실적이다. 10년 안에 실제로 시스템을 인도해야 한다면 어떻게 해야 할까? 이런 스트레스를 경감시키는 충격 흡수기를 만들기 위해 몇 가지 패턴을 살펴봐야 한다.

### 3.5 패턴과 안티패턴

나는 수백 가지의 실전 상황의 고장을 다뤄 왔는데, 하나하나가 독특했다(어쨌든 대부분 독특했는데, 그것은 내가 동일한 고장을 두 번 겪지 않도록 노력했기 때문이다!). 내 기억에 두 가지 사고에서 정확히 고장의 연쇄가 동일한 방식으로 일어났던 적은 없었다. 즉, 같은 트리거(trigger), 같은 파손, 같은 전파와 같은 방식 말이다. 그렇지만, 시간이 가면서 고장의 패턴이 드러난다. 어떤 축(axis)을 따라서 나타나는 취약성, '이런' 문제가 '저런' 방식을 증폭시키는 경향 등이 바로 그런 패턴이다. 이런 것들이 안정성 안티패턴이다. 43쪽 4장 '안정성 안티패턴'에서 이러한 고장의 패턴에 대해 다룬다.

고장에 체계적인 패턴이 있다면 적용 가능한 공통 해법이 있으리라 짐작할 것이다. 맞다. 125쪽 5장 '안정성 패턴'에서 안티패턴을 물리칠 설계와 아키텍처 패턴을 다룬다. 이런 패턴이라도 시스템에서 크랙을 막을 수 없다. 다른 어떤 것도 마찬가지다. 크랙을 일으킬 수 있는 조건들이 항상 있기 마련이다. 이런 패턴은 크랙이 전파되는 것을 막는다. 전체 시스템이 붕괴되는 대신 손상된 채로 부분적인 기능을 유지하는데 도움을 준다.

패턴과 안티패턴이 서로 상호작용한다는 데 놀라서는 안 된다. 안티패턴은 서로를 강화하는 경향이 있다. 영화에서 괴물<sup>7</sup>에 대해 마늘과 은, 그리고 불을 조합하듯이 각각의 패턴은 특정한 문제를 경감시킨다.

7 뱀파이어나 늑대인간이나 프랑켄슈타인에 나오는 괴물을 말한다.

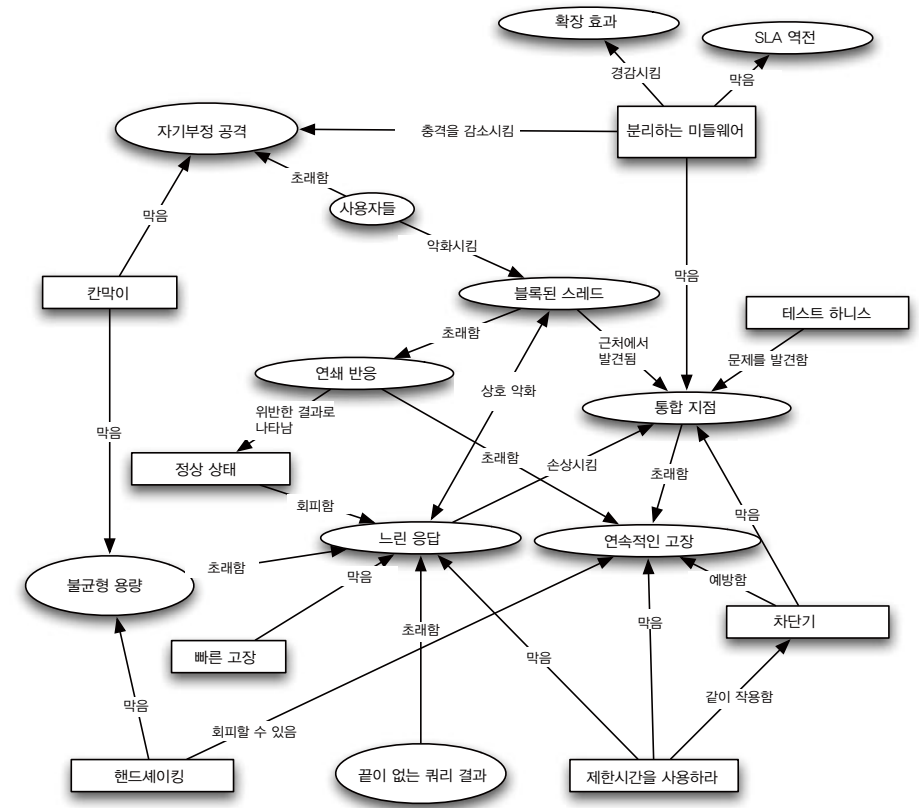


그림 3.1 | 패턴과 안티패턴의 상호 작용

그림 3.1은 이런 상호작용에서 가장 중요한 것들의 연결을 보여준다. 먼저 고장의 공통 원인, 즉 안티패턴을 살펴보는 것부터 시작하자.