제16장

우주 기반 건축 양식

대부분의 웹 기반 비즈니스 애플리케이션은 일반적인 요청 흐름을 따릅니다. 브라우저의 요청이 웹 서버를 거쳐 애플리케이션 서버, 그리고 최종적으로 데이터베이스 서버에 도달하는 방식입니다. 이러한 일반적인 요청 흐름은 동시 접속 사용자 수가 적을 때는 문제없이 작동하지만, 동시 접속 사용자 수가 증가함에 따라 병목 현상이 발생하기 시작합니다. 처음에는 웹 서버 계층에서, 그 다음에는 애플리케이션 서버 계층에서, 마지막으로는 데이터베이스 서버 계층에서 병목 현상이 나타납니다.

사용자 부하 증가로 병목 현상이 발생할 때 일반적인 대응책은 웹 서버를 확장하는 것입니다. 이는 비교적 쉽고 저렴하며, 때로는 효과가 있습니다.
하지만 대부분의 경우 사용자 부하가 높을 때 웹 서버 계층을 확장해도 병목 현상은 애플리케이션 서버로 옮겨갈 뿐입니다. 애플리케이션 서버 확장은 웹 서버 확장보다 더 복잡하고 비용이 많이 들며, 그렇게 하더라도 병목 현상은 결국 확장하기 훨씬 더 어렵고 비용이 많이 드는 데이터베이스 서버로 옮겨가는 경우가 많습니다. 데이터베이스를 확장할 수 있다고 하더라도 결국에는 그림 16-1에서처럼 가장 넓은 부분이 웹 서버(확장하기 가장 쉬움)이고 가장 좁은 부분이 데이터베이스(확장하기 가장 어려움)인 삼각형 모양의 토폴로지가 됩니다 .

동시 접속 사용자 수가 많은 대용량 애플리케이션의 경우, 데이터베이스는 애플리케이션이 동시에 처리할 수 있는 트랜잭션 수를 제한하는 최종적인 요인이 되는 경우가 많습니다. 다양한 캐싱 기술과 데이터베이스 확장 제품이 도움이 될 수 있지만, 일반적인 애플리케이션을 극한의 부하에 맞춰 확장하는 것은 여전히 매우 어려운 과제입니다.

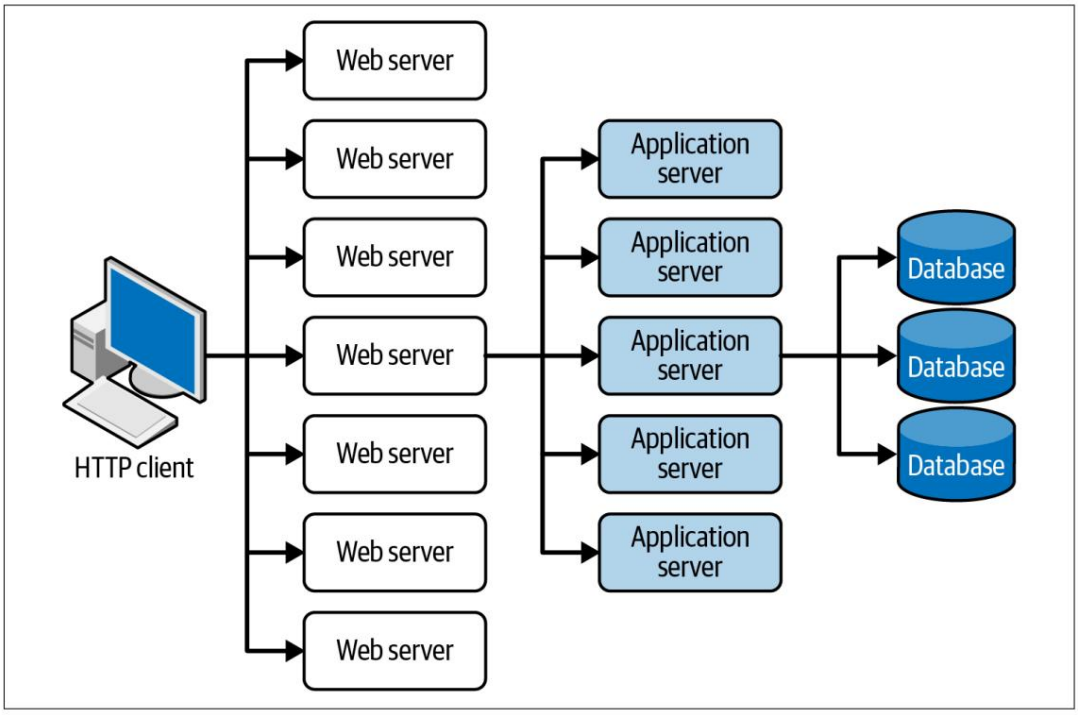

그림 16-1. 기존 웹 기반 토폴로지 내의 확장성 한계

공간 기반 아키텍처 스타일은 높은 확장성, 탄력성 및 동시성을 요구하는 문제를 해결하기 위해 특별히 설계되었습니다. 또한 동시 사용자 수가 가변적이고 예측 불가능한 애플리케이션에도 유용한 아키텍처 스타일입니다. 극단적이고 가변적인 확장성 문제는 데이터베이스를 확장하거나 확장성이 떨어지는 아키텍처에 캐싱 기술을 나중에 추가하는 것보다 아키텍처적으로 해결하는 것이 더 나은 경우가 많습니다.

위상수학

공간 기반 아키텍처는 튜플 공간 이라는 개념에서 이름을 따왔는데 , 이는 공유 메모리를 통해 통신하는 여러 병렬 프로세서를 사용하는 기술입니다.
우주 기반 시스템은 시스템 내 동기적 제약 요소인 중앙 데이터베이스를 복제된 인메모리 데이터 그리드로 대체함으로써 높은 확장성, 탄력성 및 성능을 달성합니다.

애플리케이션 데이터는 메모리에 저장되고 모든 활성 처리 장치에 복제됩니다. 처리 장치가 데이터를 업데이트하면 데이터 펌프를 통해 해당 데이터를 데이터베이스로 비동기적으로 전송하며, 일반적으로 영구 큐를 사용하는 메시징 방식을 이용합니다. 처리 장치는 사용자 부하의 증감에 따라 동적으로 시작 및 종료되므로 가변적인 확장성을 제공합니다. 애플리케이션의 표준 트랜잭션 처리에는 중앙 데이터베이스가 사용되지 않으므로 데이터베이스 병목 현상이 제거됩니다.

이를 통해 애플리케이션 내에서 거의 무한대에 가까운 확장성이 가능해집니다.

공간 기반 아키텍처는 단일 요청을 처리하기 위해 함께 작동하는 여러 가지 구성 요소로 이루어진 복잡한 아키텍처 스타일입니다. 주요 구성 요소는 다음과 같습니다.

처리 장치
: 애플리케이션 기능을 포함합니다.

가상화된 미들웨어
: 처리 장치를 관리하고 조정하는 데 사용되는 인프라 관련 아티팩트 모음

메시징 그리드
: 입력 요청 및 세션 상태를 관리하는 데 사용됩니다.

데이터 그리드
: 처리 장치 간 데이터 동기화 및 복제를 관리합니다.

처리 그리드
: 여러 처리 장치 간의 요청 조정을 관리하는 데 사용됩니다.

배포 관리자
: 부하 증가 및 감소에 따라 처리 장치 인스턴스의 시작 및 종료를 관리합니다.

데이터 펌프
: 업데이트된 데이터를 데이터베이스에 비동기적으로 전송합니다.

데이터 작성자
: 데이터 펌프에서 업데이트를 수행하십시오.

데이터 판독기
: 데이터베이스 데이터를 읽어 시작 시 처리 장치로 전달합니다.

그림 16-2는 이러한 주요 유물을 보여줍니다.

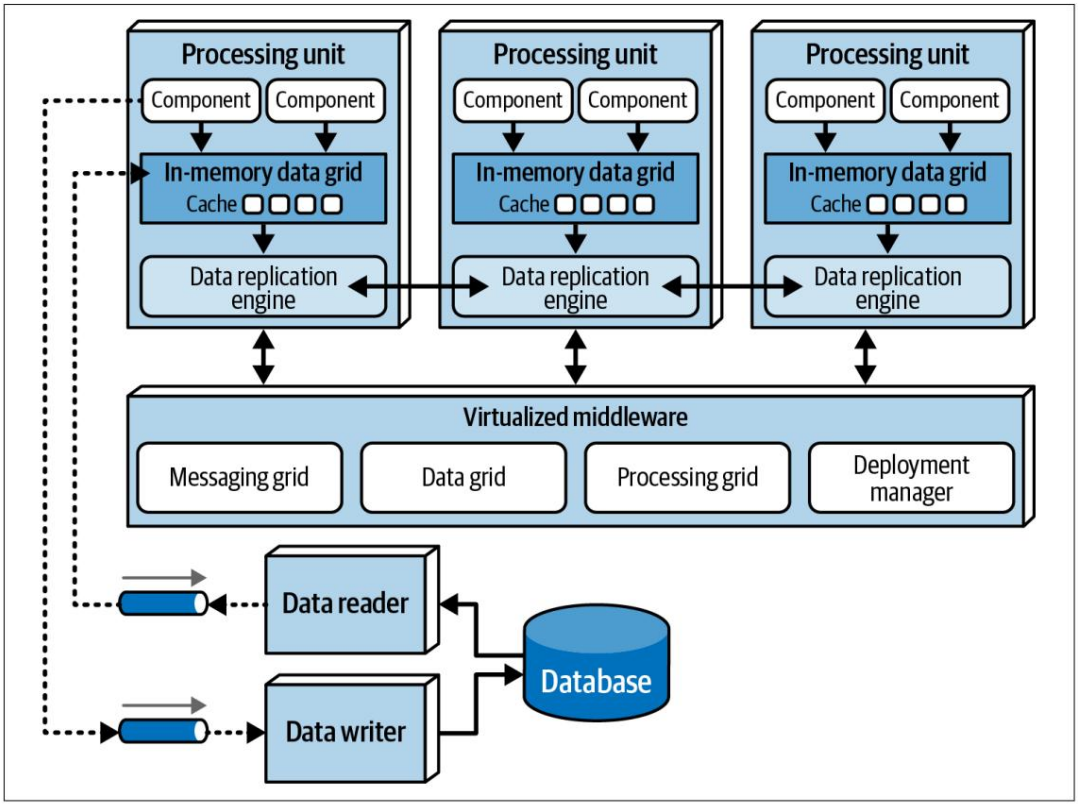

그림 16-2. 공간 기반 아키텍처의 기본 토폴로지

스타일 사양

다음 섹션에서는 이러한 주요 구성 요소와 그 작동 방식을 자세히 설명합니다.

처리 장치 (그림 16-3

참조) 는 애플리케이션 로직(또는 그 일부)을 포함하며, 일반적으로 웹 기반 구성 요소와 백엔드 비즈니스 로직을 포함합니다. 처리 장치의 내용은 애플리케이션 유형에 따라 다릅니다.

규모가 작은 웹 기반 애플리케이션은 일반적으로 단일 처리 장치에 배포되는 반면, 규모가 큰 애플리케이션은 기능 영역에 따라 여러 처리 장치로 기능을 분할하는 경우가 많습니다. 처리 장치는 마이크로서비스처럼 작고 단일 목적의 서비스들을 포함할 수도 있습니다. 또한, 처리 장치는 인메모리 데이터 그리드와 복제 엔진을 포함하며, 이는 일반적으로 Hazelcast, Apache Ignite, Oracle Coherence 와 같은 제품을 통해 구현됩니다 (288페이지의 "데이터 그리드" 참조).

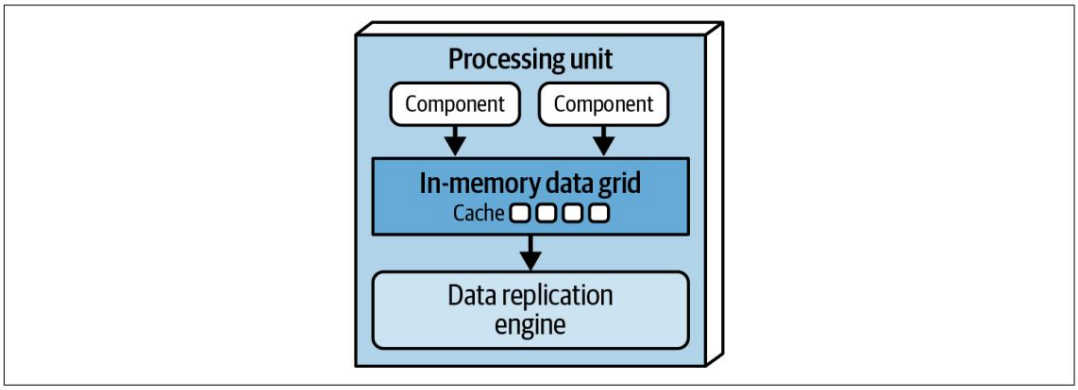

그림 16-3. e 프로세싱 유닛에는 애플리케이션의 기능이 포함되어 있습니다.

가상화된 미들웨어

그림 16-4 에 나타낸 가상화된 미들웨어는 다양한 인프라 관련 아티팩트를 포함하며 처리 장치를 관리하고 제어하는 데 사용됩니다. 최소한 이 미들웨어 아티팩트는 입력 요청 및 사용자 세션 상태를 관리하는 메시징 그리드, 데이터 복제 및 동기화를 관리하는 데이터 그리드, 그리고 필요에 따라 처리 장치 인스턴스를 시작하고 종료하는 배포 관리자를 포함합니다. 선택적으로, 가상화된 미들웨어는 단일 비즈니스 요청에 대해 둘 이상의 처리 장치를 오케스트레이션해야 하는 경우를 대비하여 처리 그리드를 포함할 수도 있습니다. 아키텍트는 필요에 따라 보안 기능, 관찰 가능성을 위한 메트릭 수집 등과 같은 다른 인프라 관련 기능을 가상화된 미들웨어에 추가할 수 있습니다.

가상화된 미들웨어의 모든 기능을 단일 제품으로 수행할 수 없기 때문에, 일반적으로 웹 서버, 캐싱 도구, 로드 밸런서, 서비스 오케스트레이터, 배포 관리자와 같은 타사 제품을 통해 구현되어 처리 장치의 모니터링, 시작 및 종료를 관리합니다. 이러한 미들웨어 구성 요소 각각에 대해서는 다음 섹션에서 자세히 설명합니다.

메시징 그리드

그림 16-4 에 나타낸 메시징 그리드는 가상화된 미들웨어의 일부이며 입력 요청과 세션 상태를 관리합니다. 가상화된 미들웨어에 요청이 들어오면 메시징 그리드 구성 요소는 요청을 수신할 수 있는 활성 처리 장치를 확인한 후 해당 처리 장치 중 하나로 요청을 전달합니다.

메시징 그리드의 복잡성은 단순한 라운드 로빈 알고리즘부터 어떤 처리 장치가 가장 사용 가능한지 추적하는 더욱 복잡한 차순위 사용 가능 알고리즘까지 다양할 수 있습니다. 이 구성 요소는 일반적으로 로드 밸런싱 기능을 갖춘 일반적인 웹 서버(예: HA Proxy 또는 Nginx)를 사용하여 구현됩니다.

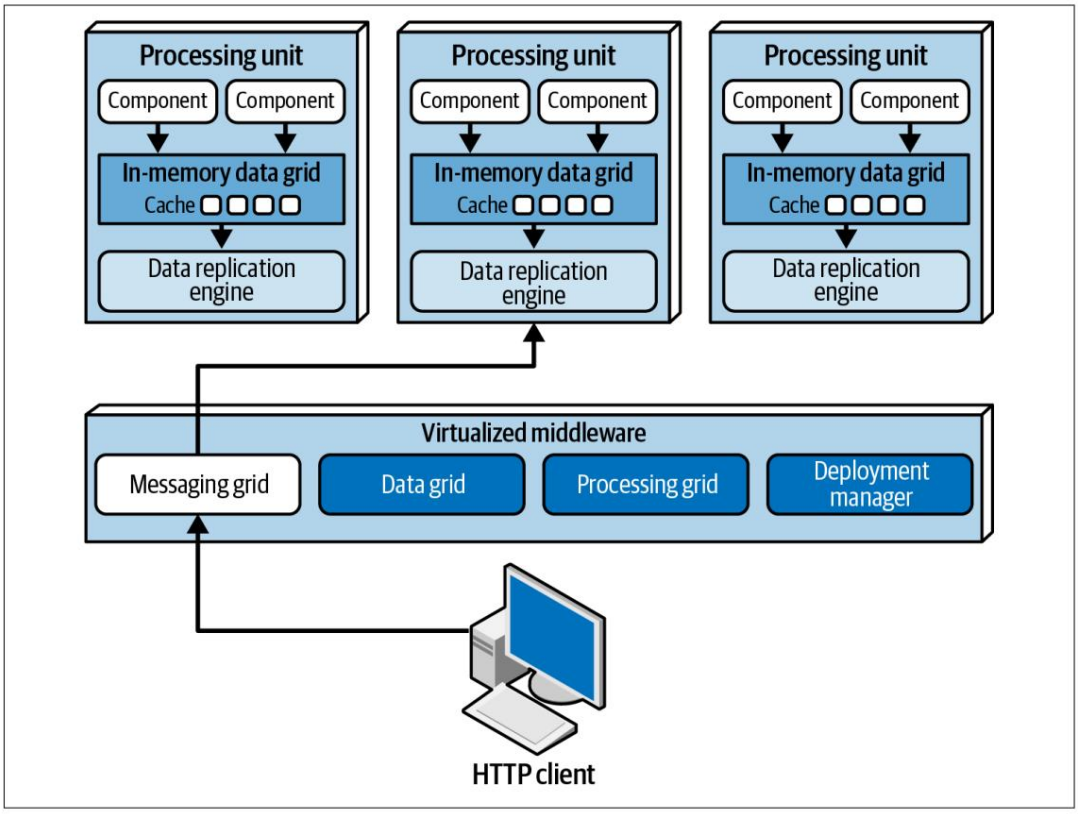

그림 16-4. e 메시징 그리드는 요청 및 세션 상태를 처리합니다.

데이터 그리드

데이터 그리드는 가상화된 미들웨어의 구성 요소, 어쩌면 가장 중요한 구성 요소일 것입니다. 대부분의 최신 구현에서 데이터 그리드는 처리 장치 내에 메모리 내 복제 캐시로만 구현됩니다(291페이지의 "복제 및 분산 캐싱" 참조). 그러나 외부 컨트롤러가 필요하거나 분산 캐시를 사용하는 복제 캐싱 구현의 경우, 이 기능은 처리 장치와 가상화된 미들웨어의 데이터 그리드 구성 요소 모두에 존재합니다.

메시징 그리드는 사용 가능한 모든 처리 장치로 요청을 전달할 수 있으므로 각 처리 장치의 메모리 데이터 그리드에 정확히 동일한 데이터가 포함되어 있어야 합니다. 그림 16-5 의 점선으로 표시된 것처럼 데이터 복제는 일반적으로 처리 장치 간에 비동기적으로 수행되며, 대개 100ms 이내에 완료됩니다.

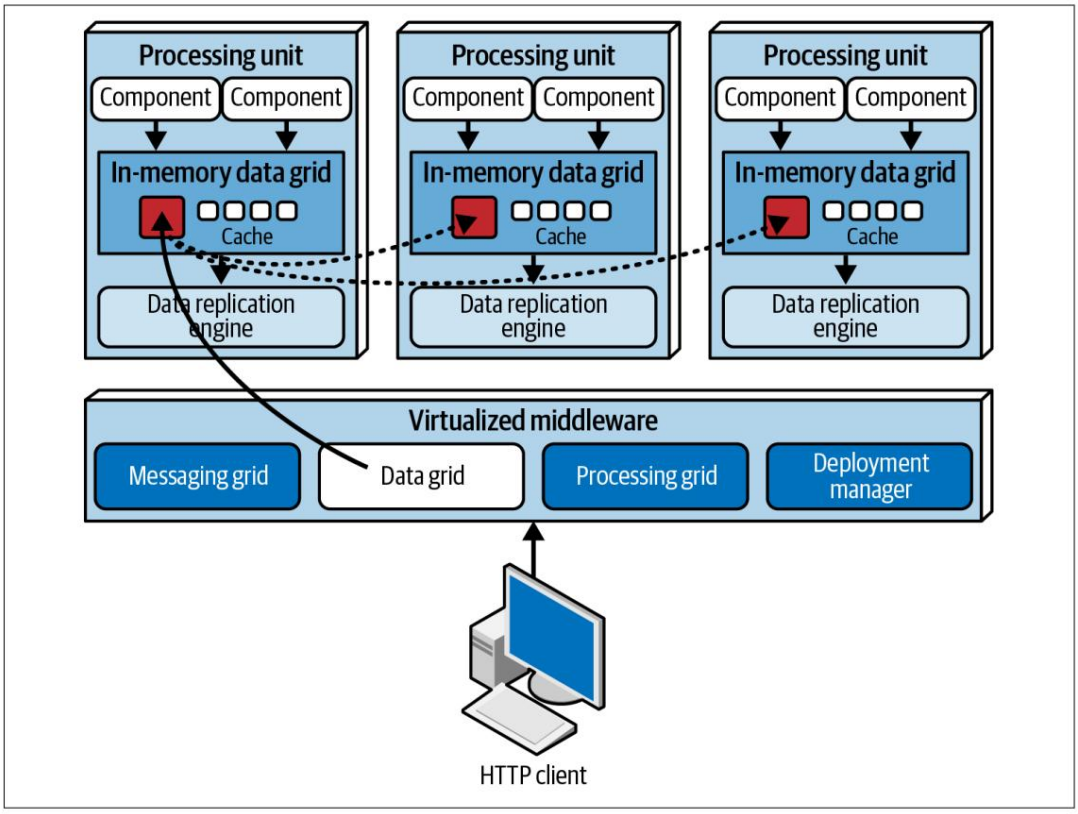

그림 16-5. e 데이터 그리드는 메모리 내 캐시를 동기화합니다.

동일한 이름의 데이터 그리드를 포함하는 처리 장치 간에 데이터가 동기화됩니다. 예를 들어, 다음 Java 코드는 Hazelcast를 사용하여 고객 프로필 정보를 포함하는 처리 장치용 내부 복제 데이터 그리드를 생성합니다.

```
HazelcastInstance hz = Hazelcast.newHazelcastInstance(); Map<String,
CustomerProfile> profileCache =
        hz.getReplicatedMap("고객 프로필");
```

고객 프로필 정보에 접근해야 하는 모든 처리 장치는 이 코드를 포함해야 합니다. 어떤 처리 장치가 CustomerProfile 캐시의 데이터를 업데이트하면 데이터 그리드는 동일한 이름의 CustomerProfile 캐시를 포함하는 다른 모든 처리 장치에 해당 업데이트를 복제합니다 . 처리 장치는 작업을 완료하는 데 필요한 만큼 메모리에 복제된 캐시를 저장할 수 있습니다. 또는, 한 처리 장치가 다른 처리 장치에 원격으로 데이터를 요청하는 호출(코레오그래피)을 수행하거나, 처리 그리드(다음 섹션에서 설명)를 활용하여 데이터 요청을 오케스트레이션할 수 있습니다(오케스트레이션과 코레오그래피에 대한 자세한 내용은 20장 375페이지 의 "오케스트레이션 과 코레오그래피의 차이점"을 참조하십시오).

처리 장치 내에서 데이터 복제를 사용하면 적어도 하나의 인스턴스에 명명된 복제 캐시가 있는 한, 추가 인스턴스가 데이터베이스에서 데이터를 읽지 않고 시작할 수 있습니다. 새 처리 장치 인스턴스가 시작될 때,

이 프로세스는 캐싱 공급자(예: Hazelcast)를 통해 동일한 이름의 캐시를 사용하는 다른 처리 장치에 연결을 요청하는 브로드캐스트를 보냅니다. 다른 처리 장치가 브로드캐스트 요청을 승인하고 새 처리 장치에 연결되면, 그중 하나(일반적으로 새 처리 장치에 가장 먼저 연결된 장치)가 캐시 데이터를 새 인스턴스로 전송하여 동일한 이름의 캐시를 사용하는 다른 모든 인스턴스와 동기화합니다.

각 처리 장치 인스턴스에는 동일한 이름의 캐시를 포함하는 다른 모든 처리 장치 인스턴스의 IP 주소와 포트가 포함된 멤버 목록이 있습니다.
예를 들어, 고객 프로필 기능과 메모리에 복제되어 캐시된 데이터를 포함하는 단일 처리 장치 인스턴스가 있다고 가정해 보겠습니다. 이 경우 인스턴스가 하나뿐이므로 멤버 목록에는 자기 자신만 포함됩니다. 이는 Hazelcast를 사용하여 생성된 다음 로깅 문에서 확인할 수 있습니다.

```
사례 1:
멤버 {크기:1, 버전:1} [ 멤버
        [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268 이 항목
]
```

다른 처리 장치가 동일한 이름의 캐시를 사용하여 시작되면 두 서비스의 멤버 목록이 각 처리 장치의 IP 주소와 포트를 반영하도록 업데이트됩니다.

```
사례 1:
멤버 {크기:2, 버전:2} [ 멤버
        [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268 이 멤버 [172.19.248.90]:5702
        - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316
]

사례 2:
멤버 {크기:2, 버전:2} [ 멤버
        [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268 멤버 [172.19.248.90]:5702
        - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316 이
]
```

세 번째 처리 장치가 시작되면 인스턴스 1과 인스턴스 2의 멤버 목록이 모두 새 세 번째 인스턴스를 반영하도록 업데이트됩니다.

```
사례 1:
멤버 {크기:3, 버전:3} [ 멤버
        [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268 이 멤버 [172.19.248.90]:5702
        - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316 멤버 [172.19.248.91]:5703 -
        1623eadf-9cfb-4b83-9983-d80520cef753
]

사례 2:
멤버 {크기:3, 버전:3} [ 멤버
        [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268 멤버 [172.19.248.90]:5702
        - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316 이 멤버 [172.19.248.91]:5703 -
        1623eadf-9cfb-4b83-9983-d80520cef753
]
```

```
인스턴스 3: 멤버 {크
기:3, 버전:3} [ 멤버 [172.19.248.89]:5701 -
        04a6f863-dfce-41e5-9d51-9f4e356ef268 멤버 [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316 멤버
        [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753 this

]
```

이제 세 인스턴스 모두 서로에 대해 알고 있습니다(멤버 라인 끝에 있는 ' this' 라는 단어로 표시되는 것처럼 자기 자신에 대해서도 알고 있습니다). 인스턴스 1이 고객으로부터 청구지 주소 업데이트 요청을 받았다고 가정해 보겠습니다. 인스턴스 1이 `cache.put()` 또는 이와 유사한 캐시 업데이트 메서드를 사용하여 캐시를 업데이트하면, 데이터 그리드(예: Hazelcast)는 동일한 업데이트를 다른 복제된 캐시에도 비동기적으로 적용하여 세 개의 고객 프로필 캐시 모두에 새 청구지 주소가 포함되도록 합니다. 따라서 세 인스턴스는 항상 동일한 데이터로 동기화된 상태를 유지합니다.

처리 장치 인스턴스가 다운되면 다른 모든 처리 장치의 멤버 목록이 자동으로 업데이트되어 변경 사항을 반영합니다. 예를 들어 인스턴스 2가 다운되면 캐싱 제품은 인스턴스 1과 3의 멤버 목록을 즉시 업데이트하여 다운된 인스턴스를 제거합니다.

```
사례 1:
멤버 {크기:2, 버전:4} [ 멤버 [172.19.248.89]:5701
        - 04a6f863-dfce-41e5-9d51-9f4e356ef268 이 멤버 [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753

]

인스턴스 3: 멤버 {크
기:2, 버전:4} [ 멤버 [172.19.248.89]:5701 -
        04a6f863-dfce-41e5-9d51-9f4e356ef268 멤버 [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753 이 항목

]
```

복제 및 분산 캐싱 공간 기반 아키텍처는

애플리케이션의 트랜잭션 처리를 위해 캐싱에 의존합니다.
공간 기반 아키텍처는 데이터베이스에 대한 직접적인 읽기 및 쓰기 작업을 없애주므로 높은 확장성, 탄력성 및 성능을 지원할 수 있습니다. 이 아키텍처 스타일은 주로 인메모리 복제 캐싱에 의존하지만 분산 캐싱도 사용할 수 있습니다.

그림 16-6 에 나타낸 복제 캐싱 방식에서는 각 처리 장치가 자체 메모리 데이터 그리드를 가지며, 동일한 이름의 캐시를 사용하여 모든 처리 장치 간에 동기화됩니다. 처리 장치 내의 캐시가 업데이트되면 다른 처리 장치도 자동으로 새 정보로 업데이트됩니다.

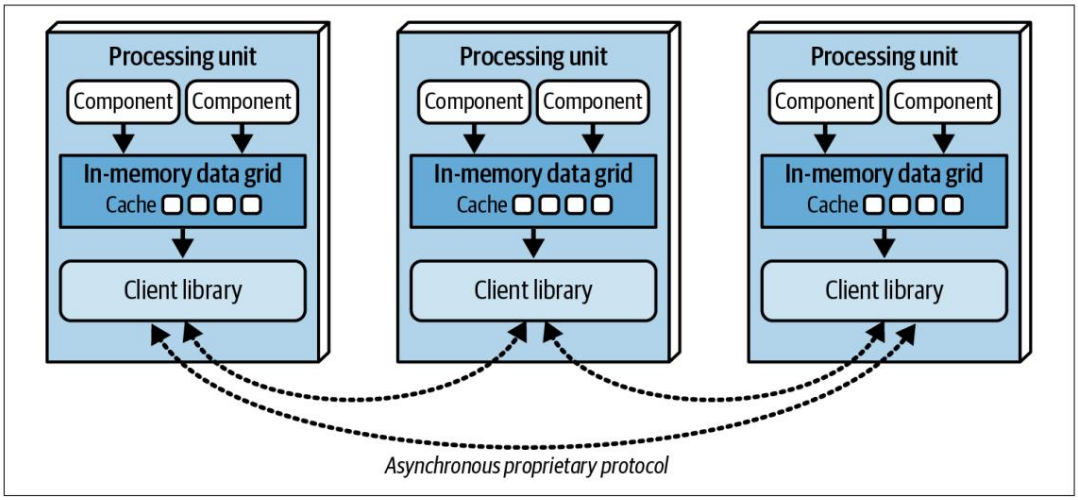

그림 16-6. 복제 캐싱은 처리 장치 간의 메모리 캐시를 동기화합니다.

복제 캐싱은 매우 빠를 뿐만 아니라 높은 수준의 내결함성을 지원합니다. 캐시를 저장하는 중앙 서버가 없기 때문에 복제 캐싱에는 단일 장애 지점이 없습니다.¹

복제 캐싱은 공간 기반 아키텍처의 표준 캐싱 모델이지만, 사용할 수 없는 경우도 있습니다. 예를 들어 시스템이 대용량 데이터를 처리해야 하는 경우가 그렇습니다. 내부 메모리 캐시가 100MB를 초과하면 각 처리 장치가 너무 많은 메모리를 사용해야 하므로 탄력성과 확장성에 문제가 발생할 수 있습니다. 처리 장치는 일반적으로 가상 머신이나 컨테이너(예: Docker) 내에 배포되는데, 각 가상 머신이나 컨테이너는 내부 캐시 사용에 사용할 수 있는 메모리 용량이 제한되어 있습니다. 따라서 높은 처리량을 처리해야 하는 상황에서 시작할 수 있는 처리 장치 인스턴스 수가 제한됩니다.

복제된 데이터 캐시가 제대로 작동하지 않는 또 다른 상황은 캐시 데이터가 매우 자주 업데이트되는 경우입니다. 304페이지의 "데이터 충돌" 에서 보여주는 것처럼 캐시 데이터의 업데이트 속도가 너무 높으면 데이터 그리드가 이를 따라가지 못해 모든 처리 장치 인스턴스에서 데이터 일관성이 손상될 수 있습니다. 이러한 상황이 발생하면 대부분의 아키텍트는 분산 캐시를 사용하는 것을 선택합니다.

1. 사용되는 캐싱 제품의 구현 방식에 따라 이 규칙에 예외가 있을 수 있습니다. 일부 캐싱 제품은 처리 장치 간의 데이터 복제를 모니터링하고 제어하기 위해 외부 컨트롤러를 필요로 하지만, 대부분은 이러한 모델에서 벗어나고 있습니다.

그림 16-7 에 나타낸 분산 캐싱은 중앙 집중식 캐시를 저장하는 전용 외부 서버 또는 서비스를 필요로 합니다. 이 모델에서 처리 장치는 내부 메모리에 데이터를 저장하지 않고, 자체 프로토콜을 사용하여 중앙 캐시 서버의 데이터에 접근합니다. 분산 캐싱은 모든 데이터가 한 곳에 저장되고 복제할 필요가 없기 때문에 높은 수준의 데이터 일관성을 지원합니다. 그러나 캐시 데이터에 원격으로 접근해야 하므로 시스템의 전체 지연 시간이 증가하여 복제 캐싱만큼 성능이 좋지는 않습니다.

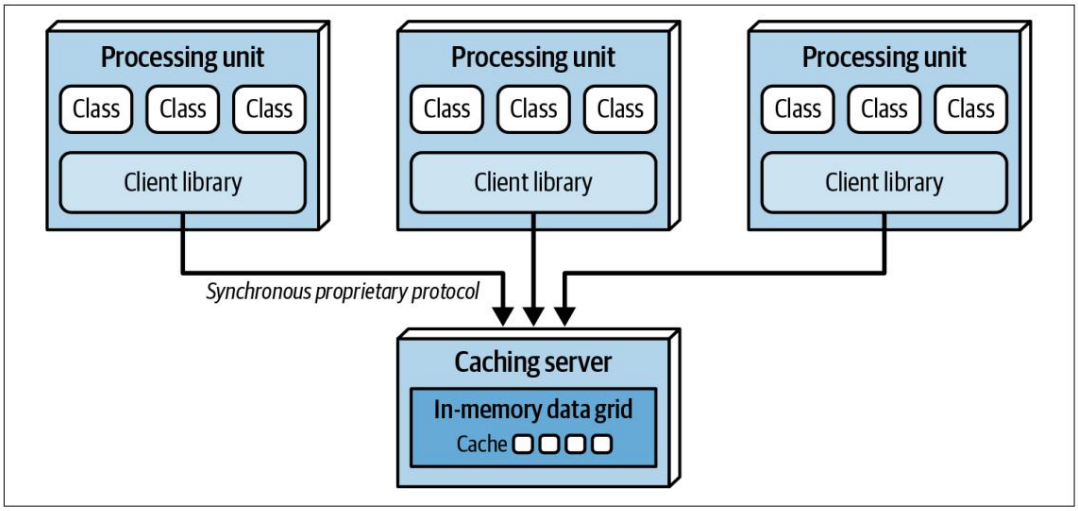

그림 16-7. 분산 캐싱은 처리 장치 간에 우수한 데이터 일관성을 제공합니다.

분산 캐싱에서 내결함성 또한 중요한 문제입니다. 데이터가 저장된 캐시 서버가 다운되면 어떤 처리 장치도 데이터에 접근하거나 업데이트할 수 없게 됩니다. 이러한 단일 장애 지점은 분산 캐시를 미러링하여 완화할 수 있지만, 기본 캐시 서버가 예기치 않게 다운되어 데이터가 미러링된 캐시 서버에 도달하지 못하는 경우 데이터 일관성 문제가 발생할 수 있습니다.

캐시 크기가 비교적 작을 때(100MB 미만) 업데이트 속도가
캐시 용량이 충분히 낮아서 캐싱 제품의 복제 엔진이 유지 관리를 할 수 있습니다.
따라서 복제 캐시와 분산 캐시 중 어느 것을 사용할지 결정해야 합니다.
데이터 일관성과 성능 및 내결함성 중 무엇을 우선시할 것인지에 대한 문제입니다.
분산 캐시는 복제 캐시보다 항상 더 나은 데이터 일관성을 제공합니다.
데이터가 여러 처리 장치에 분산되지 않고 한 곳에 있기 때문입니다. 어떻게 그럴 수 있죠?
하지만 복제 캐시를 사용하면 성능과 내결함성이 항상 더 좋아집니다.
많은 경우, 결정적인 요소는 캐시되는 데이터 유형이 됩니다.
처리 장치. 시스템의 주요 요구 사항이 높은 일관성을 가진 데이터(예:
사용 가능한 제품의 재고량을 파악하는 것은 일반적으로 분산 캐시를 필요로 합니다.
데이터가 자주 변경되지 않는 경우(예: 이름/값 쌍, 제품 등과 같은 참조 데이터)
코드 및 제품 설명과 같은 정보를 빠르게 조회하려면 복제 캐시를 사용하는 것을 고려해 보세요.
표 16-1은 분산형 시스템과 단일형 시스템을 언제 사용할지 선택하는 데 도움이 되는 몇 가지 기준을 요약한 것입니다.
복제된 캐시.

표 16-1. 분산 캐싱과 복제 캐싱 비교

결정 기준	복제 캐시	분산 캐시
최적화	성능	일관성
캐시 크기	소형(<100MB)	대형(>500MB)
데이터 유형	상대적으로 정적	매우 역동적인
업데이트 빈도가 비교적 낮음		높은 업데이트 속도
내결함성	높은	낮은

공간 기반 아키텍처에서 사용할 캐싱 모델을 선택할 때 다음 사항을 기억하세요!
대부분의 경우 두 모델 모두 적용 가능하다는 점을 기억하십시오. 다시 말해, 어느 쪽도 정답이 될 수 없습니다.
복제 캐싱이나 분산 캐싱이 모든 문제를 해결하는 것은 아닙니다. 서로 다른 처리 방식이 필요합니다.
각 장치는 서로 다른 모델을 사용할 수도 있습니다. 단일 모델을 선택하여 타협하는 대신,
애플리케이션 전체에 걸쳐 일관된 캐싱 모델을 사용하고, 각 모델의 강점을 활용합니다.
예를 들어, 현재 재고를 유지하는 처리 장치의 경우 다음을 선택하십시오.
데이터 일관성을 유지하기 위한 분산 캐싱 모델; 데이터 일관성을 유지하는 처리 장치에 적용
고객 프로필에 따라 성능 및 내결함성을 위해 복제 캐시를 선택하십시오.

니어캐시 고려 사항

니어 캐시는 인메모리 데이터 그리드와 분산 캐시를 결합한 하이브리드 캐싱 모델입니다. 이 모델(그림 16-8 참조) 에서 분산 캐시는 전체 백킹 캐시라고 하며, 각 처리 장치 내의 인메모리 데이터 그리드는 프런트 캐시라고 합니다. 프런트 캐시는 항상 전체 백킹 캐시의 작은 부분 집합을 저장하며, 오래된 항목을 제거하고 새로운 항목을 추가하는 제거 정책을 사용합니다. 프런트 캐시는 세 가지 제거 정책 옵션을 제공합니다. 가장 최근에 사용된 항목을 저장하는 '최근 사용 캐시', 가장 자주 사용된 항목을 저장하는 '최근 사용 캐시', 그리고 공간이 필요할 때 무작위로 항목을 제거하는 '임의 교체' 정책이 있습니다. 무작위 교체는 최근 사용 캐시나 가장 자주 사용된 캐시 중 어느 쪽을 유지해야 할지 명확한 데이터 분석 결과가 없을 때 적합한 제거 정책입니다.

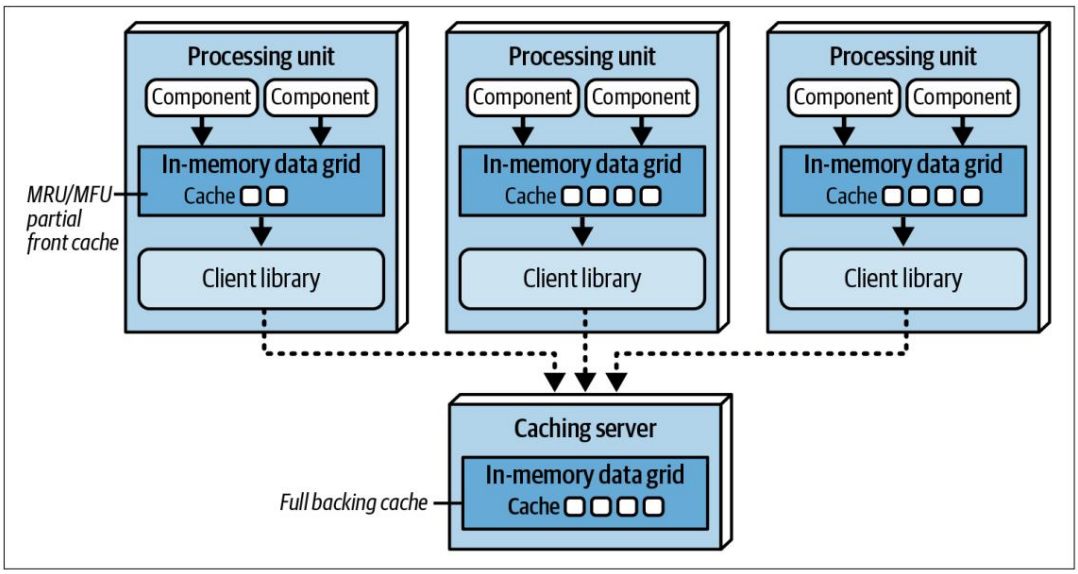

그림 16-8. 니어캐시 모델은 프런트캐시와 백캐시를 모두 사용합니다.

프런트 캐시는 항상 전체 백킹 캐시와 동기화되지만, 각 처리 장치 내에 있는 프런트 캐시는 동일한 데이터를 공유하는 다른 처리 장치 간에 동기화되지 않습니다. 즉, 동일한 데이터 컨텍스트(예: 고객 프로필)를 공유하는 여러 처리 장치의 프런트 캐시에 저장된 데이터가 모두 다를 수 있습니다. 이는 처리 장치 간의 성능 및 응답성 불일치를 초래하므로 공간 기반 아키텍처에서 니어 캐시 모델을 사용하는 것은 권장하지 않습니다.

처리 그리드 (그림

16-9 참조) 는 가상화된 미들웨어 내의 선택적 구성 요소로, 단일 비즈니스 요청에 여러 처리 장치가 관련된 경우 조정된 요청 처리를 관리합니다. 요청에 둘 이상의 유형의 처리 장치(예: 주문 처리 장치와 결제 처리 장치) 간의 조정이 필요한 경우, 처리 그리드는 해당 두 처리 장치 간의 요청을 중재하고 조정합니다.

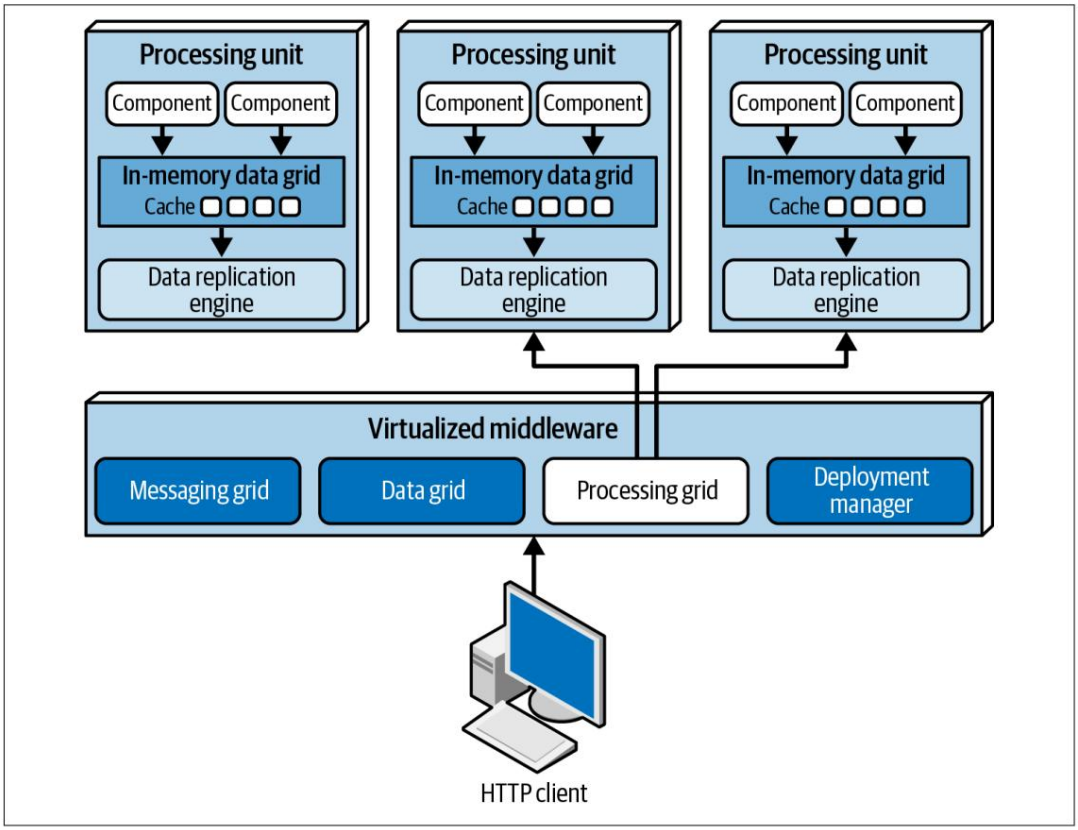

그림 16-9. e 프로세싱 그리드는 처리 장치 간의 오케스트레이션을 관리합니다.

대부분의 최신 공간 기반 구현(특히 세분화된 서비스를 사용하는 경우)은 단일의 거시적 오케스트레이션 엔진 대신 개별적인 세분화된 오케스트레이션 처리 장치를 통해 처리 그리드 기능을 구현하며, 각 오케스트레이션 처리 장치는 하나의 주요 워크플로를 처리합니다. 전자상거래를 예로 들면, 고객이 주문을 할 때 주문 접수, 결제, 재고 조정의 세 가지 처리 장치가 조정되어야 한다고 가정해 보겠습니다. 아키텍트는 이 세 가지 처리 장치를 조율하는 주문 접수 오케스트레이터 처리 장치를 만들 수 있습니다. 또한 주문 반품 처리 및 재고 보충과 같은 다른 주요 워크플로에 대해서도 별도의 오케스트레이션 처리 장치를 만들 수 있습니다.

배포 관리자

배포 관리자 구성 요소는 부하 조건에 따라 처리 장치 인스턴스의 동적 시작 및 종료를 관리합니다. 이 구성 요소는 응답 시간과 사용자 부하를 지속적으로 모니터링하고, 부하가 증가하면 새로운 처리 장치를 시작하고, 부하가 감소하면 처리 장치를 종료합니다. 이는 애플리케이션의 가변적 확장성(탄력성)을 달성하는 데 매우 중요합니다. 대부분의 클라우드 기반 인프라와 Kubernetes와 같은 서비스 오케스트레이션 제품은 이러한 기능을 제공합니다.

데이터 펌프

는 데이터를 다른 프로세서로 전송하여 데이터베이스를 업데이트하는 방식입니다. 공간 기반 아키텍처에서는 처리 장치가 데이터베이스에서 직접 읽고 쓰지 않기 때문에 데이터 펌프가 필요합니다. 공간 기반 아키텍처의 데이터 펌프는 항상 비동기적으로 작동하여 메모리 캐시와 데이터베이스 간의 최종 일관성을 보장합니다. 처리 장치 인스턴스가 요청을 수신하고 캐시를 업데이트하면 해당 처리 장치가 업데이트의 소유자가 되어 데이터 펌프를 통해 데이터베이스를 최종적으로 업데이트할 수 있도록 업데이트를 전송해야 합니다.

데이터 펌프는 일반적으로 그림 16-10 에 표시된 것처럼 메시징을 사용하는 공간 기반 아키텍처에서 구현됩니다 .

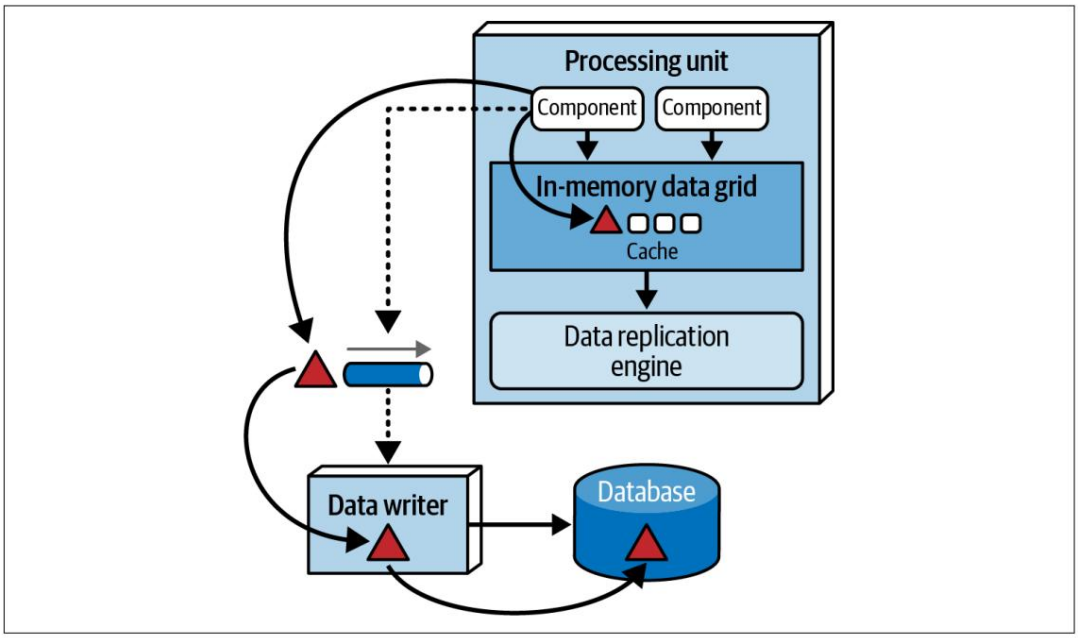

그림 16-10. 데이터 펌프는 데이터를 데이터베이스로 전송하는 데 사용됩니다.

메시징은 비동기 통신을 지원할 뿐만 아니라, 선입선출(FIFO) 큐잉을 통해 메시지 전달 보장, 메시지 영속성 및 메시지 순서 보장을 제공합니다. 또한 메시징은 처리 장치와 데이터 기록기를 분리하여 데이터 기록기를 사용할 수 없는 경우에도 처리 장치 내의 처리가 중단되지 않도록 합니다.

대부분의 공간 기반 아키텍처에는 여러 개의 데이터 펌프가 있습니다. 일반적으로 각 데이터 펌프는 특정 도메인 또는 하위 도메인(예: 고객 또는 재고)에 전용으로 사용되지만, 각 캐시 유형(예: 고객 프로필, 고객 위시리스트 등) 또는 훨씬 더 큰 일반 캐시를 포함하는 처리 장치 도메인(예: 고객) 에 전용으로 사용될 수도 있습니다.

데이터 펌프는 일반적으로 계약(Contract)을 포함하며, 이 계약에는 계약 데이터와 관련된 작업(추가, 삭제 또는 업데이트)이 명시되어 있습니다. 계약은 JSON 스키마, XML 스키마, 객체 또는 값 기반 메시지(이름-값 쌍을 포함하는 맵 메시지)일 수 있습니다.
업데이트의 경우, 데이터 펌프의 메시지 페이로드에는 일반적으로 새로운 데이터 값만 포함됩니다. 예를 들어, 고객이 프로필에서 전화번호를 변경한 경우, 새 전화번호와 고객 ID, 그리고 데이터를 업데이트하기 위한 작업만 전송됩니다.

데이터 작성자

데이터 라이터 구성 요소는 데이터 펌프에서 메시지를 수신하고 페이로드에 포함된 정보로 데이터베이스를 업데이트합니다(그림 16-10 참조). 데이터 라이터는 서비스, 애플리케이션 또는 데이터 허브(예: Ab Initio)로 구현될 수 있습니다. 데이터 라이터의 세분성은 데이터 펌프 및 처리 장치의 범위에 따라 달라질 수 있습니다.

도메인 기반 데이터 라이터는 읽어들이는 데이터 펌프의 수와 관계없이 특정 도메인(예: 주문 처리) 내의 모든 업데이트를 처리하는 데 필요한 데이터베이스 로직을 포함합니다. 그림 16-11 의 시스템은 고객 도메인 (프로필, 위시리스트, 지갑, 선호도) 을 나타내는 4개의 서로 다른 처리 장치와 4개의 서로 다른 데이터 펌프를 가지고 있지만 , 데이터 라이터는 하나뿐입니다.

해당 단일 고객 데이터 기록기는 네 개의 데이터 펌프 모두의 데이터를 수신하고 데이터베이스에서 고객 관련 데이터를 업데이트하는 데이터베이스 로직(예: SQL)을 포함합니다.

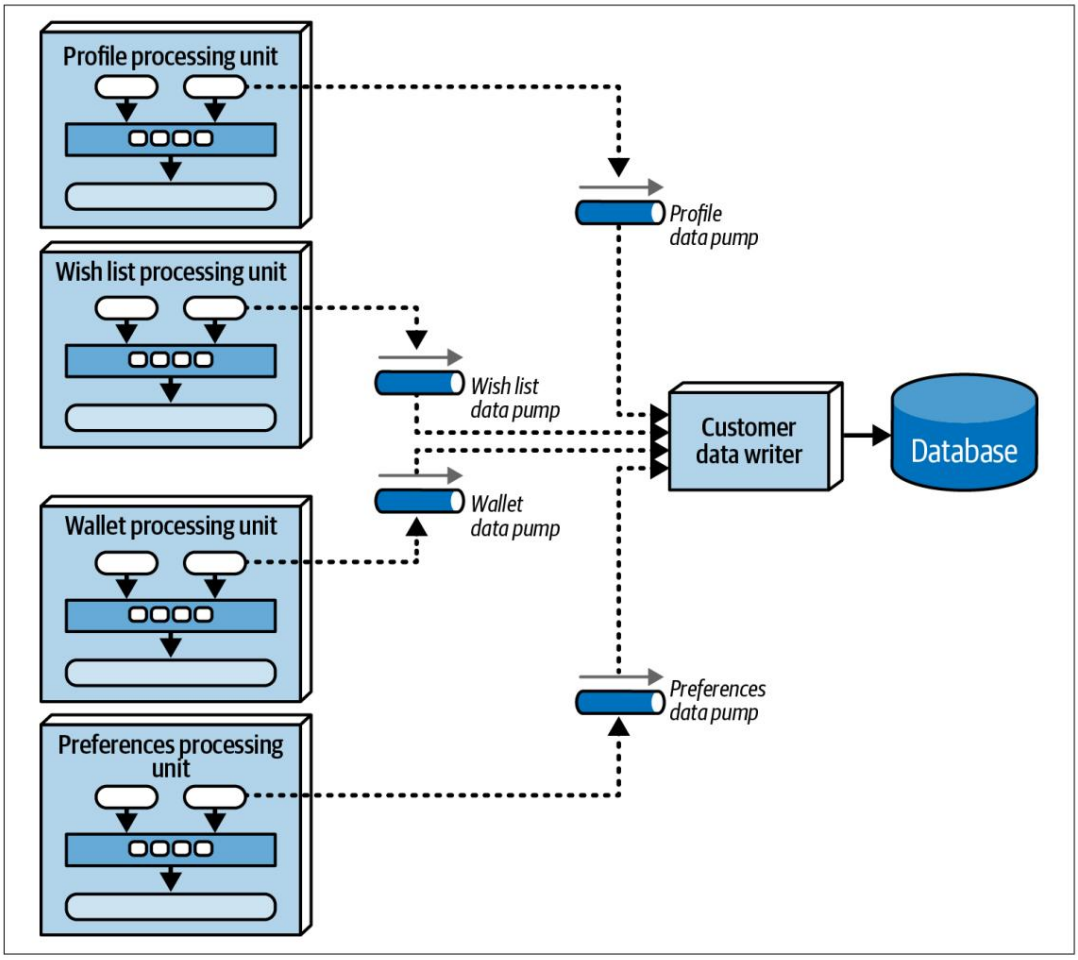

그림 16-11. 도메인 기반 데이터 작성기

또는 그림 16-12 에서처럼 각 처리 장치 클래스에 전용 데이터 라이터 구성 요소를 할당할 수도 있습니다. 이 모델에서 각 데이터 라이터는 해당 데이터 펌프에 전용으로 연결되며 특정 처리 장치(예: 지갑)에 대한 데이터베이스 처리 로직만 포함합니다. 이 모델은 데이터 라이터 구성 요소가 많아지는 경향이 있지만, 처리 장치, 데이터 펌프 및 데이터 라이터를 정렬하여 확장성과 민첩성을 향상시킵니다.

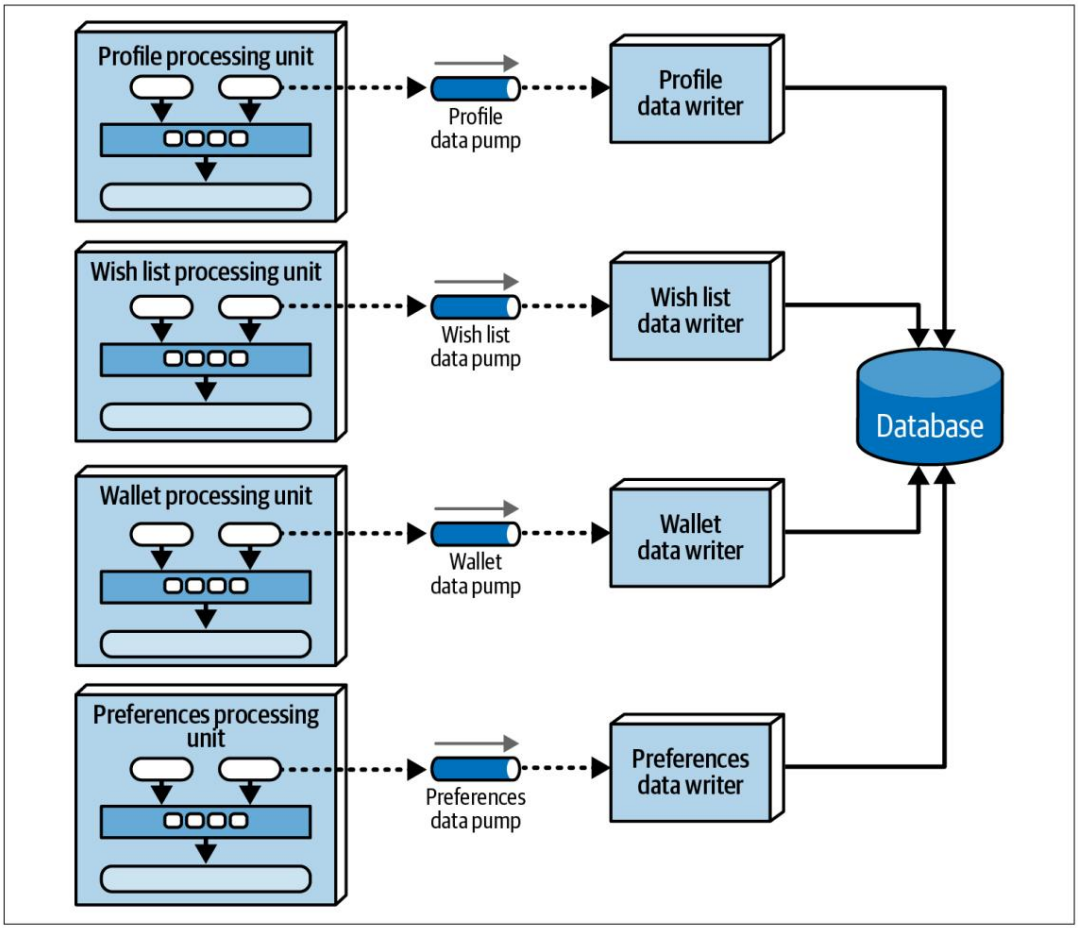

그림 16-12. 각 데이터 펌프용 전용 데이터 기록기

데이터 판독기

데이터 작성자는 데이터베이스를 업데이트하는 역할을 담당하는 반면, 데이터 판독자는 데이터베이스에서 데이터를 읽어 역방향 데이터 펌프(잠시 후 자세히 설명)를 통해 처리 장치로 전송합니다. 공간 기반 아키텍처에서 데이터 판독기는 다음 세 가지 상황 중 하나에서만 호출됩니다. 동일한 이름의 캐시에 속한 모든 처리 장치 인스턴스가 충돌하는 경우, 동일한 이름의 캐시에 속한 모든 처리 장치가 재배포되는 경우, 또는 복제된 캐시에 포함되지 않은 아카이브 데이터를 검색해야 하는 경우입니다.

시스템 전체 충돌이나 재배포로 인해 모든 인스턴스가 다운되면 데이터베이스에서 캐시로 데이터를 로드해야 합니다. 이는 일반적으로 공간 기반 아키텍처에서 피하려고 하는 상황입니다. 특정 유형의 처리 장치 인스턴스가 다시 시작되면 각 인스턴스는 캐시에 대한 잠금을 획득하려고 시도합니다. 잠금을 가장 먼저 획득한 인스턴스가 임시 캐시 소유자가 되고, 나머지 인스턴스는 잠금이 해제될 때까지 대기 상태가 됩니다. (캐시 구현 방식에 따라 다를 수 있지만, 이 시나리오에서는 기본적으로 캐시의 기본 소유자가 한 명 있습니다.) 데이터를 로드하려면

임시 캐시 소유자는 데이터 요청 메시지를 큐로 보냅니다. 데이터 판독기 구성 요소는 읽기 요청을 수락한 후 필요한 데이터를 검색하기 위한 데이터베이스 쿼리 로직을 수행합니다. 검색된 데이터는 역방향 데이터 펌프라는 다른 큐로 전송되고, 역방향 데이터 펌프는 데이터를 임시 캐시 소유자 처리 장치로 보냅니다. 캐시에 데이터가 로드되면 임시 소유자는 잠금을 해제합니다. 그러면 다른 모든 인스턴스가 동기화되고 처리가 시작될 수 있습니다. 이 처리 흐름은 그림 16-13에 나와 있습니다.

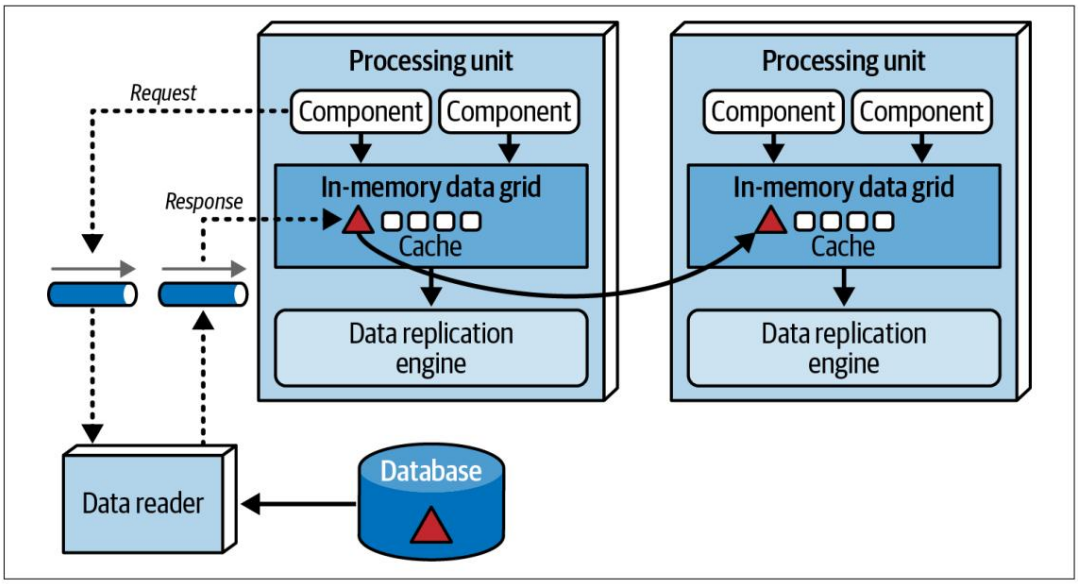

그림 16-13. 데이터 판독기는 처리 장치로 데이터를 전송합니다.

데이터 기록기와 마찬가지로 데이터 판독기도 도메인 기반이거나 특정 유형의 처리 장치에 전용으로 사용될 수 있습니다(후자가 더 일반적입니다). 구현 방식 또한 데이터 기록기와 동일하게 서비스, 애플리케이션 또는 데이터 허브를 사용할 수 있습니다.

데이터 기록기와 데이터 판독기는 기본적으로 데이터 추상화 계층(또는 경우에 따라 데이터 접근 계층)을 형성합니다. 둘의 차이점은 처리 장치가 데이터베이스 스키마(테이블 구조)에 대해 얼마나 자세한 정보를 가지고 있는지에 있습니다. 데이터 접근 계층에서는 처리 장치가 데이터베이스의 기본 데이터 구조에 직접 연결되어 있으며, 데이터 판독기와 기록기를 통해서만 간접적으로 데이터베이스에 접근합니다. 반면, 데이터 추상화 계층에서는 처리 장치가 별도의 계약을 사용하여 기본 스키마로부터 분리됩니다.

공간 기반 아키텍처는 일반적으로 데이터 추상화 계층 모델에 의존하므로 각 처리 장치의 복제된 캐시 스키마가 기본 데이터베이스 스키마와 다를 수 있습니다. 즉, 데이터베이스에 대한 증분 변경이 처리 장치에 반드시 영향을 미치는 것은 아닙니다. 이러한 증분 변경을 용이하게 하기 위해 데이터 기록기와 읽기 장치에는 변환 로직이 포함되어 있습니다. 열 유형이 변경되거나

컬럼이나 테이블이 삭제되면 데이터 읽기/쓰기 작업은 필요한 변경 사항이 처리 장치 캐시에 반영될 때까지 데이터베이스 변경 사항을 버퍼링할 수 있습니다.

데이터 토폴로지

처리 장치가 데이터베이스와 직접 상호 작용하지 않기 때문에 공간 기반 아키텍처는 사용할 수 있는 데이터베이스 토폴로지 측면에서 매우 유연합니다. 비동기 데이터 펌프와 데이터 읽기/쓰기 장치를 결합하여 사용하면 요청 처리(트랜잭션)가 데이터베이스와 거의 독립적으로 이루어지므로 아키텍트는 데이터베이스 토폴로지와 데이터베이스 유형에 대해 폭넓은 선택권을 갖게 됩니다.

데이터베이스 토폴로지 선택은 여러 요인의 영향을 받습니다. 공간 기반 아키텍처에서 가장 중요한 결정 요인은 시스템이 백업 데이터베이스를 어떻게 활용할 것인가입니다. 예를 들어, 보고 및 데이터 분석이 특히 중요하다면 단일체 데이터베이스 토폴로지가 더 효과적일 수 있습니다. 단, 보고 및 데이터 분석이 데이터 메시를 통해 이루어지는 경우에는 도메인 기반 데이터베이스 토폴로지가 더 적합할 수 있습니다.

데이터베이스 토폴로지를 선택할 때 처리량과 전반적인 도메인 기반 데이터 일관성 또한 고려해야 할 사항입니다. 단일 모놀리식 데이터베이스는 동기화 과정에서 병목 현상을 일으켜 전체 동기화 시간을 지연시키고 데이터 일관성을 저해할 수 있습니다. 반면, 데이터를 도메인별로 명확하게 분할할 수 있다면 도메인 기반 데이터베이스 토폴로지가 전반적인 동기화 시간을 단축하고 데이터 일관성을 향상시킬 수 있습니다. 마지막으로, 하위 시스템에서 데이터베이스를 추가 처리에 사용해야 하는 경우에는 모놀리식 데이터베이스 토폴로지가 더 적합할 수 있습니다.

클라우드 고려 사항

클라우드 기반 아키텍처는 배포 환경과 관련하여 몇 가지 고유한 옵션을 제공합니다. 처리 장치, 가상화된 미들웨어, 데이터 펌프, 데이터 읽기/쓰기 장치 및 데이터베이스를 포함한 전체 시스템을 클라우드 기반 환경 또는 온프레미스(온프레미스)에 배포할 수 있습니다. 하지만 이 아키텍처 스타일은 그림 16-14에서 보는 것처럼 두 환경 모두에 동시에 배포할 수도 있는데, 이는 다른 아키텍처 스타일에서는 찾아볼 수 없는 독특하고 강력한 기능입니다. 이러한 하이브리드 토폴로지에서 애플리케이션은 관리형 클라우드 기반 환경의 처리 장치와 가상화된 미들웨어를 통해 배포되는 반면, 물리적 데이터베이스와 관련 데이터는 온프레미스에 유지됩니다. 이는 비동기 데이터 펌프와 최종 일관성 모델 덕분에 매우 효율적인 클라우드 기반 데이터 동기화를 지원합니다. 트랜잭션 처리는 동적이고 탄력적인 클라우드 기반 환경에서 수행되는 반면, 물리적 데이터 관리, 보고 및 데이터 분석은 안전한 온프레미스 환경에서 이루어집니다.

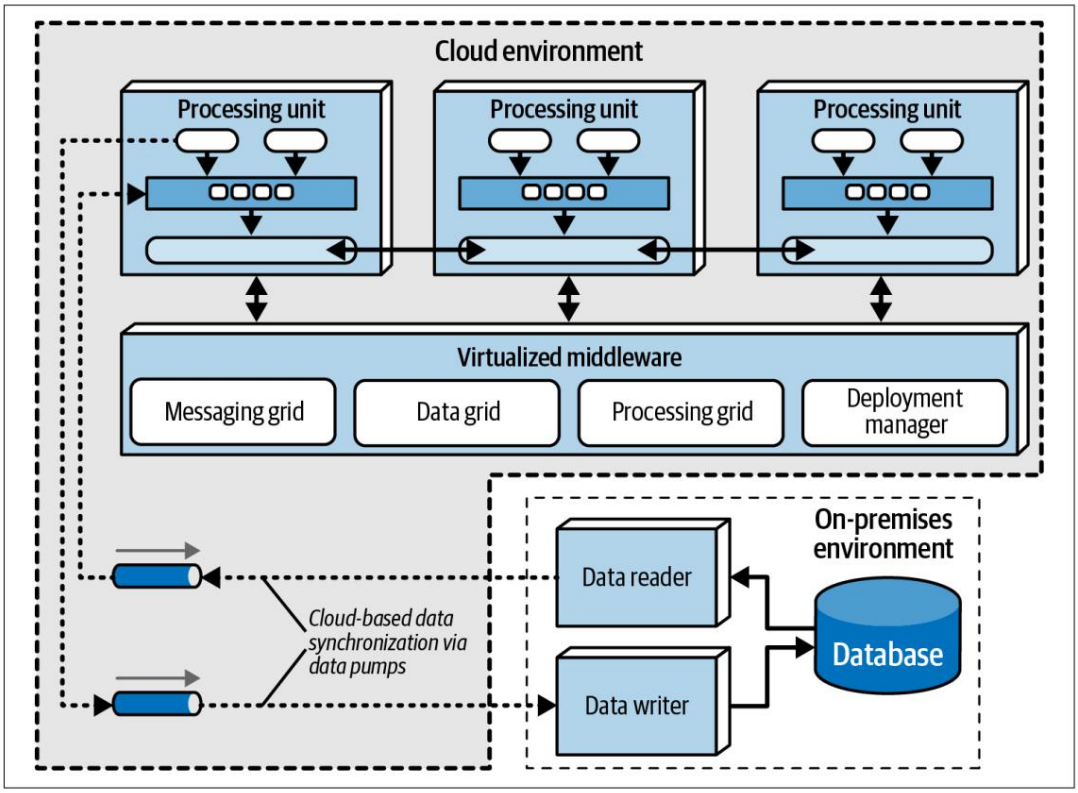

그림 16-14. 클라우드 기반 및 온프레미스 하이브리드 토폴로지

하이브리드 방식이든 아니든 클라우드 인프라와 클라우드 기반 서비스의 탄력적인 특성은 이러한 아키텍처 스타일과 잘 어울리므로 클라우드 기반 환경은 우주 기반 아키텍처에 적합한 선택입니다.

일반적인 위험

놀랄 것도 없이, 공간 기반 건축 양식과 관련된 대부분의 위험은 캐싱 및 백그라운드 데이터 동기화를 사용한다는 점을 고려할 때 데이터와 관련이 있습니다.
다음 섹션에서는 몇 가지 일반적인 위험 요소를 설명합니다.

데이터베이스에서 자주 읽는 항목

공간 기반 아키텍처는 모든 트랜잭션 데이터에 캐싱을 사용하여 높은 확장성과 동시성을 달성합니다. 이는 시스템이 과도한 데이터베이스 읽기 및 쓰기 작업을 수행하는 것을 방지하여 시스템의 전반적인 확장성, 탄력성 및 응답성을 저해하는 요소를 제거하고, 이러한 아키텍처 스타일의 장점을 실현하는 데 도움이 됩니다.

데이터베이스 읽기는 일반적으로 두 가지 시나리오에서만 발생합니다. 하나는 아카이브된 데이터(예: 주문 내역 또는 과거 은행 거래 내역)를 읽는 것이고, 다른 하나는 처리 장치를 콜드 스타트하는 것(다른 인스턴스가 실행 중이지 않은 상태에서 처리 장치를 처음 시작하는 것)입니다. 캐시된 데이터가 있는 경우

데이터 양이 매우 많아서 대부분의 데이터를 백업 데이터베이스에 보관하고 검색해야 하거나, 처리 장치가 다운되거나 자주 재배포되는 경우, 이러한 아키텍처는 문제 영역에 적합하지 않을 수 있습니다.

데이터 동기화 및 일관성: 데이터 펌프와 데이터 라이터

가 메모리 캐시와 데이터베이스 간의 데이터를 동기화하기 때문에 공간 기반 아키텍처에서는 데이터가 항상 최종적으로 일관성을 유지합니다. 그러나 이 방식은 일반적으로 동시 사용자 부하가 매우 높은 상황에서 사용되므로 데이터 펌프에서 병목 현상이 발생하기 쉽습니다. 이러한 병목 현상은 데이터가 백업 데이터베이스에 도달하는 데 상당한 지연을 초래할 수 있습니다. 하위 시스템에서 업데이트된 데이터를 신속하게 사용해야 하는 경우 이는 심각한 위험이 될 수 있습니다.

데이터 동기화와 관련된 또 다른 내재적 위험은 데이터 펌프 내에서의 데이터 손실입니다. 이러한 위험은 일반적으로 영구 큐(큐의 데이터가 메모리뿐만 아니라 디스크에도 저장되는 방식)를 사용하고, 데이터 펌프에서 데이터를 읽을 때 데이터 라이터에서 클라이언트 승인 모드를 사용함으로써 완화됩니다. 클라이언트 승인 모드는 데이터 라이터가 메시지 브로커에 처리가 완료되었음을 승인할 때까지 메시지를 큐에 유지합니다. 메시지 처리 중 메시지 브로커는 다른 데이터 라이터가 처리 중인 메시지를 읽지 못하도록 합니다.

이러한 기술은 데이터 펌프에서 데이터 손실을 방지하는 데 도움이 되지만, 전반적인 응답 속도를 저하시키고 데이터 일관성을 떨어뜨릴 수도 있습니다.

대용량 데이터

모든 트랜잭션 메모리가 처리 장치에 캐시되기 때문에 데이터 용량은 상대적으로 낮게 유지되어야 하며, 특히 처리 장치 인스턴스가 추가될수록 더욱 그렇습니다. 따라서 처리 장치의 메모리 부족으로 인한 충돌을 방지하려면 메모리 캐시 크기를 면밀히 관리하는 것이 매우 중요합니다.

데이터 충돌

데이터 충돌은 한 캐시 인스턴스(캐시 A)에서 데이터가 업데이트된 후, 다른 캐시 인스턴스(캐시 B)로 복제되는 과정에서 동일한 데이터가 캐시 B에서도 업데이트될 때 발생합니다. 이는 액티브/액티브 상태로 복제 캐싱을 사용할 때, 즉 여러 처리 장치가 동시에 동일한 데이터를 업데이트할 때 발생할 수 있습니다. 충돌은 일반적으로 캐시의 데이터 업데이트 속도가 복제 지연 시간(RL, 동일한 이름을 가진 각 캐시를 동기화하는 데 걸리는 시간)을 초과할 때 발생합니다. 이 시나리오에서는 캐시 B의 로컬 업데이트가 캐시 A의 기존 데이터로 덮어쓰여지고, 캐시 A의 동일한 데이터는 캐시 B의 업데이트로 덮어쓰여집니다. 이로 인해 각 캐시의 데이터가 불일치하게 됩니다.

데이터 충돌 문제를 설명하기 위해 주문 서비스의 두 서비스 인스턴스(A와 B)가 있다고 가정해 보겠습니다. 각 인스턴스에는 제품 재고의 복제된 캐시(파란색 위젯)가 있습니다. 흐름은 다음과 같습니다.

1. 현재 재고량은 A와 B 각각 500개입니다.
2. 인스턴스 A는 고객으로부터 10개 구매 요청을 받고 업데이트합니다.
   파란색 위젯의 재고 캐시는 490개로 제한됩니다.
3. 인스턴스 A의 데이터가 인스턴스 B로 복제되기 전에 인스턴스 B는 5개 단위의 구매 요청을 받고 파란색 위젯의 재고 캐시를 495개 단위로 업데이트합니다.
4. 인스턴스 A의 업데이트로 인해 복제가 진행되어 인스턴스 B의 캐시가 490개 단위로 업데이트됩니다.
5. 인스턴스 B의 업데이트로 인해 복제가 진행되어 인스턴스 A의 캐시가 495개 단위로 업데이트됩니다.
6. 두 캐시 모두 잘못되었고 동기화되지 않았습니다. 재고는 485개여야 합니다.
   각각의 경우.

데이터 충돌 발생 횟수에 영향을 미치는 요인으로는 동일한 캐시를 보유한 처리 장치 인스턴스 수, 캐시 업데이트 속도, 캐시 크기, 캐싱 제품의 RL(Reliability Level) 등이 있습니다. 이러한 요인들을 기반으로 다음 공식을 사용하여 잠재적인 데이터 충돌 발생 횟수를 확률적으로 예측할 수 있습니다.

$$\text{충돌률} = N * \frac{UR^2}{\text{에스}} * RL$$

여기서 N은 동일한 이름의 캐시를 사용하는 서비스 인스턴스 수를 나타냅니다. UR은 업데이트 속도(밀리초 제곱), S는 캐시 크기(행 수), RL은 캐싱 제품의 복제 지연 시간(밀리초)을 나타냅니다.

이 공식은 주어진 시간 간격(예: 매시간) 내 업데이트를 기반으로 발생할 가능성이 있는 데이터 충돌 비율을 결정하는 데 유용하며, 따라서 이 시스템에서 복제 캐싱을 사용하는 것이 얼마나 실현 가능한지 판단하는 데 도움이 됩니다. 예를 들어, 이 계산에 사용된 요소에 대한 값은 표 16-2 를 참조하십시오.

표 16-2. 기본값

업데이트율(UR):	초당 20회 업데이트
인스턴스 수(N): 5	
캐시 크기(S):	50,000행
복제 지연 시간(RL): 100밀리초	
업데이트:	시간당 72,000개,
충돌률:	시간당 14.4개
백분율:	0.02%

이러한 요소들을 공식에 적용하면 시간당 72,000건의 업데이트가 발생하며, 최고치는 다음과 같습니다.
동일한 데이터에 대한 14개의 업데이트가 충돌할 확률. 낮은 비율을 고려할 때
(0.02%), 복제가 실행 가능한 옵션이 될 것입니다.

강화 학습(RL)의 변형은 데이터 일관성에 상당한 영향을 미칠 수 있습니다. 복제
지연 시간은 네트워크 유형 및 물리적 환경을 포함한 여러 요인에 따라 달라집니다.
처리 장치 간의 거리. RL 값은 계산되어야 하며 다음으로부터 도출되어야 합니다.
실제 생산 환경에서의 측정값이기 때문에 공개되는 경우가 드뭅니다!
완료되었습니다. 이전 예시에서 사용된 100ms 값은 적절한 계획 수치입니다.
실제 RL 값을 알 수 없는 경우. 예를 들어, RL 값을 100ms에서 1ms로 변경하는 경우.
동일한 업데이트 횟수(시간당 72,000회)를 제공하지만 훨씬 낮은 값을 생성합니다.
충돌 발생 확률(시간당 0.1회 충돌). 이 시나리오는 다음과 같습니다.
표 16-3.

표 16-3. 복제 지연 시간에 미치는 영향

업데이트율(UR):	초당 20회 업데이트
인스턴스 수(N): 5	
캐시 크기(S):	50,000행
복제 지연 시간(RL):	1밀리초 (기존 100에서 변경됨)
업데이트:	시간당 72,000
충돌률:	시간당 0.1
백분율:	0.0002%

동일한 이름의 캐시를 포함하는 처리 장치의 수(표시된 대로)
(N에 의한) 또한 가능한 데이터의 수와 직접적인 비례 관계를 갖습니다.
충돌을 방지합니다. 예를 들어 처리 장치 수를 5개에서 10개로 줄이면 충돌이 발생합니다.
2개의 인스턴스를 사용하면 시간당 72,000건의 업데이트 중 데이터 충돌률은 6건에 불과합니다.
시간은 표 16-4 에 나와 있습니다 .

표 16-4. 처리 장치 인스턴스 수에 미치는 영향

업데이트 속도(UR): 초당 20회 업데이트	
인스턴스 수(N): 2 (기존 5에서 변경됨)	
캐시 크기(S):	50,000행
복제 지연 시간(RL): 100밀리초	
업데이트:	시간당 72,000
충돌률:	시간당 5.8
백분율:	0.008%

캐시 크기는 충돌률과 반비례하는 유일한 요소입니다. 캐시 크기가 줄어들면 충돌률은 증가합니다. 이 예에서 캐시 크기를 50,000행에서 10,000행으로 줄이면(다른 모든 조건은 첫 번째 예와 동일하게 유지) 시간당 충돌률이 72회로, 50,000행일 때보다 훨씬 높아집니다(표 16-5 참조).

표 16-5. 캐시 크기에 미치는 영향

업데이트율(UR):	초당 20회 업데이트
인스턴스 수(N): 5	
캐시 크기(S):	10,000행 (기존 50,000행에서 변경됨)
복제 지연 시간(RL): 100밀리초	
업데이트:	시간당 72,000
충돌률:	시간당 72.0
백분율:	0.1%

일반적인 상황에서 대부분의 시스템은 장시간(예: 8시간) 동안 일정한 업데이트 속도를 유지하지 않습니다. 따라서 이 계산을 사용할 때는 시스템 사용량이 가장 많은 시간대의 최대 업데이트 속도를 파악하고 최소, 보통, 최대 충돌률을 계산하는 것이 좋습니다.

통치

우주 기반 아키텍처는 다양한 구성 요소로 이루어져 있어 설계 및 구현이 복잡한 아키텍처 스타일입니다. 특히 내부 메모리 사용과 관련된 문제점(앞서 303페이지의 "일반적인 위험"에서 언급했듯이)을 고려할 때, 메모리 사용량 제어를 포함한 적절한 관리 체계는 성공을 보장하는 데 매우 중요합니다.

이러한 메모리 문제를 해결하기 위해 각 처리 장치 인스턴스가 주기적으로 현재 메모리 사용량을 관찰할 수 있도록 지속적이고 자동화된 관리 적합성 함수를 작성하는 것이 좋습니다. 모든 처리 장치 인스턴스는 동일한 복제 캐시를 사용하므로 적합성 함수는 처리 장치 이름만 보고하면 됩니다. 각 처리 장치의 인스턴스 수를 기록하는 별도의 적합성 함수를 사용하여 처리 장치별 총 메모리 사용량을 계산할 수 있습니다. 그림 16-15는 이러한 연속 적합성 함수의 출력 예시를 보여줍니다.

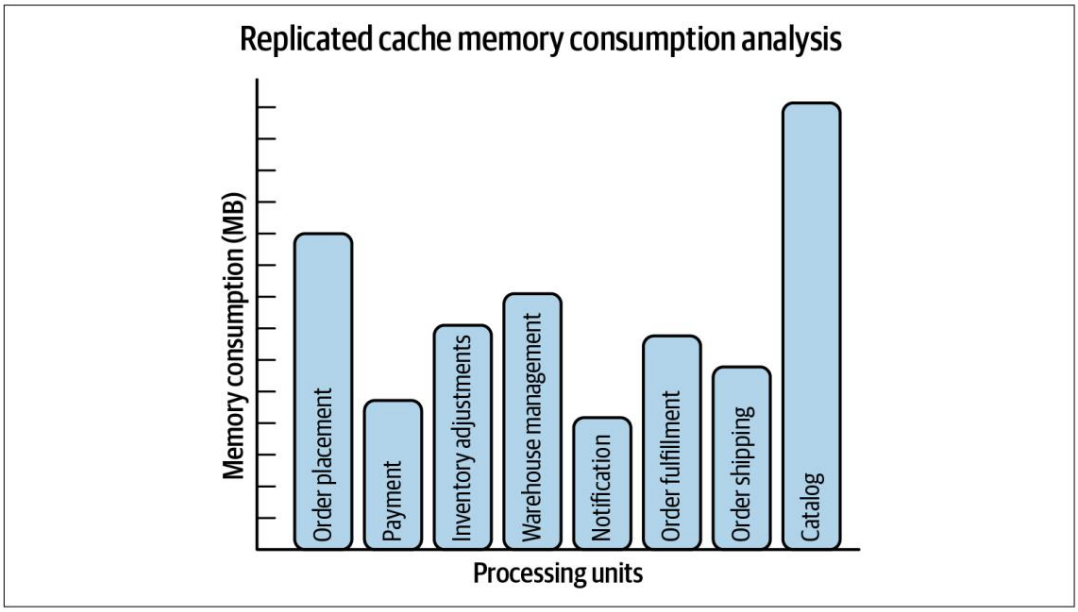

그림 16-15. 메모리 사용량을 추적하는 적합성 함수 예시

데이터 일관성을 유지하기 위한 또 다른 유용한 거버넌스 전략은 동기화 시간, 특히 캐시 업데이트가 해당 데이터베이스에 동기화되는 데 걸리는 시간을 추적하고 측정하는 것입니다. 좋은 방법은 각 처리 장치가 업데이트 요청 ID와 해당 타임스탬프를 스트리밍하고, 각 데이터 기록 장치가 데이터베이스 커밋 후 동일한 요청 ID와 타임스탬프를 스트리밍하도록 하는 것입니다. 그런 다음 이러한 요청 ID를 연결하고 타임스탬프를 빼서 동기화 시간을 계산하는 적합도 함수를 작성합니다. 적합도 함수는 특정 처리 장치와 관련된 시간을 사용하여 원자 수준에서 추적하거나 전체 동기화 시간을 평균하여 전체적으로 추적할 수 있습니다. 이러한 추세를 분석하면 아키텍트는 아키텍처 변경의 영향을 추적할 수 있습니다. 예를 들어 변경 사항이 동기화 시간을 개선하는지 악화시키는지, 그리고 아키텍처가 비즈니스의 동기화 시간 목표를 충족하는지 여부를 확인할 수 있습니다. 그림 16-16은 연속 적합도 함수를 통해 이러한 유형의 거버넌스를 보여줍니다.

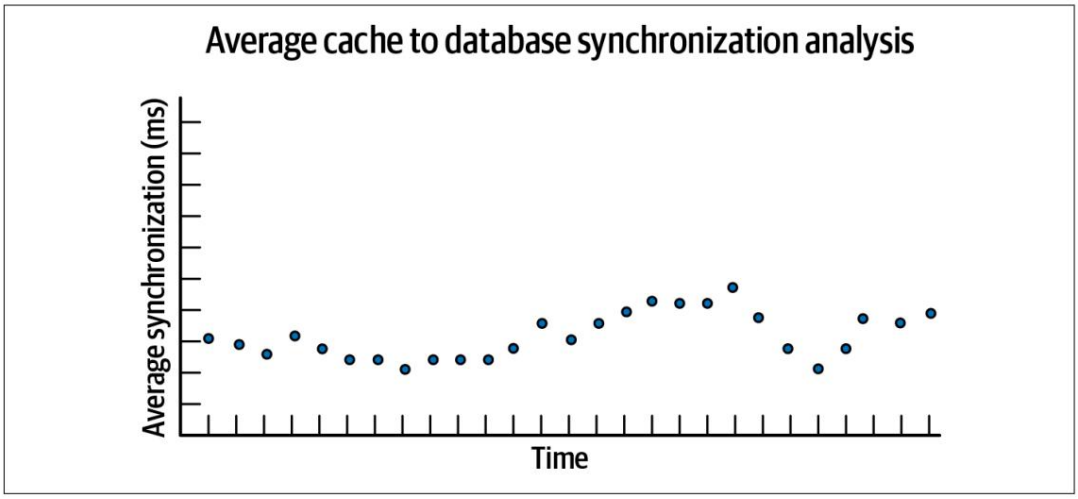

그림 16-16. 집계된 평균 동기화를 추적하기 위한 적합성 함수 예시
타임스

앞서 언급했듯이, 데이터 펌프는 공간 기반 아키텍처에서 역압력 지점 역할을 하고 데이터베이스 쓰기가 캐시 쓰기보다 시간이 더 오래 걸리기 때문에 전체 시스템의 병목 현상을 일으킬 수 있습니다. 이를 방지하기 위해 병목 현상이 발생할 때 이를 추적하고 측정하는 것이 좋습니다. 먼저 병목 현상의 정도를 파악하기 위해 데이터 펌프에서 사용되는 큐의 깊이를 추적하는 적합도 함수를 작성합니다. 병목 현상이 심하면 동기화 시간(따라서 데이터 일관성)이 증가하고, 특히 사용자 동시 접속이 많은 시간대에 데이터 손실 및 데이터 충돌 가능성이 높아집니다. 이전 적합도 함수와 마찬가지로 이 함수는 각 데이터 펌프 큐에 대한 결과를 개별적으로 보고하거나 전체 시스템 평균을 집계할 수 있습니다. 그림 16-17은 일반적인 주문 처리 시스템에서 주문 처리 장치와 해당 데이터 펌프에 대한 병목 현상 분석을 보여줍니다.

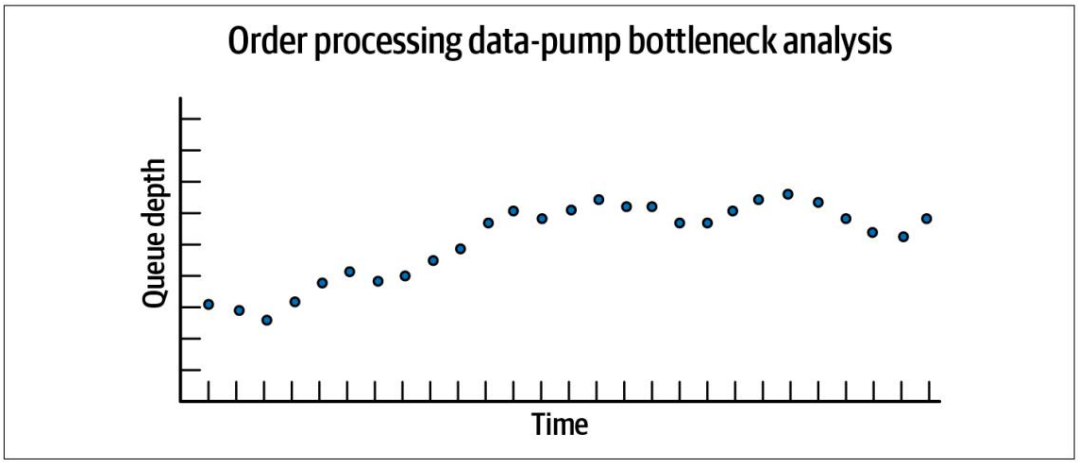

그림 16-17. 데이터 펌프 병목 현상을 추적하기 위한 적합성 함수 예시

우주 기반 아키텍처의 다른 거버넌스 적합성 평가 기능으로는 데이터베이스 읽기 빈도(데이터 판독기에 대한 요청)를 추적하는 것이 있습니다. 이는 확장성, 탄력성 및 응답성에 영향을 미칠 수 있습니다. 또한 적합성 평가 기능을 사용하여 확장성, 탄력성 및 응답성을 측정하는 것이 좋습니다. 이러한 아키텍처적 특성은 우주 기반 아키텍처를 사용하는 주요 이유이므로, 이를 추적하고 측정하는 것이 중요합니다.

팀 토폴로지 고려 사항

공간 기반 아키텍처는 특정 도메인 또는 하위 도메인을 구성하는 수많은 아티팩트 때문에 기술적으로 분할된 아키텍처로 간주되는 경우가 많지만, 기능, 데이터 펌프, 데이터 읽기/쓰기, 백엔드 데이터베이스 관리와 같은 기술 영역별로 정렬된 기술적으로 분할된 팀에서 가장 효과적입니다. 하지만 도메인 영역별로 정렬된 팀(예: 전문화된 교차 기능 팀)에서도 잘 작동할 수 있습니다. 151페이지의 "팀 토폴로지 및 아키텍처" 에 설명된 특정 팀 토폴로지를 공간 기반 아키텍처에 맞출 때 고려해야 할 몇 가지 사항은 다음과 같습니다.

스트림 기반 팀은 시스
: 템 규모에 따라 기술적 분할에 기반한 도메인 기반 변경 사항을 구현하는 데 어려움을 겪을 수 있습니다. 예를 들어, 스트림 기반 변경 사항은 하나 이상의 처리 장치, 데이터 펌프, 데이터 판독기, 데이터 기록기, 캐시 계약 또는 오케스트레이터는 물론 백업 데이터베이스에도 영향을 미칠 수 있습니다. 특히 이러한 구성 요소가 다른 팀과 공유되는 경우, 단일 스트림 기반 팀이 관리하기에는 부담이 클 수 있습니다. 시스템이 크고 복잡할수록 스트림 기반 팀은 공간 기반 아키텍처에서 더욱 비효율적으로 작동할 것입니다.

팀 역량 강화: 이
: 스타일의 구성 요소(데이터 펌프, 데이터 판독기, 데이터 기록기 및 가상화된 미들웨어) 중 일부는 공유되거나 여러 분야에 걸쳐 사용될 수 있으므로 팀 역량을 강화하는 데 적합합니다. 특정 구성 요소(예: 데이터 기록기와 해당 데이터 펌프)에 전담 팀을 구성하면 팀 구성원은 특정 처리 장치의 주요 기능을 담당하는 팀과 관계없이 이러한 구성 요소를 더욱 효율적으로 만드는 방법을 실험하고 찾아낼 수 있습니다.

복잡한 하위 시스템 팀
: 복잡한 시스템 팀은 공간 기반 아키텍처 스타일의 기술적으로 분할된 특성을 활용하여 시스템의 특정 부분(예: 데이터 그리드 또는 데이터 펌프)에 집중할 수 있습니다. 이러한 구성 요소 중 일부는 매우 복잡해질 수 있으므로 복잡한 하위 시스템 팀 토폴로지에 적합합니다. 데이터 충돌(304페이지의 "데이터 충돌" 참조) 및 데이터 기록기에서 발생하는 기타 비동기 데이터 동기화 오류를 처리하는 것은 상당히 복잡합니다.

처리 장치 개발에 집중하는 기능 중심 팀이 신경 쓸 필요가 없는 복잡한 하위 시스템의 훌륭한 예입니다.

플랫폼 팀은 대부
분의 아키텍처 스타일과 마찬가지로, 공간 기반 아키텍처에서 작업하는 개발자가 공통 도구, 서비스, API 및 작업을 활용하여 플랫폼 팀 토폴로지의 이점을 누릴 수 있도록 지원합니다. 특히 아키텍처의 인프라 관련 부분(예: 데이터 펌프 및 가상화된 미들웨어)이 플랫폼 관련으로 간주되는 경우 더욱 그렇습니다.

스타일 특징

그림 16-18 의 특성 평가표에서 별 1개는 해당 아키텍처 특성이 아키텍처에서 제대로 지원되지 않음을 의미하고, 별 5개는 해당 아키텍처 특성이 아키텍처 스타일에서 가장 강력한 기능 중 하나임을 의미합니다. 평가표에 제시된 각 특성에 대한 정의는 4 장 에서 확인할 수 있습니다 .

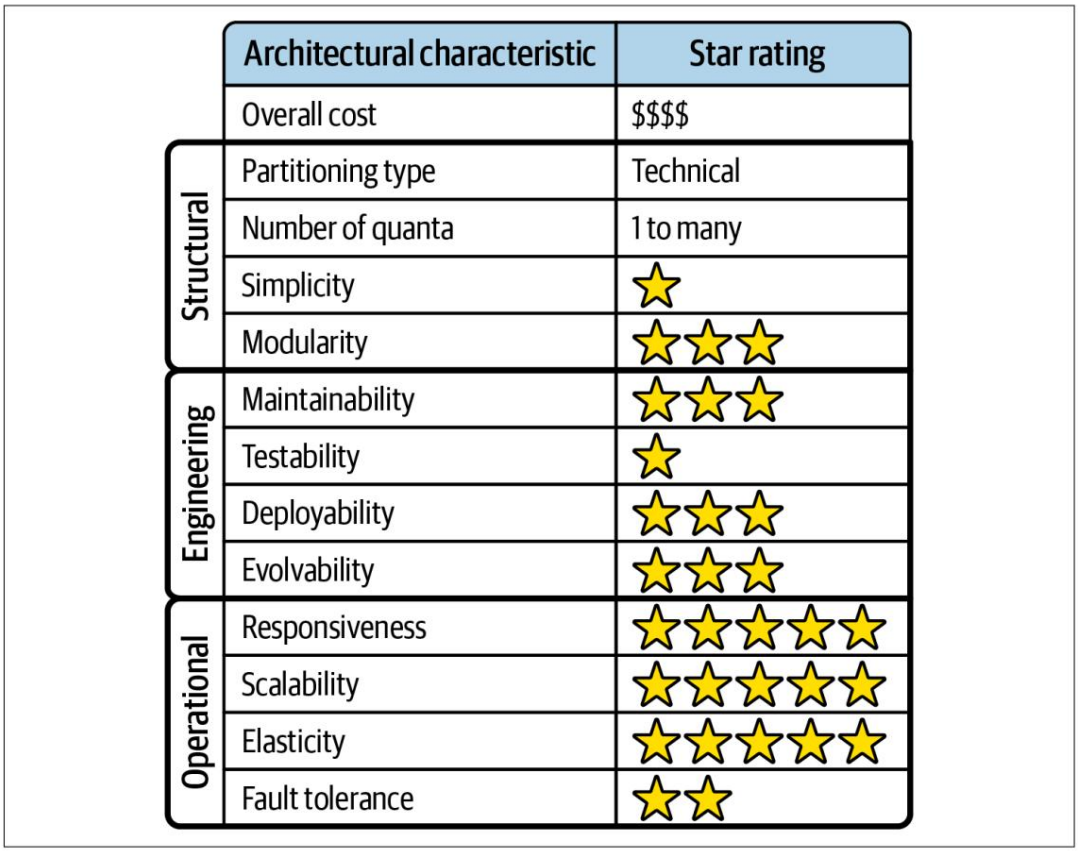

그림 16-18. 우주 기반 아키텍처 특성 평가

공간 기반 아키텍처는 탄력성, 확장성 및 성능을 극대화합니다. 따라서 이러한 모든 특성에 별 5개 만점을 부여했습니다. 이것이 바로 이 아키텍처 스타일의 핵심 특징이자 주요 장점입니다. 인메모리 데이터 캐싱을 활용하고 데이터베이스를 제약 요소에서 제거함으로써, 이 아키텍처 스타일로 구축된 시스템은 수백만 명의 동시 사용자를 처리하는 데 필요한 높은 수준의 탄력성, 확장성 및 성능을 확보할 수 있습니다.

이러한 장점의 대가로 시스템의 전반적인 단순성과 테스트 용이성이 저하됩니다. 공간 기반 아키텍처는 캐싱과 최종 기록 시스템인 기본 데이터 저장소의 최종 일관성 유지를 사용하기 때문에 매우 복잡합니다. 이러한 방식은 구성 요소가 많기 때문에 설계자는 시스템 충돌 시 데이터 손실을 방지하기 위해 특별히 주의해야 합니다(15 장 253페이지 의 "데이터 손실 방지" 참조).

테스트 용이성은 이 방식이 지원하는 높은 수준의 확장성과 탄력성을 시뮬레이션하는 복잡성 때문에 별점 1점을 받았습니다. 수십만 명의 동시 사용자를 최대 부하에서 테스트하는 것은 매우 복잡하고 비용이 많이 드는 작업이므로, 대부분의 대용량 테스트는 실제 극한 부하가 발생하는 운영 환경에서 이루어지며, 이는 정상적인 운영에 상당한 위험을 초래합니다.

비용 또한 고려해야 할 사항입니다. 공간 기반 아키텍처는 전반적인 복잡성, 캐싱 제품 라이선스 비용, 그리고 높은 확장성과 탄력성을 지원하기 위해 클라우드 및 온프레미스 시스템 내에서 필요한 리소스 사용량 때문에 상대적으로 비용이 많이 듭니다.

처리 장치가 별도로 배포되는 방식은 도메인 분할에 적합할 수 있지만, 공간 기반 아키텍처는 기술적으로 분할된 아키텍처로 간주합니다. 왜냐하면 주어진 도메인은 처리 장치, 데이터 펌프, 데이터 판독기 및 기록기, 데이터베이스와 같은 다양한 기술 구성 요소로 표현되기 때문입니다.

공간 기반 아키텍처 내의 양자(quanta) 수는 UI 설계 방식과 처리 장치 간 통신 방식에 따라 달라질 수 있습니다. 처리 장치들이 데이터베이스와 동기적으로 통신하지 않기 때문에 데이터베이스 자체는 양자 방정식의 일부가 아닙니다. 결과적으로 공간 기반 아키텍처 내의 양자는 일반적으로 다양한 UI와 처리 장치 간의 연결을 통해 구분됩니다. 동기적으로 통신하는 처리 장치들(서로 또는 오케스트레이션을 위한 처리 그리드를 통해)은 모두 동일한 아키텍처 양자에 속하게 됩니다.

예시 및 사용 사례

공간 기반 아키텍처는 사용자 또는 요청량이 급증하는 애플리케이션이나 동시 접속 사용자 수가 10,000명을 초과하는 애플리케이션에 적합합니다. 여기서는 공간 기반 아키텍처의 두 가지 활용 사례, 즉 온라인 콘서트 티켓 예매 애플리케이션과 온라인 경매 시스템을 살펴보겠습니다. 두 애플리케이션 모두 높은 수준의 성능, 확장성 및 탄력성을 요구합니다.

콘서트 티켓 예매 시스템은 독특한 문제 영역입

니다. 인기 콘서트가 발표되기 전까지는 동시 접속자 수가 비교적 적지만, 특히 인기 있는 아티스트의 티켓 판매가 시작되면 동시 접속자 수가 수백 명에서 수천 명, 심지어 수만 명에 이르기까지 급증합니다. 모두 좋은 자리를 확보하기 위해 몰려들기 때문입니다! 티켓은 보통 몇 분 만에 매진되므로, 이러한 시스템은 공간 기반 아키텍처가 지원하는 특성을 반드시 필요로 합니다.

이러한 시스템에는 여러 가지 어려움이 있습니다. 첫째, 좌석 선호도와 관계없이 이용 가능한 티켓 총량이 제한되어 있습니다. 따라서 동시 요청이 많기 때문에 좌석 현황을 지속적으로 최대한 빠르게 업데이트해야 합니다. 중앙 데이터베이스에 지속적으로 동기적으로 접근하는 방식은 현실적으로 어려울 것입니다. 일반적인 데이터베이스가 이러한 규모와 업데이트 빈도로 수만 건의 동시 요청을 표준 트랜잭션으로 처리하는 것은 매우 힘들기 때문입니다.

공간 기반 아키텍처는 높은 탄력성이 요구되는 콘서트 티켓팅 시스템에 적합합니다. 배포 관리자는 동시 접속 사용자 수의 갑작스러운 증가를 즉시 감지하고 대량의 티켓 구매 요청을 처리하기 위해 많은 처리 장치를 가동할 수 있습니다. 최적의 경우, 배포 관리자는 티켓 판매 시작 직전에 필요한 수의 처리 장치를 가동하도록 구성하여 사용자 부하가 크게 증가하기 직전에 대기 상태를 유지할 수 있습니다.

온라인 경매 시스템 (eBay와 같

이 경매를 통해 상품에 입찰하는 사이트)은 앞서 설명한 온라인 콘서트 티켓 예매 시스템과 여러 가지 공통점을 가지고 있습니다. 두 시스템 모두 높은 수준의 성능과 유연성을 요구하며, 사용자 및 요청 부하가 예측 불가능하게 급증하는 현상이 발생합니다. 경매가 시작될 때, 얼마나 많은 사람이 참여할지, 각 호가에 대해 얼마나 많은 동시 입찰이 발생할지 예측할 방법이 없습니다.

우주 기반 아키텍처는 부하가 증가함에 따라 여러 처리 장치를 시작하고 부하가 줄어들면 종료할 수 있으므로 이러한 문제 영역에 매우 적합합니다.

경매가 종료되면 더 이상 필요하지 않게 됩니다. 각 경매에 개별 처리 장치를 할당할 수 있으므로 입찰 데이터의 일관성을 보장할 수 있습니다. 또한 데이터 펌프의 비동기적 특성 덕분에 입찰 데이터를 다른 처리(입찰 내역, 입찰 분석 및 감사 등)로 지연 없이 전송할 수 있어 입찰 프로세스의 전반적인 성능이 향상됩니다.

공간 기반 아키텍처는 복잡하지만 매우 강력한 아키텍처 스타일입니다. 캐싱 활용 방식과 데이터베이스에 직접 접근하지 않는다는 특징 덕분에 응답성, 확장성, 탄력성을 극대화하는 유일한 아키텍처 스타일입니다. 따라서 이러한 특정 아키텍처 특성을 극대화해야 하는 상황에서 사용되는 특수한 아키텍처 스타일로 볼 수 있습니다.