게임서버프로그래밍

2019년 2학기

한국산업기술대학교
게임공학부

정내훈

9장 분산 서버 구조

9.1 | 수직 확장과 수평 확장

- 확장성(scalability)

  확장성이란 사용자 수가 늘어나더라도 쉽게 대응할 수 있어야 한다는 의미

  사용자 수가 늘어나도 서버 성능을 유지하려면 보통 다음 두 가지 중 하나를 수행한다.
  ·스케일 업(scale-up) = 수직 확장(서버의 하드웨어를 더 좋은 것으로 교체하여 처리량을 늘림)
  · 스케일 아웃(scale-out) = 수평 확장(서버 대수를 늘려서 더 많은 처리를 하는 것)

  수평 확장이 더 많이 사용되는 방법이지만, 소프트웨어 설계가 더 복잡하다는 대가가 따른다.

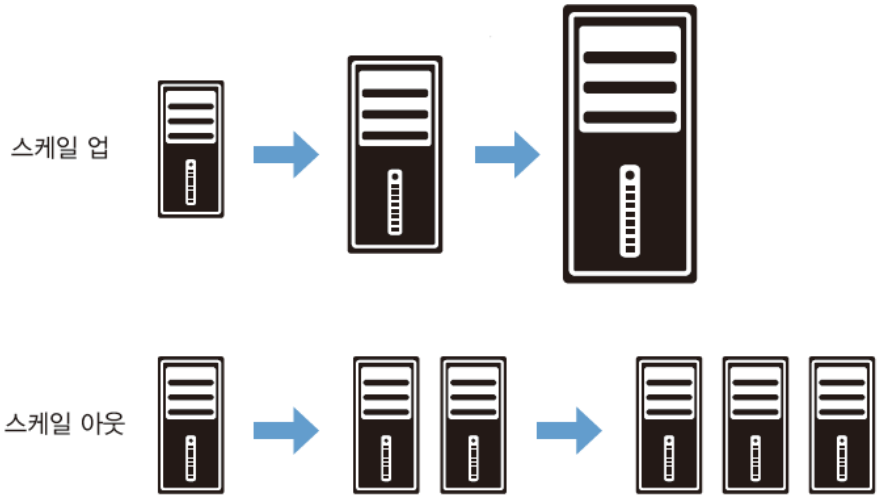


그림 9-1 수직 확장과 수평 확장

9.1 | 수직 확장과 수평 확장

구분	수직 확장	수평 확장
확장 종류	서버 머신의 부품을 업그레이드 혹은 서버 머신 안의 CPU, 램을 증설한다.	서버 머신의 대수를 증설한다.
서버 소프트웨어 설계 비용	낮다.	높다.
확장 비용	높다(기하급수적으로 높아진다).	낮다(선형적으로 높아진다).
과부하 지점	서버 컴퓨터 자체	네트워크 장치
오류 가능성	낮다(로컬 머신 안에서 동기 프로그래밍 방 식으로 작동하므로).	높다(여러 머신에 걸쳐 비동기 프로그래밍 방 식으로 작동하므로).
단위 처리 속도	높다(로컬 컴퓨터의 CPU와 램만 사용).	낮다(여러 서버 컴퓨터 간 메시징이 오가면서 처리하므로).
처리 가능 총량	낮다(서버 컴퓨터 한 대 성능만 사용하므로).	높다(여러 서버 컴퓨터로 부하가 분산되므로).

표 9-1 수직 확장과 수평 확장 비교

9.2 | 서버 분산이 없다면?

그림 9-2 분산 처리를 하지 않는 서버

- 동시접속자 수가 무제한으로 증가했을 때

1. 서버로 보낸 메시지에 대한 처리 응답이 늦게 도착한다.
2. 서버 접속 과정이 매우 오래 걸린다.
3. 서버와 연결이 돌발 해제된다. 드물지만 TCP 재전송 타임아웃(retransmission timeout) 때문에 이러한 현상이 발생하기도 한다. 또는 사용자 정의 킵얼라이브(keep-alive) 메시징 타임아웃 때문에 발생하기도 한다.
4. 서버 접속 자체가 실패하여 타임아웃 현상이 발생한다.

서버에서는

1. CPU 사용량이 증가한다.
2. 클라이언트에서 메시지를 받는 속도보다 메시지를 처리하는 속도가 느릴 때 발생한다. 이때 서버의 메모리 사용량이 증가한다.
3. 클라이언트에 보낼 메시지의 발생 속도보다 실제로 메시지를 보내는 속도가 느릴 때도 발생한다. 이때도 서버의 메모리 사용량이 증가한다.
4. 즉, CPU 과부하는 램 사용량 증가로 이어진다.

9.2 | 서버 분산이 없다면?

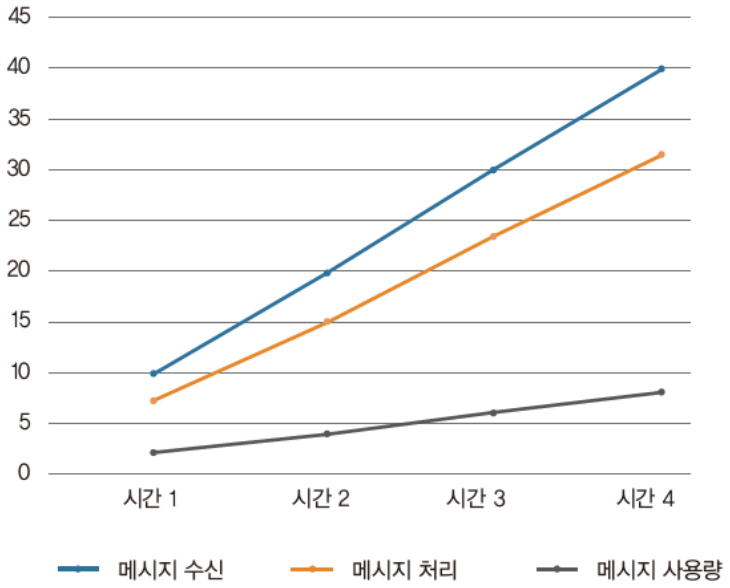


그림 9-3 과부하에 걸린 서버가 시간이 지남에 따라 발생하는 현상

```
int* a = malloc(1000);
// a가 null이면 접근 위반 오류를 발생한다.
a[0] = 1;
// 메모리 공간이 모자라면 bad_alloc 예외를 발생한다.
MyClass* b = new MyClass();
```

64비트(x64)에서는 서버의 물리적 메모리보다 더 많은 양의 메모리를 할당하면서 대량의 메모리스와핑(memory swapping)이 발생한다.

이 때문에 프로그램 실행 속도가 급락하면서 메모리 할당량이 더욱 증가하는 악순환도 발생한다.

단일 서버에 물려 있는 네트워크 기기에 과부하가 걸리면

- 라우터 과부하로 패 킷 유실(p acket drop)이 발생한다.
- TCP 재전송 타임아웃으로 TCP 연결 해제(disconnection)가 발생한다.
- TCP 소켓에서 E CONNABORTED 오류가 발생한다.

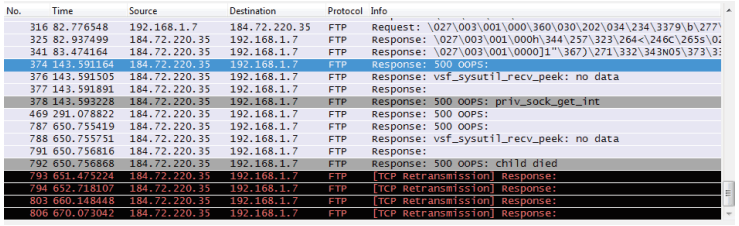


그림 9-4 전송 실패 기록

9.3 | 고전적인 서버 분산 방법

- 서버 클러스터(서버의 집합)의 구성

  인증 서버 한 대

  채널 서버 네 대

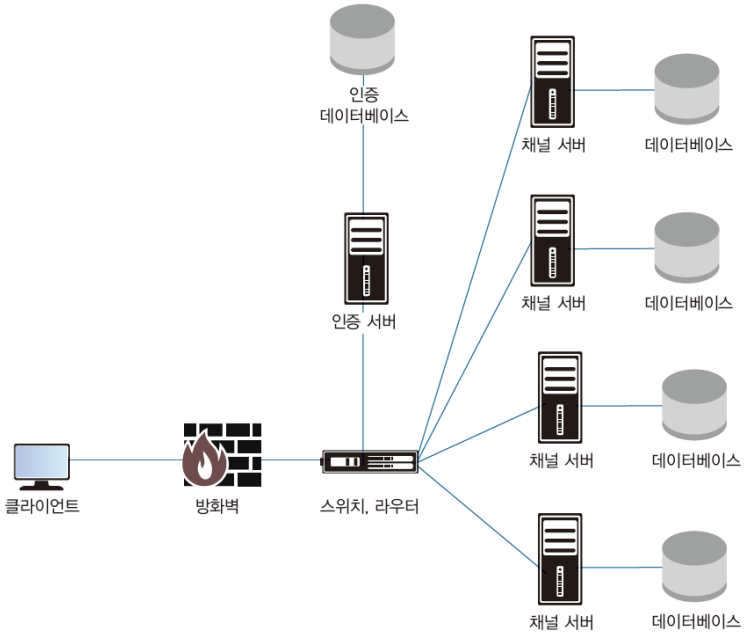


그림 9-5 분산 게임 서버 방식

- 게임 클라이언트와 서버의 상호작용

1. 게임 클라이언트는 인증 서버에 접속한 후 ID와 비밀번호로 로그인 처리를 한다.
2. 로그인에 성공한 게임 클라이언트에서 게임 사용자는 게임 플레이를 할 수 있는 채널 서버를 선택.
3. 게임 클라이언트는 사용자가 선택한 채널 서버에 [illegible] 게임 플레이를 시작한다.

그림 9-6 <월드 오브 워크래프트> 게임에서는 로그인 후 채널 서버를 선택해야 함

9.3 | 고전적인 서버 분산 방법

- 간단한 분산 서버 구성의 문제

1.같은 계정이라도 플레이어 정보가 서로 다른 채널 서버에 있으므로 플레이어가 열심히 키워놓은 캐릭터를 다른 채널 서버에서
쉽게 사용할 수 없다.
2. 플레이어는 자기가 플레이했던 채널 서버에서만 계속 게임을 해야 한다. 따라서 다양한 플레이어 간에 상호 작용을 할 수 있는 기회가 줄어든다.
3. 모바일 게임과 글로벌 서비스 게임에서는 채널 서버를 선택하는 과정이 없을 때가 많다.

9.4 | 논리적 단일 서버 분산

- 게임 서버 분산의 절차

  1. 단일 서버 기준에서 과부하가 걸리는 지점을 분석해서 파악한다.
  2. 과부하가 걸리는 지점을 앞으로 소개할 여러 가지 분산 처리 방식으로 분산한다.

그림 9-7 논리적 & 물리적 단일 서버

라우터, 스위치: 1GB/초
방화벽: 500MB/초
CPU: 4코어 * 3GHz
스토리지: 100MB/초(SSD), 10MB/초(HDD)

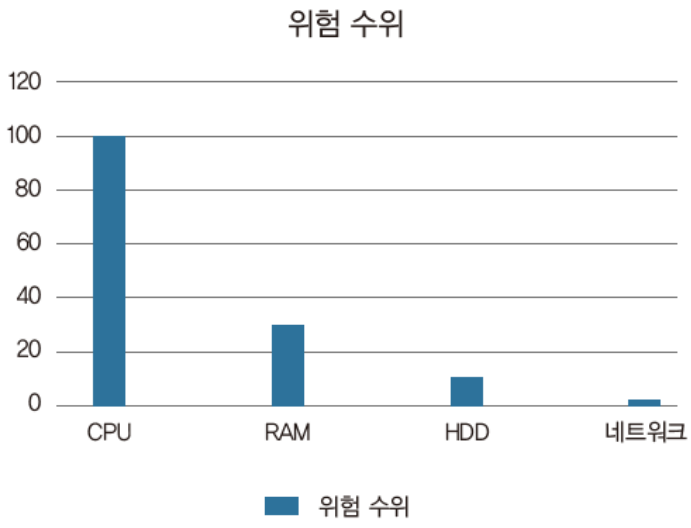


그림 9-8 단일 게임 서버의 성능 분석 결과

9.4 | 논리적 단일 서버 분산

- 성능 분석 도구

  게임 서버 안에서 CPU 사용량이 매우 높은 경우, CPU 안에서도 어느 부분이 사용량이 높은지 따로 체크할 때 사용

  윈도 서버에서 개발하는 경우 비주얼 스튜디오 코드 프로파일러(Visual Studio Code Prof iler)나 윈도 성능 도구(Windows Performance Toolkit)를 사용,리눅스에는 perf가 있다.

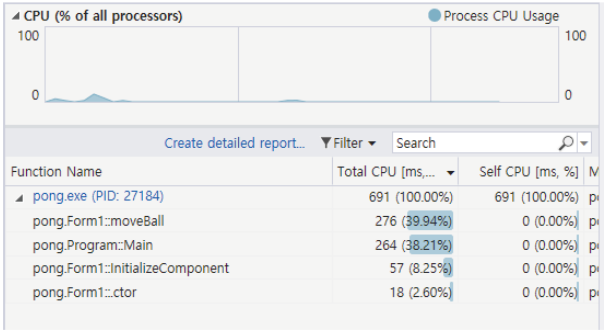


그림 9-9 비주얼 스튜디오의 코드 프로파일러

- 분산 단위

  데이터 단위 분산

  기능 단위 분산

- 게임 로직의 분산 처리 방식

  동기 분산 처리

  비동기 분산 처리

  데이터 복제 및 로컬 처리

분산 처리 유형은 이들의 2×3 조합.

9.5 | 데이터 분산 vs 기능적 분산

- 데이터 분산

  한 머신이 처리해야 하는 데이터를 같은 역할을 하는 여러 머신이 나누어서 처리하는 것.

  각 서버는 모두 똑같은 종류의 일을 한다.

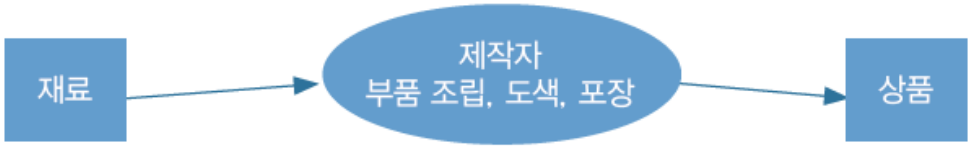


그림 9-10 시계를 만드는 가게의 제작자가 하는 일

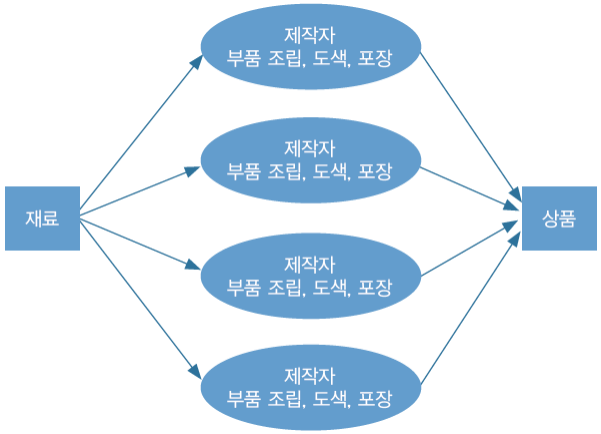


그림 9-11 시계 공장의 기술자들은 서로 같은 일을 함

9.5 | 데이터 분산 vs 기능적 분산

- 기능적 분산

  한 머신이 처리해야 하는 데이터의 처리 단계를 세분화해서 여러 머신이 나누어 처리한다.

  데이터 분산과 기능적 분산을 혼용할 수도 있다.

그림 9-12 시계 제작의 단계를 여러 사람이 서로 다른 역할로 나누어서 함

이러한 분산 처리 원리는 게임 서버뿐만 아니라 데이터베이스에서도 비슷하다. 데이터 단위 분산이란 테이블 1개를 테이블 안의 키 필드 단위로 분배하는 것을 의미한다.(파티셔닝)

기능 단위 분산이라면 서로 다른 테이블을 서로 다른 서버에 배치하는 것을 의미한다.

9.6 | 로직 처리의 분산 방식들

- 로직 처리의 분산 방식

  동기 분산 처리

  비동기 분산 처리

  데이터 복제 및 로컬 처리

- 상호 작용 시나리오(몬스터 사냥)
  1. 플레이어가 가진 총알 1개를 소모한다.
  2. 몬스터의 체력을 깎는다.
  3. 몬스터의 체력이 바닥난다.
  4. 몬스터가 사망한다(10초 후 소멸).
  5. 플레이어는 아이템을 획득한다.

```
Player_Attack(player, monster)
{
        player.bullet--;
        monster.hitPoint -= 10;
        if (monster.hitPoint < 0)
        {
                player.item.Add(gold, 30);
                DeleteEntity(monster,
10sec);
        }
}
```

그림 9-13
몬스터 사냥 로직은 서버 1에서만 사용

9.6 | 로직 처리의 분산 방식들

- 9.6.1 동기 분산 처리

  어떤 연산을 다른 서버에 던져 놓고 그 결과가 올 때까지 대기한다.

  대기할 뿐만 아니라 그 연산과 관계된 데이터가 도중에 변경되지 않게 잠금(lock)을 해야 한다.

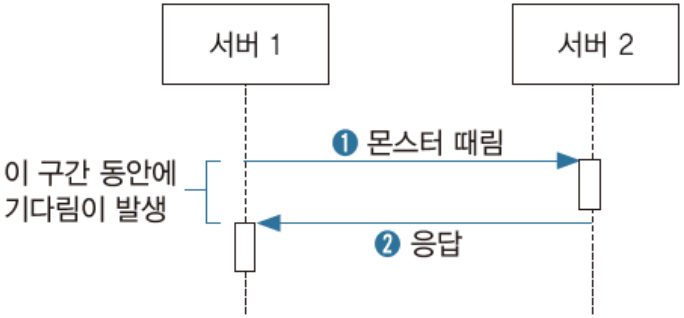


그림 9-14 서버 1이 서버 2가 일을 끝낼 때까지 락을 걸고 있는 상황

```
Player_Attack(player, monster)
{
        lock(player)
        {
                player.bullet--;
                e = otherServer.DamageCharacter(
                        player.id, monster.id, 10);
                waitForResult(e);
                if (e.hitPoint < 0)
```

9.6 | 로직 처리의 분산 방식들


```
            {
                player.item.Add(gold, 30);
            }
        }

        DamageCharacter(attacker, character, damage)
        {
            character.hitPoint -= damage;
            result.hitPoint = character.hitPoint;
            Reply(result);
        }
    }
```

'동기식 명령 처리법

9.6 | 로직 처리의 분산 방식들

- 분산 락 기법을 이용하여 원격지에 있는 데이터를 액세스하는 방식(동기식 데이터 변경법)

```
Player_Attack(player, monster)
{
    lock(player)
    {
        player.bullet--;
        otherServer.RemoteLock(monster);                    //
        m = otherServer.RemoteGet(monster.id);              //
        m.hitPoint -= damage;
        if (m.hitPoint < 0)
        {
            player.item.Add(gold, 30);
            otherServer.RemoteDelete(monster.id);  //3
        }
        else
        {
            otherServer.RemoteSet(m);                       //3
        }
        otherServer.RemoteUnlock(m);                        //4

    }
}
```

1 몬스터 정보를 가진 서버 2에 "몬스터 정보를 액세스하기 위해서 분산 락을 걸겠다."라고 요청하고 응답을 받는다.

2 서버 2에서 몬스터 정보를 얻어 온다.

3 서버 2에 몬스터 정보를 업데이트한다.

4 서버 2에 분산 락 해제를 요청하고 응답을 받는다.

9.6 | 로직 처리의 분산 방식들

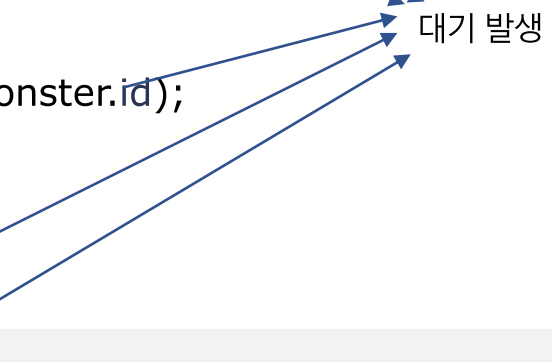
'동기식 데이터 변경법'에서는
메시지 왕복이 총 세 번 오가므로
최소 120마이크로초가 걸린다.

9.6 | 로직 처리의 분산 방식들

```
Player_Attack(player, monster)
{
    lock(player)
    {
        player.bullet--;
        otherServer.RemoteLock(monster);
        m = otherServer.RemoteGet(monster.id);
        m.hitPoint -= damage;
        if (m.hitPoint < 0)
        {
            player.item.Add(gold, 30);
            otherServer.RemoteDelete(monster.id);
        }
        else
        {
            otherServer.RemoteSet(m);
        }
        otherServer.RemoteUnlock(m);

    }
}
```

❶



서버 1에서 플레이어 정보를 보호하는
임계 영역이나 뮤텍스가 지나치게 광범위하게 보호하는 경우,
서버 1에서 처리해야 할 다른 일이 40~120마이크로초만큼
시간이 지연될 가능성이 높아진다.

9.6 | 로직 처리의 분산 방식들

- 암달의 법칙(암달의 저주)

  병렬 처리할 수 있는 장치에서 병렬로 처리하지 못하는 시간에 비례하여 병렬 효과가 급감하는 현상.

  따라서 동기 분산 처리의 잘못된 사용은분산 서버의 효과를 떨어뜨릴 수 있다.

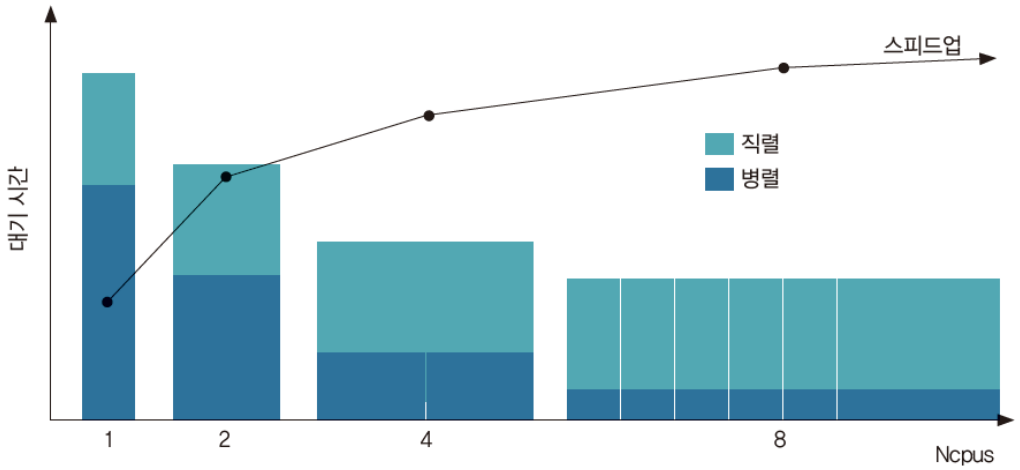


그림 9-15
병렬로 처리하지 못하는 시간(붉은색)의 비중이 클수록 병렬 처리 효과는 크게 감소

9.6 | 로직 처리의 분산 방식들

- 9.6.2 비동기 분산 처리

1. 서버 1은 어떤 연산 명령을 다른 서버 2에 송신한다.
2. 서버 1은 서버 2의 명령 처리 결과를 기다리지 않는다. 그리고 일방적으로 자기가 해야 하는 다음 일을 시작한다.

비동기 I/O와 비슷한 처리 방식

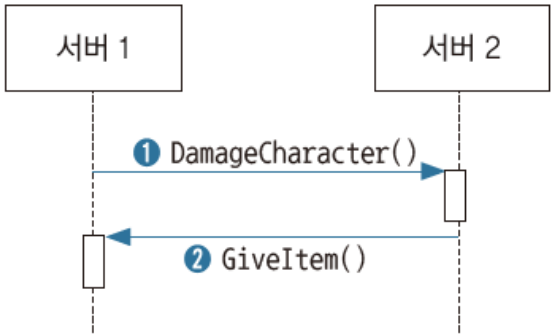


그림 9-16 비동기 분산 처리 방식

서버 1은 플레이어 정보를 가졌고, 서버 2는 몬스터 정보를 가졌다고 가정

❶ 서버 1은 서버 2에 "몬스터에 대미지를 주었다."라고 통보한다.

❷ 서버 2는 서버 1에 "플레이어에게 아이템을 주었다."라고 통보한다.

9.6 | 로직 처리의 분산 방식들

- 코드 레벨

```
Player_Attack(player, monster)
{
	player.bullet--;
	otherServer.DamageCharacter(player.id, monster.id, 10);		//❶
}

DamageCharacter(callerServer, attacker, character, damage)		//❷
{
	character.hitPoint -= damage;
	if (character.hitPoint < 0)
	{
		callerServer.GiveItem(attacker.id, gold, 30);		//❶
		DeleteEntity(character, 10sec);
	}
}

GiveItem(character, item, amount)		//❷
{
	character.item.Add(item, amount);
}
```

9.6 | 로직 처리의 분산 방식들

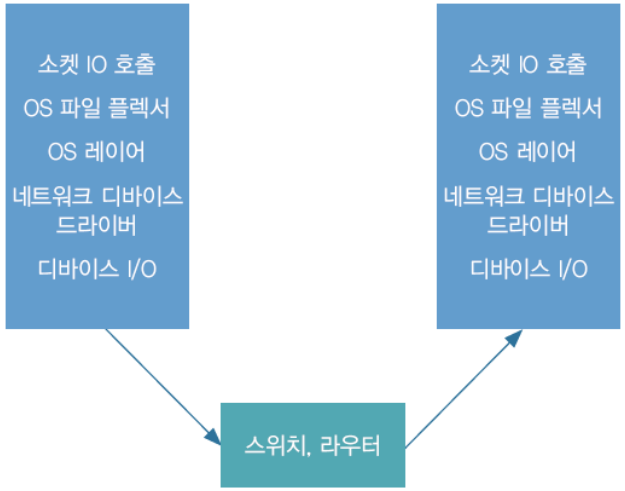
장점 :잠금으로 인한 병목이 없으므로 각 서버가 가진 성능을 최대한으로 활용할 수 있다.

단점 : 모든 로직을 이 방식으로 구현하기 어렵거나 불가능하다. (프로그램이 복잡해진다.)

이 방식으로 프로그래밍을 할 때는 요청에 대한 응답을 기다리는 과정이 없기 때문에 반환 값을 주고받을 수 없다. 그 대신에 다음 방식으로 구현해야 한다.(서로 일방적인 통보를 주고받아야 한다.)

1. 한쪽에서 저쪽으로 명령을 보낸다.
2. 저쪽에서는 이쪽으로 명령을 또 보낸다.

- 동기 분산 처리와 비동기 분산 처리 모두에서 발생하는 단점



정작 중요한 핵심 처리는 기계어 명령어 수십~수백 개인데, 분산된 서버 간 대화에 기계어 명령어가 수천 개 사용된다.

이처럼 과도하게 서버간 분산 처리를 하면 쓸데없이 비효율적인 상황이 발생할 수 있음을 알 수 있다.

그림 9-17 한쪽 서버에서 다른 쪽 서버로 메시지를 보낼 때

9.6 | 로직 처리의 분산 방식들

- 9.6.3 데이터 복제(Proxy)에 기반을 둔 로컬 처리

서버 1에서 몬스터 사냥에 대한 처리를 하면, 이에 관련된 플레이어와 몬스터 정보가 변경된다.

변경 사항은 서버 2에 전송되고, 서버 2는 이를 받아 자기 자신이 가진 플레이어와 몬스터 정보를 변경한다.

=> 효과 : 상태를 READ만 할 경우 네트워크 부하 ZERO!!!

```
Player_Attack(player, monster)
{
        player.bullet--;
        monster.hitPoint -= 10;
        if (monster.hitPoint < 0)
        {
                player.item.Add(gold, 30);
                DeleteEntity(monster, 10sec);
        }
}
```

사실상 분산 처리를 하기 전 코드와 동일하고 코드도 단순하다.

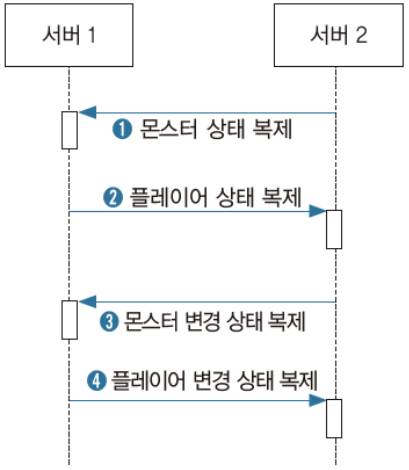


그림 9-18 서버 1과 서버 2의 데이터 복제

각 서버 프로세스는 데이터 원본이나 사본을 가진다.

이 전제하에 어떤 연산을 할 때는 서버 프로세스 안의 원본(자기 서버에서 연산되는 데이터)과 사본(다른 서버에서 받은 데이터)을 가지고 연산을 수행한다.

서버가 가진 데이터를 모두 신뢰할 경우
데이터 불일치로 잘못된 연산이 발생할 수 있는데,
이를 하이젠버그 (Heisenbug)*라고 한다.

9.7 | 데이터 응집도

- 응집도(coherency 또는 locality)

  특정 영역 안에 얼마나 많은 데이터가 관련되고 뭉쳐 있는지 의미.

  어떤 데이터가 있을 때 그 데이터 와 자주 상관되는(혹은 거의 항상) 다른 데이터가 얼마나 많은지 의미.

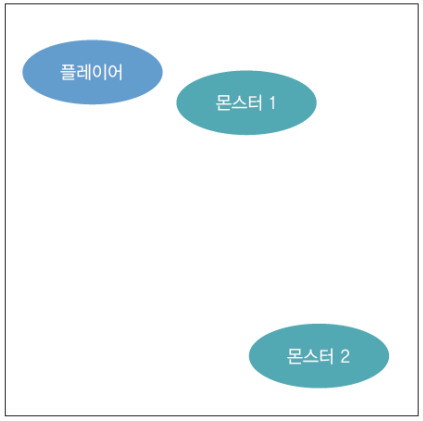


그림 9-19
평면 월드에 플레이어 하나와 몬스터 둘

플레이어와 몬스터 1을 담당하는 서버가 같은 서버라면, 플레이어와 몬스터 1의 상호 작용을 처리하는 데 서버 간 통신은 전혀 필요하지 않다.

플레이어와 몬스터 2를 담당하는 서버가 다른 서버라면, 플레이어와 몬스터 2의 상호 작용을 처리하는 데 서버 간 통신이 발생할 것이다. 게임 기획 내용상 몬스터 2는 플레이어와 상호 작용할 일이 전혀 없다면, 분산 처리 프로그래밍을 하지 않으면 된다.

플레이어와 몬스터 1을 담당하는 서버는 같은 서버라야 좋다.

몬스터 2를 담당하는 서버는 같은 서버일 필요가 없다.

플레이어와 몬스터 1 간 응집도는 높다.
플레이어와 몬스터 2 간 응집도는 낮다.

9.7 | 데이터 응집도

- 데이터 응집도를 무시한 극단적인 사례

  좁은 지역에 플레이어 캐릭터와 몬스터 캐릭터가 뒤섞여 있고 각 캐릭터를 담당하는 서버가 서로 다르다.

  이러한 상황에서 여기 있는 모든 플레이어가 자기 주변에 광역으로 효과를 줄 수 있는 행동을 취한다고 가정.

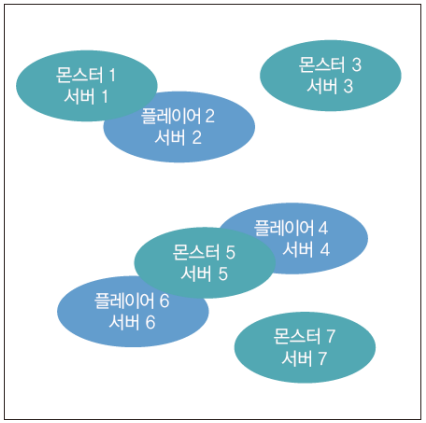


그림 9-20 응집도를 무시한 극단적인 사례

데이터 간 응집도를 고려한다면, 좁은 지역에 많은 캐릭터가 있을 경우 서버 한 대가 이것을 모두 처리하게 하는 것이 낫다.

캐릭터의 지리적 위치를 응집도를 기준으로 하여 가까운 거리에 있는 캐릭터끼리는 같은 서버에 둔다.

실력이 비슷한 플레이어끼리 자주 상호 작용을 한다.

실력 차이가 많이 나는 플레이어끼리는 상호 작용을 하지 않는다.

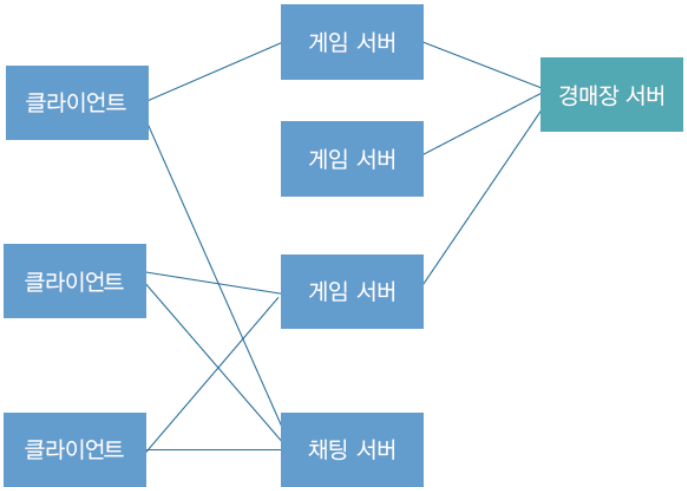
데이터 응집도의 기준을
플레이어 실력으로 잡을 수 있다.

9.8 | 기능적 분산 처리

- 기능적 분산 처리 혹은 수직 분산 처리

  수직 분산 처리는 데이터 단위 분산 처리, 즉 수평 분산 처리를 할 수 없을 때 선택할 수 있는 대안으로, 대표적인 예가 경매장을 담당하는 처리이다.

  게임 서버에서 경매장을 담당하는 처리를 분리하고, 경매장 처리만 담당하는 서버를 개발한다.



장점 : 구현이 쉽다. (통신 부분만 추가하면 된다.)

단점 : 적절한 부하 분산이 어렵다. (게임 서버와 경매장 서버는 여유가 있더라도 상대방의 작업을 대신할 수 없다.)

확장이 어렵다. (기능적 분산이 효과가 있을 만한 모듈들이 별로 없다.)

그림 9-22 경매장 처리만 담당하는 서버와 나머지 서버

기능적 분산이 가능하고 성능이 나온다면 하지 않을 이유는 없다. 분산 대상을 찾기 힘들 뿐이다. (예: AI서버, 인던서버, 경매서버, 채팅서버, DB서버, Query Server...)

9.9 | 분산 처리를 엄선해야 하는 이유

분산 처리 프로그램은 디버깅이 까다롭다. 여러 프로그램에 원격 디버깅을 하거나 로 그 출력을 보고 번거로운 원인 분석을 해야 한다.

지나친 분산 처리는 이 운영체제가 해야 하는 일을 불필요하게 증가시킨다.

클라우드 서버 환경에서는 클라우드 서버 인스턴스 간에 통신 회선의 신뢰성도 문제가 될 수 있다.

분산 처리는 꼭 해야 하는 이유를 설명할 수 없다면 피하는 것이 좋다.

콘텐츠 업데이트나 시스템 업데이트가 계속 있어야 하는 게임 서버의 특성상 서버를 만드는 것도 중요하지만,안정적인 서비스가 더 중요하기 때문.

9.10 | 분산 처리 전략

성능 분석을 하여 분산 처리가 필요한 지점을 엄선한다.

데이터 응집력을 확인한다. 다룰 데이터 간 상호 작용이 높은 것은 분산하지 말고, 다룰 데이터 간 상호 작용이 매우 적은 것들만 골라서 분산한다. 즉, 응집력이 높은 데이터를 구별하는 기준부터 찾아야 한다.

수평 분산 처리 방식은 다음 세 가지 중에서 선택한다.

- 동기 분산 처리
- 비동기 분산 처리
- 데이터 동기화에 기반을 둔 로컬 처리

어쨌건 수평 분산 처리 자체는 구현과 디버깅이 까다롭고 불필요한 과부하를 일으킨다.
불필요한 분산 처리라고 생각되는 부분은 피할 수 있으면 피하는 것이 좋다.

9.12 | 고가용성

- 고가용성(High Availability, HA)

  사용자가 항상 서비스를 이용할 수 있게 하는 것

  서버가 고장 나거나 껐다 켜는 상황이 발생함에도 서버 역할을 지속하는 것

- 장애극복

  서버 클러스터에 있는 서버 중 하나가 죽었을 때, 다른 서버가 죽은 서버를 대신해서 일을 하고 그동안 죽은 서버가 다시 살아나는 것

  극복을 위해서 필요 이상의 서버를 두는 것을 이중화 혹은 다중화라고 한다.

9.12 | 고가용성

- 고가용성을 위한 서버 구성 패턴

  액티브-패시브(active-passive) 패턴 : 마스터-슬레이브(master-slave) 패턴이라고도 하며, 액티브 서버와 패시브 서버를 가짐.
  서버 한 대만 액티브 역할을 수행하고 나머지 서버는 패시브 모드로 대기하다가 액티브 서버가 죽으면 패시브가 그 역할을 대행한다.
  패시브 서버는 그저 백업 역할만 할 뿐 다른 하는 일이 없는 서버 자원의 낭비가 생긴다.

그림 9-24 액티브-패시브 패턴

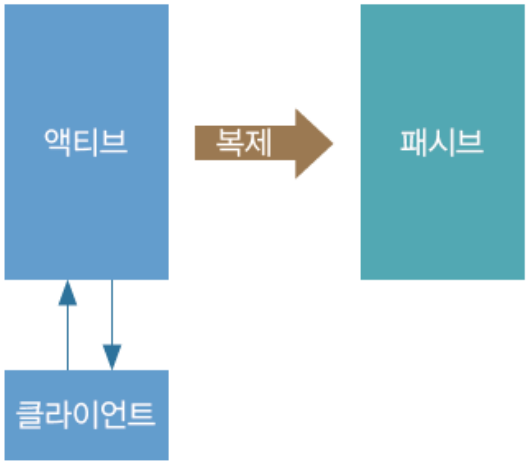


그림 9-25 패시브 서버는 필요할 때 액티브 서버의 데이터를 복제

미러링(Mirroring)이 더 표준적인 용어이다.
Master/Slave는 다른 의미로 많이 사용된다.

9.12 | 고가용성

- 고가용성을 위한 서버 구성 패턴

  액티브-액티브(active-active)패턴 : 마스터-슬레이브(master-slave) 패턴이라고도 하며, 액티브 서버와 패시브 서버를 가짐.

  서버 두 대가 클라이언트 측 요청을 분담하여 처리하고, 필요할 때면두 서버는 각자 가진 데이터를 상대방에게 전송한다.

  액세스된 데이터 X가 정말로 최신 정보인지 장담할 수 없는 스테일 문제(stale problem)가 발생하고, 이를 해결하려면 두 액티브 서버 사이에 메모리 저장소 역할을 담당하는 서버(공유 메모리 서버)를 두면 된다.

  메모리 저장소 서버가 죽어 버리면 고가용성을 지킬 수 없으므로 이중화를 해야 한다.

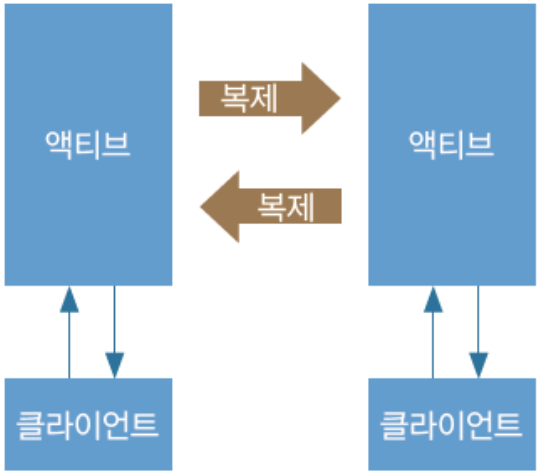


그림 9-26
액티브-액티브 패턴, 양방향 동기화

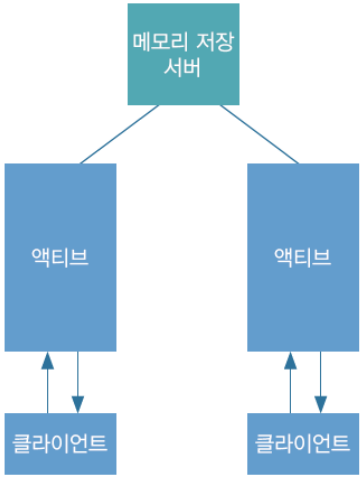


그림 9-27
데이터 스테일 문제의 해결책

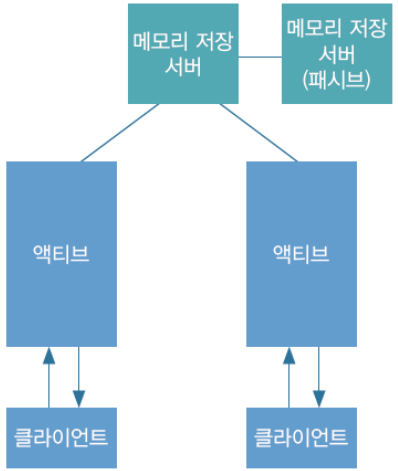


그림 9-28
고가용성을 위한
메모리 저장소 서버의 이중화

9.13 | 데이터베이스의 분산

- 파티셔닝

  더 많은 사용자를 처리하고자 데이터베이스가 수평 확장을 할 때는 갖고 있는 레코드를 서로 다른 데이터베이스에 나누어 놓는다.

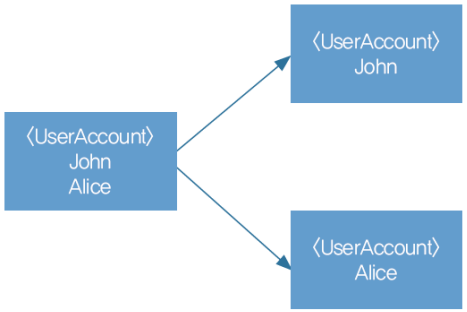


그림 9-29 데이터베이스의 수평 파티셔닝

- 수직 파티셔닝

  한 테이블을 수직으로 잘라서, 즉 레코드 일부 필드를 다른 테이블로 나누어 놓고 그것을 다른 데이터베이스 서버에 둔다.

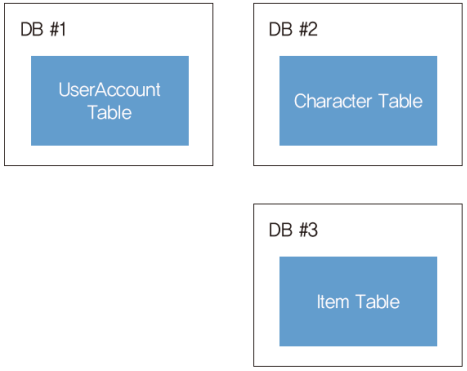


그림 9-30 데이터베이스의 수직 파티셔닝

9.13 | 데이터베이스의 분산

- 이중화

  같은 내용의 레코드를 서버 두 대 이상 에 저장.

  이렇게 하면 불필요한 자원 낭비로 보일 수 있지만, 그 대가로 고가용성을 얻는다.

  특히 데이터베이스는 게임 서버보다 고가용성이 더 요구되기도 한다.

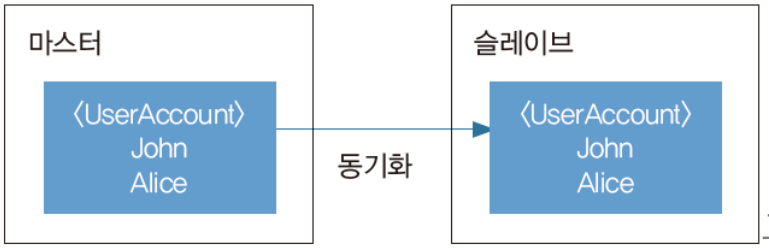


그림 9-31 데이터베이스 이중화

고가용성을 위해 데이터베이스를 이중화할 때는 액티브-패시브와 유사하게 마스터-슬레이브 모델 형태도 자주 사용된다.

게임 서버가 데이터베이스를 액세스할 때는 마스터에 하되, 마스터가 죽어 있으면 슬레이브에 하고, 슬레이브가 둘 이상이면 슬레이브 2, 슬레이브 3...에 액세스한다.

9.14 | 요약

분산 서버: 서버 한 대가 처리할 수 없는 양을 여러 대가 나누어서 처리하는 것.
서버 클러스터: 서버 여러 대의 집합.
수평 확장: 서버 개수를 늘려서 총 처리량을 늘리는 것으로 스케일 아웃과 의미가 같다.
샤드: 거대한 데이터를 여러 서버 기기로 분산한 후 데이터 일부를 지닌 각 서버 기기를 지칭한다.
로드 밸런싱: 한쪽에 과부하가 몰리는 것을 다른 서버와 분배하는 것.
고가용성: 서버 하드웨어나 소프트웨어가 죽더라도 사용자 입장에서 계속해서 서비스를 이용할 수 있는 것.