게임서버프로그래밍

2019년 2학기

한국산업기술대학교
게임공학부

정내훈

10장 분산 서버 구조 사례

1 | 로그온 처리의 분산
2 | 데이터베이스의 수평 확장
3 | 매치메이킹의 분산 처리
4 | 몬스터 NPC 처리의 분산 처리
5 | 플레이어 간 상호 작용 분산 처리
6 | 로그 및 통계 분석의 분산 처리
7 | 게임 장르별 분산 서버 형태
8 | 요약 및 결론

10.1 | 로그온 처리의 분산

- 로그온 처리

1. 클라이언트는 서버에 ID와 비밀번호를 보낸다. 물론 암호화해서 보낸다.
2. 로그온 담당 서버(인증 서버)에서 이 메시지를 받고 메시지를 복호화한다.
3. 서버는 복호화된 메시지에서 ID와 비밀번호를 구한다. 그리고 DB에 질의를 던지고 결과를 기다린다.
4. DB에서 질의를 수행한 후 응답을 서버에 회신한다.
5. 서버는 이를 받아 로그인 성공 혹은 실패를 판단한다. 그리고 로그온 결과를 클라이언트에 보낸다.

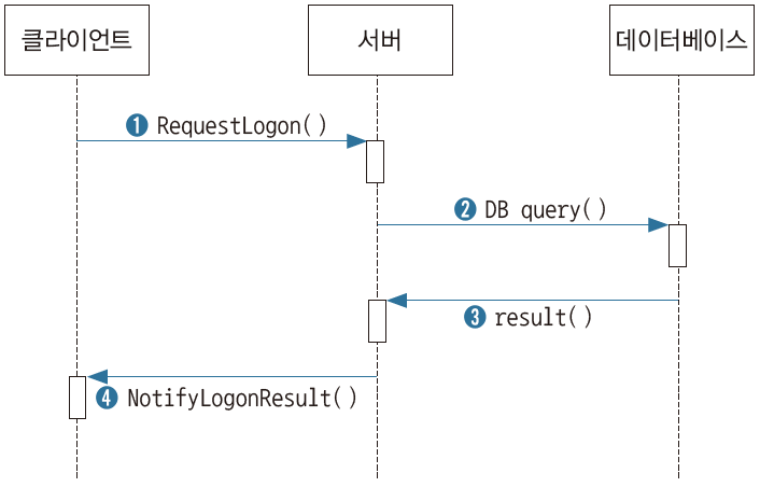


그림 10-1 로그온 처리 과정

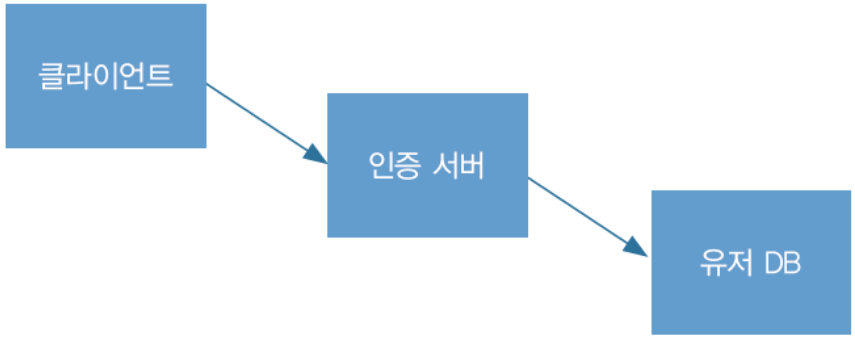


그림 10-2 로그온 처리를 하는 단일 서버

10.1 | 로그온 처리의 분산

- 클라이언트 개수가 늘어날 때마다 서버에서 부하가 걸리는 지점 찾기.

  성능 분석 도구를 이용하면 쉽다.

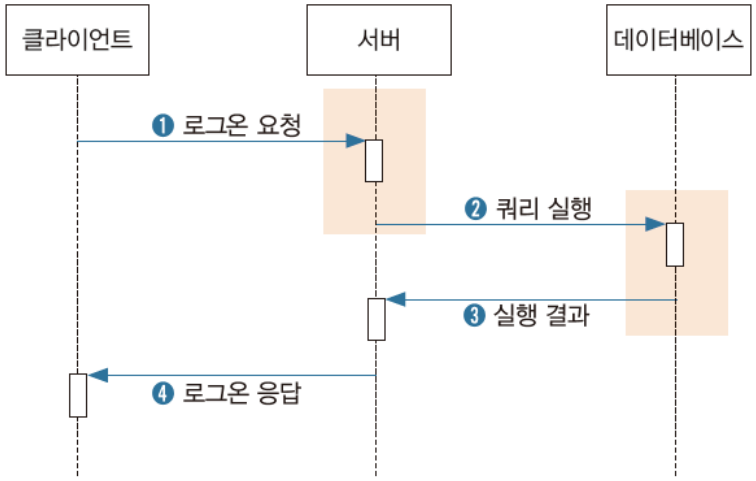


그림 10-3 색칠한 부분은 주요 과부하 지점

동시접속자가 많아지면 로그온 메시지를 복호화하는 연산 과정 때문에 인증 서버에서 CPU 사용량이 점점 과부하에 가까워진다.

자연스럽게 데이터베이스 서버도 과부하에 점점 가까워지는데, 이는 질의 분석이나 디스크 스토리지에서 읽기 액세스를 하는 데 걸리는 시간 때문이다.

10.1 | 로그온 처리의 분산

- 데이터 단위 분산 하기

  인증 서버를 서버 기기 여러 대로 수평 확장한다.

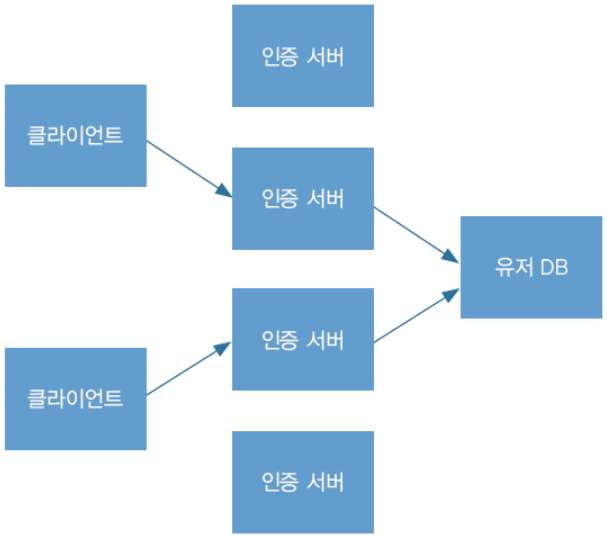


그림 10-4 데이터 단위 수평 확장

수평 확장한 후에는 다음과 같이 작동한다.

1. 클라이언트는 인증 서버 중 하나에 접속한다.
2. 클라이언트는 서버 주소 목록을 가지고 있다. 그리고 서버 중 랜덤으로 하나를 선택하고 접속한다.
3. ID와 비밀번호를 보낸다. 인증 서버는 이것을 앞서 알아본 방식과 동일하게 처리한다. 인증 서버는 플레이어 정보를 액세스하기 위해 데이터베이스에 질의를 실행한다.

10.1 | 로그온 처리의 분산

- 로드 밸런서(load balancer)

  클라이언트가 서버 주소 목록을 가지고 있지 않을 경우, 클라이언트에서 들어오는 연결을 서로 다른 인증 서버에 분배해 주는 역할을 하는 하드웨어를 그 사이에 둔다.

  이러한 역할을 하는 것을 로드 밸런서(load balancer)라고 한다.

  로드 밸런서 자체가 과부하 지점이 될 수도 있습니다만, 로드 밸런서는 매우 많은 양을 처리할 수 있다는 장점이 있다.

  로드 밸런서도 과부하에 종종 걸리는데, 이를 해결하고자 DNS 서버의 작동 방식을 특이하게 하는 방법을 이용한다.

  (예: L4-Switch)

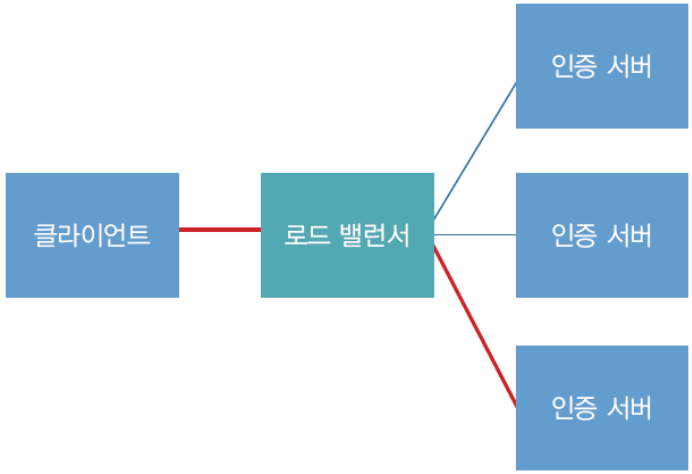


그림 10-5 로드 밸런서

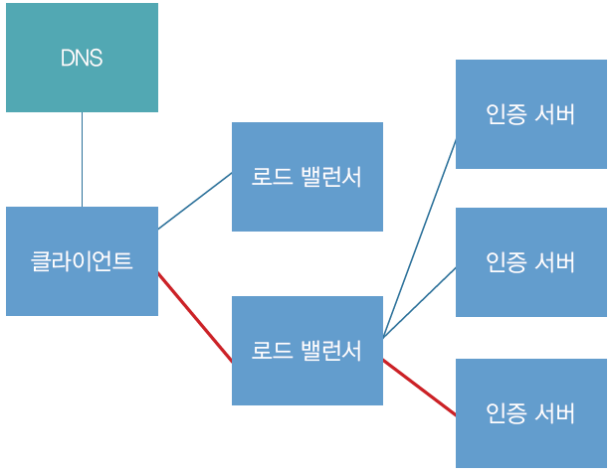


그림 10-6 로드 밸런서의 부하를 분산해 주는 DNS

10.1 | 로그온 처리의 분산

로드 밸런서를 여러 개 둔 후 각 로드 밸런서 뒤에 서버들을 둔다.

클라이언트는 서버 주소를 도메인 이름 형태로 가진다.

클라이언트가 서버에 접속하려면 도메인 이름을 IP 주소로 변경해야 하는데, 이를 위해 클라이언트는 도메인 이름 서버(Domain NameServer, DNS)에 server.mygame.com의 IP 주소가 무엇인지 묻습니다(소켓의 getaddrinfo() 함수가이 역할을 한다).

DNS는 그림 10-6의 로드 밸런서 중 주소 하나를 무작위로 리턴한다.

클라이언트는 자연스럽게 로드 밸런서 여러 개 중 하나에 접근하며, 이후 처리는 나머지와 같다.

이렇게 함으로써 로드 밸런서가 과부하에 걸리는 것을 예방할 수 있다.

이렇게 하는 대표적인 방식이 바로 라운드 로빈(round robin) DNS.

클라이언트는 DNS에서 도메인 이름에 대한 IP 주소를 일단 확보하면 로컬 디스크에 저장해 두고, 그 후 며칠 동안은 그것을 계속 사용한다.
즉, DNS조차 과부하에 걸리는 상황은 없다고 보면 된다.

10.2 | 데이터베이스의 수평 확장

- 인증 서버는 부하가 분산되나, 데이터베이스 서버(이하 DB 서버)에는 여전히 과부하가 몰린다

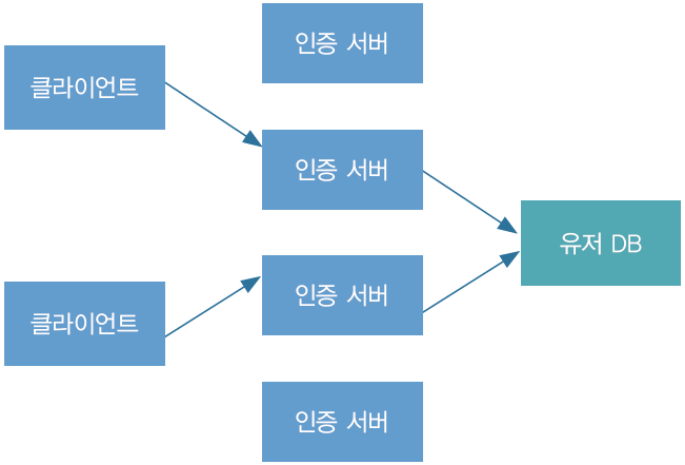


그림 10-7 데이터베이스 서버에는 여전히 과부하 발생

이제 DB 서버를 여럿으로 나눈다.

플레이어 간 상호 작용은 없다고 전제했으므로 데이터베이스 자체는 수평 확장하기가 쉽다.

플레이어 정보를 서로 다른 데이터베이스 샤드에 골고루 나누어 저장한다.

클라이언트는 여러 인증 서버 중 하나에 접속하고, ID와 비밀번호를 보낸다.

인증 서버는 입력받은 ID 문자열에서 어느 데이터베이스 샤드에 질의를 실행해야 할지 판단해야 한다.

10.2 | 데이터베이스의 수평 확장

- 사용자의 ID 문자열을 이용하여 해시 함수를 실행(해시함수는 일반적으로 많이 사용하는 형태면 됨.)

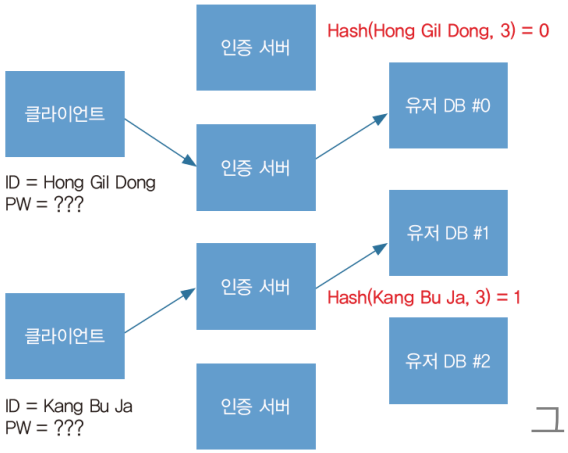
```
int Hash(string userID, int shardCount)
{
        int x = 0;
        foreach(ch in userID)
        {
                x ^= ch;                // XOR 연산으로 조합한다.
        }
        return x % shardCount;  // MOD 연산
}
```

중간정리

클라이언트는 여러 서버 중 하나를 무작위로 골라서 로그온 요청을 한다.

서버는 로그온 요청을 받으면 입력받은 ID를 해시 함수에 넣어서 샤드 인덱스를 구한다.

서버는 해당하는 데이터베이스 샤드에 질의를 던져서 나머지를 실행한다.



이렇게 하면 수평 확장이 완성된다.

그림 10-8 해시 함수를 이용하여 샤드 액세스

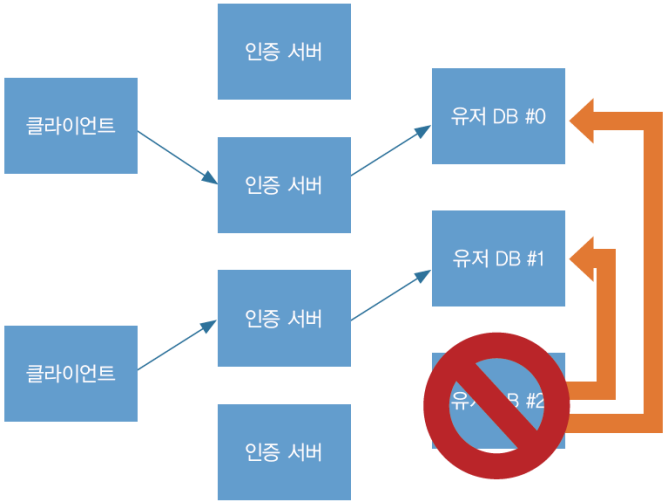
10.2 | 데이터베이스의 수평 확장

- 리해시(rehash)

  해시 테이블에서 각 항목(bucket)은 샤드에 해당한다.

  해시 테이블 알고리즘에서는 항목 개수가 달라지면 기존에 있던 모든 데이터에서 항목 재배치를 한다. 이를 리해시(rehash)라고 한다.

  데이터베이스 샤드가 추가되면 기존 레코드를 리해시해 주어야 한다.- 기존 데이터베이스 샤드에 있던 레코드 중 일부는 다른 샤드로 이동시켜야 한다.



데이터베이스 샤드 개수가 늘어나거나 줄어들 때, 리해시를 해야 하는 총 레코드 개수가 10억 개 정도로 매우 많아지는 문제를 개선하려면

1. 이동하는 레코드 개수가 최소가 되는 알고리즘을 사용한다.
2. 이동할 일이 없게 만든다.
3. 이동하더라도 이동하는 레코드 크기를 최소로 만든다.

그림 10-9 사라질 샤드의 데이터를 다른 샤드로 옮기기

10.2 | 데이터베이스의 수평 확장

- 매핑 DB

  덩치가 큰 데이터 구조체를 다룰 때, 성능상 구조체 자체보다 구조체의 포인터를 다루어야 할 때가 있다.
  여기서 핵심은 데이터 자체보다 데이터를 가리키는 또 다른 데이터를 다룬다는 데 있다.
  이러한 요령을 데이터베이스에 응용- 이를 매핑 DB라고 한다.

- 시퀀스

  1. 클라이언트는 인증 서버 중 하나에 접속한다.
  2. 인증 서버는 매핑 DB에 질의를 해서 대응하는 DB 인덱스, 즉 샤드 넘버를 얻어 온다.
  3. 인증 서버는 해당 DB에 플레이어 관련 CRUD 질의를 실행한다.

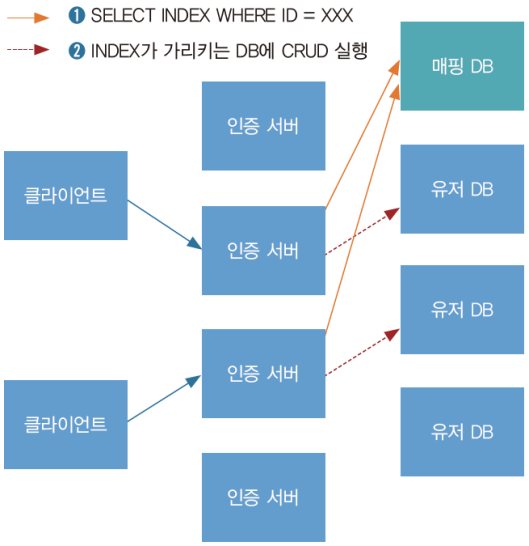


그림 10-17 매핑 DB를 추가

10.2 | 데이터베이스의 수평 확장

- 데이터베이스 샤드를 증설하면

새 플레이어 정보를 추가할 때는 새로 추가된 샤드의 빈 공간에 집중해서 채운다.

샤드를 줄일 때는 사라질 샤드에 대한 레코드 이동을 해야 한다. 물론 매핑 DB에서도 변경을 가해 주어야 한다.

어차피 레코드 이동과 매핑 DB 변경 때문에 DB가 한동안 블로킹되는 것은 어쩔 수 없습니다.

실제로 게임 서비스를 할 때, 사용자 수 폭증으로 샤드 개수를 늘려야 할 때는 신속하게 해야 하지만, 사용자 수 급감으로 샤드 개수를 줄일 때는 신속하게 하지 않아도 된다.

천천히 샤드를 줄이면서 레코드를 이동시켜 준다.

이렇게 만들면 매핑 DB는 사용자가 매우 많을 때 확장성의 병목이 될 수 있다.

다른 모든 것이 확장성에 대비했더라도 이것 하나 때문에 아무 소용이 없어진다. 이렇게 사고뭉치가 될 위험성을 가진 것을 단일 실패 지점(Single Point Of Failure, SPOF)이라고 한다.

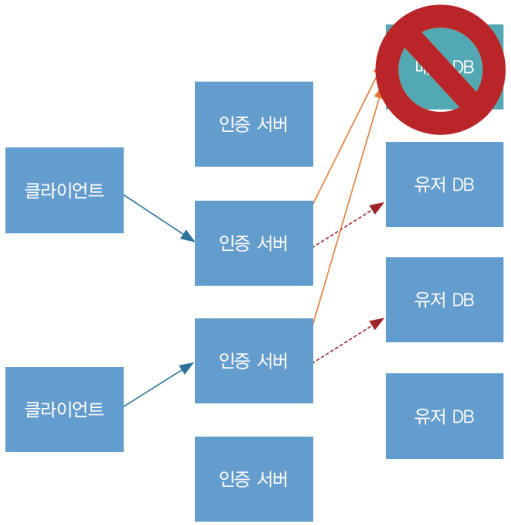


그림 10-18 정작 매핑 DB에 과부하가 걸리면 안 됨

10.2 | 데이터베이스의 수평 확장

- 매핑 DB에 확장성 추가하기

  매핑 DB가 SPOF가 되지 않게 하려면 매핑 DB도 확장성을 갖추어야 한다.

  매핑 DB의 레코드를 여러 샤드에 나눈다.

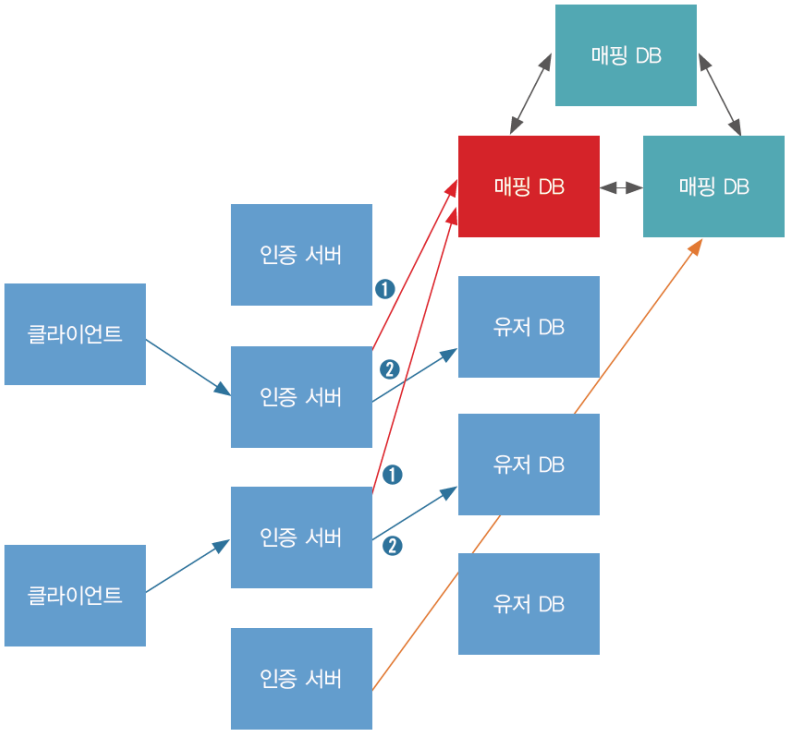


그림 10-19 매핑 DB에 확장성 추가

10.2 | 데이터베이스의 수평 확장

- 로그인을 담당하는 서버가 분산 처리된 형태

  로그인 서버

  매핑 DB 서버 샤드

  플레이어 정보를 담은 DB 서버 샤드

- 프로세스 시퀀스

  1. 클라이언트는 인증 서버 중 하나에 접속한다.
  2. 인증 서버는 매핑 데이터베이스 중 하나를 골라 유저 ID에 대응하는 데이터베이스 #ID를 얻는 질의를 던진다.
  3. 인증 서버는 해당 DB에 CRUD 질의를 던진다.

- 로그인 서버 (서버에 인증 정보를 전달하는 방법)

  1. 클라이언트와 로그인 서버가 있다. 서버는 클라이언트가 John임을 인증함.
  2. 클라이언트가 다른 서버로 접속하려고 한다. 그리고 John으로 인증하고자 한다.
  3. 로그인 서버는 클라이언트에 어떤 비밀 징표를 보낸다.
  4. 클라이언트는 그 비밀 징표를 갖고 다른 서버에 보낸다.
  5. 그 서버는 징표를 보고 클라이언트가 John임을 알아챈다.

10.2 | 데이터베이스의 수평 확장

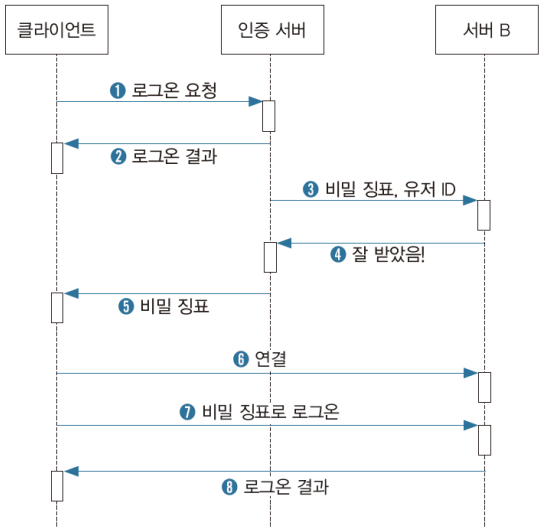


그림 10-20
로그온 처리가 별도 서버로 분리된 후 시퀀스도

1. 클라이언트는 로그인 서버에 ID와 비밀번호를 보낸다.
2. 로그인 서버는 클라이언트에 로그인이 성공했다고 알려 준다. 클라이언트는 서버 B에 접속하고, 로그인 인증도 받으려고 한다.
3. 로그인 서버는 서버 B에 그 비밀 징표와 유저 ID를 보낸다. 이를 크리덴셜(credential)이라고 한다. 즉, "크리덴셜 어쩌고저쩌고를 누가 보내면 걔는 John이야~"를 알려 준다.
4. 서버 B는 "잘 받았다(acknowledge)."를 응답.
5. 로그인 서버는 클라이언트에 크리덴셜을 보낸다.
6. 클라이언트는 서버 B에 연결한다.
7. 크리덴셜을 보낸다.
8. 서버는 크리덴셜을 받고 나서 유저가 누군지 알아챈다. 그리고 그 결과 "로그인 성공"을 클라이언트에 회신한다.

10.3 | 매치메이킹의 분산 처리

- 매치메이킹(matchmaking)

  멀티플레이 게임에서는 서로 다른 플레이어 간 대전이나 협력 플레이를 위해, 다른 플레이어들을 찾아 모은다.

  일반적으로 이러한 일을 담당하는 서버를 로비(lobby) 서버라고 한다.

  먼저 매치메이킹 사용자 시나리오, 즉 유즈 케이스(use case)를 준비한다. 게임 기획자가 주기도 한다.

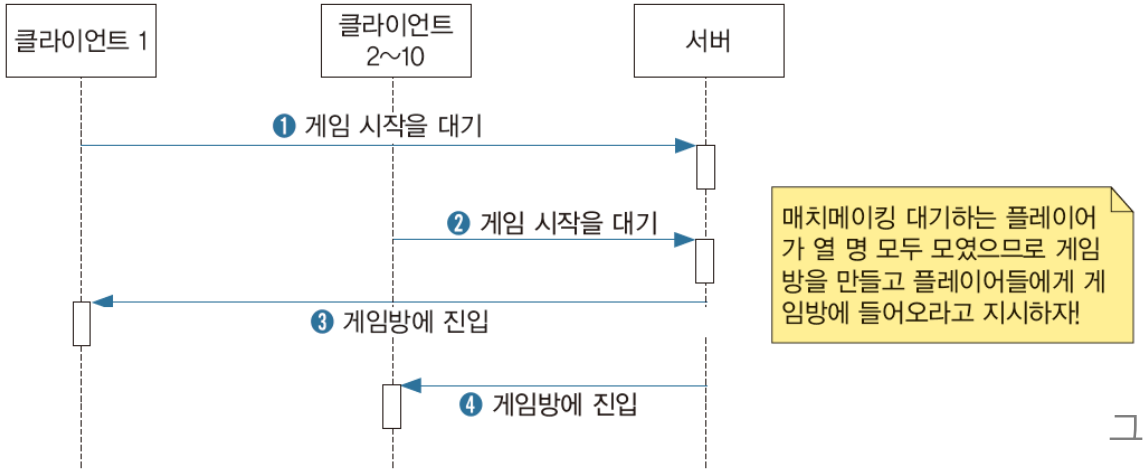


그림 10-21 매치메이킹 사용자 시나리오

❶ 게임을 플레이하려는 플레이어는 게임 시작하기를 누르고 대기한다.

❷ 플레이어와 실력이 비슷한 다른 플레이어 아홉 명이 게임 시작하기를 누르면 매칭 성공. 대기를 멈추고 매칭 준비를 하면 된다.

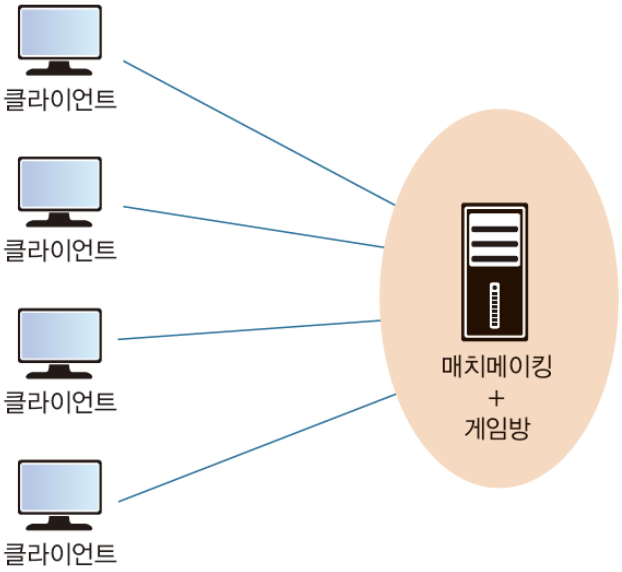
❸ 매칭 성공 후 플레이어 열 명이 플레이가 가능한 방을 만든다.

❹ 게임방에서 플레이어들이 채팅으로 작전 회의를 한다. 그리고 게임을 시작한다.

10.3 | 매치메이킹의 분산 처리

- 분산 서버를 설계할 때 우리가 해야 하는 절차
  1. 단일 서버에서 과부하 지점을 찾는다.
  2. 과부하 지점을 중심으로 분산 서버를 설계한다.

- 매치메이킹과 전투를 담당하는 서버가 분산되어 있지 않은 경우



동시접속자가 늘어나면 다음 현상이 발생한다.
게임방 안의 월드 시뮬레이션을 위한 서버 CPU 과부하
매치메이킹에 동시접속자가 몰릴 경우 서버 CPU, 램 사용량 과부하

그림 10-22 서버 한 대가 매치메이킹과 게임방 처리를 담당하는 경우

10.3 | 매치메이킹의 분산 처리

- 배틀 서버(battle server)

  한 게임방 안에서 플레이어들은 잦은 상호 작용을 한다.

  즉, 방 안에 있는 플레이어끼리는 데이터 응집력이 높으며, 서로 다른 방에 있는 플레이어끼리는 응집력이 없다.

  따라서 게임 서버에서 게임방 처리를 여러 서버로 분산시켜야 하고, 게임방을 처리하는 서버를 배틀 서버라고 한다.

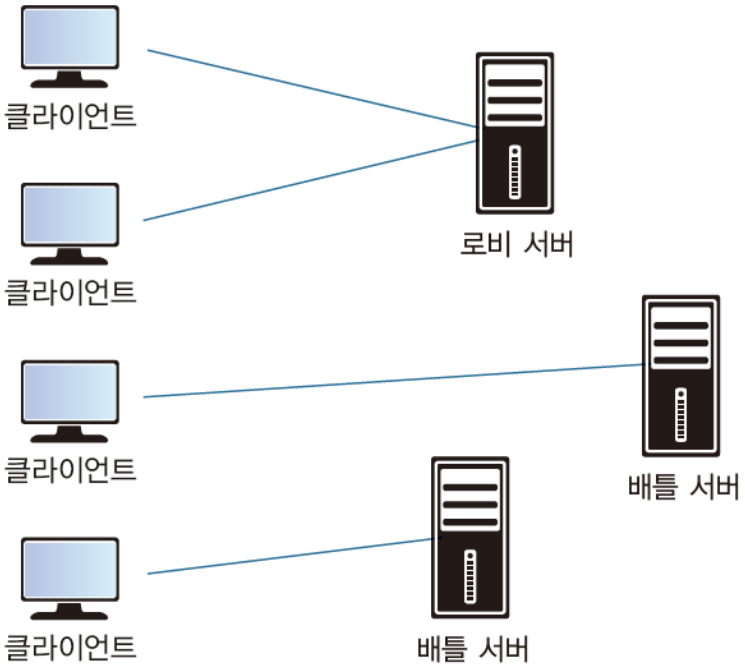


그림 10-23 게임방 처리를 배틀 서버로 분산

매치메이킹 후 멀티플레이 중 레이턴시를 줄이기 위해 배틀 서버를 여러 리전(region)으로 나눈다.(리전 : 어떤 국가나 거대한 주)

그림 10-24 로비 서버는 한 리전에, 배틀 서버는 여러 리전에 분산

10.3 | 매치메이킹의 분산 처리

- 배틀 서버를 분산 처리한 후 네트워크 시퀀스

1. 클라이언트는 매치메이킹을 하는 동안에는 로비 서버에 접속을 유지한다.
2. 매치메이킹이 완료되어 게임방을 만들 때 가장 접속자가 적은 배틀 서버에 접속한다. 이때 매치메이킹 서버와 연결을 끊는 것을 권장.
3. 게임방에서 게임을 즐기고, 게임이 끝난 후에 클라이언트는 다시 매치메이킹을 하기 위해 로비 서버로 돌아온다.

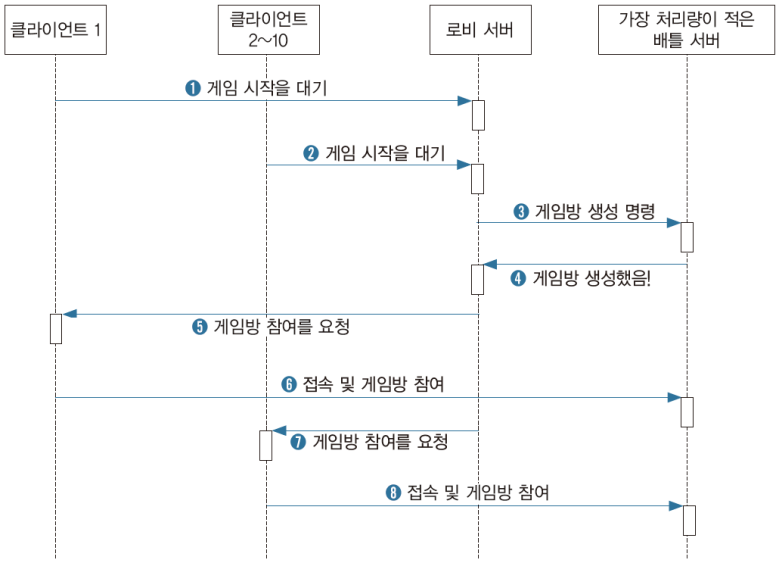


그림 10-25 배틀 서버 분산 후 네트워크 시퀀스

❶ 클라이언트 1은 "다른 플레이어를 찾아 달라."라고 매칭을 요청한다.

❷ 클라이언트 2~10도 같은 요청을 한다.

❸ 로비 서버는 매칭되는 클라이언트 1~10을 찾아낸다. 그리고 가장 처리량이 적은 배틀 서버를 찾아낸다. 그 배틀 서버에 "방을 만들어라."라고 요청한다.

❹ 배틀 서버는 ❸ 요청에 대한 응답을 한다.

❺ 방이 성공적으로 만들어졌으므로 클라이언트 1에 "너를 위한 방이 만들어졌다. 그 방으로 들어가라."라고 요청한다.

❻ 클라이언트 1은 요청에 따라 배틀 서버에 접속하고 인증을 한다. 앞서 언급했던 크리덴셜 방법으로 인증을 한다.

❼ 클라이언트 2~10도 ❺처럼 한다.

❽ 클라이언트 2~10도 ❻ 처럼 한다.

10.3 | 매치메이킹의 분산 처리

- 로비 서버를 분산 처리 하기

  매치메이킹만 담당하는 로비 서버는 한 대 뿐이므로 동시접속자가 계속 증가하면 결국 로비 서버가 과부하에 걸릴 것이다.

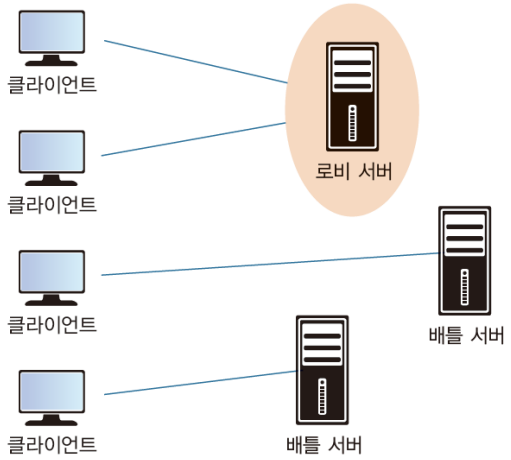


그림 10-26 한 대뿐인 로비 서버는 결국 과부하에 걸림

- 플레이어 가 매칭을 할 때 필요한 것들

  1. 플레이어의 실력 정보: 대다수 게임에서는 실력이 비슷한 플레이어끼리 매칭하려고 한다. 플레이어의 실력 정보는 게임 콘텐츠마다 다르다.

  2. 매칭하기 버튼을 눌러서 대기 중인지 여부: 모든 플레이어는 "나는 지금 매칭을 기다리는중이다."라는 여부를 담은 불형(boolean) 값을 가져야 한다.

10.3 | 매치메이킹의 분산 처리

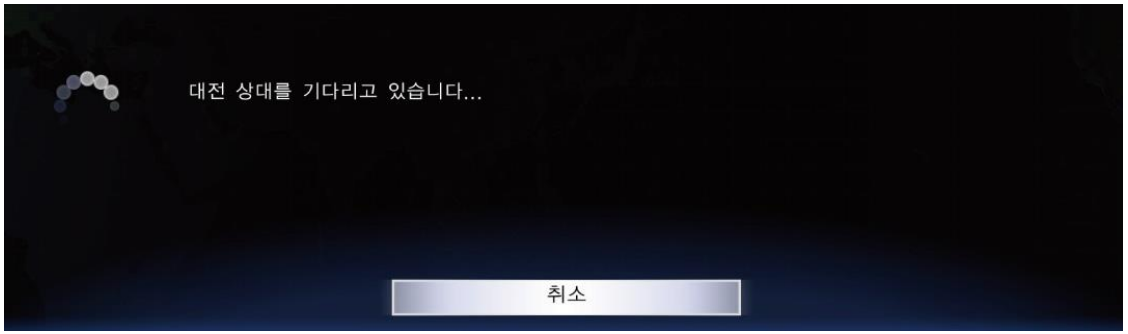


그림 10-27
매치메이킹 대기 여부 확인

- 매치메이킹 성립 조건

  나는 지금 매칭을 기다리고 있습니다.
  너도 지금 매칭을 기다리고 있습니다.
  너와 내 실력은 비슷합니다.

→ 매치메이킹을 위해 필요한 데이터의 이러한 조건들을 만족하는 플레이어 목록

- 서버 간 상호 작용을 구현하는 방식 세 가지

  동기 분산 처리
  비동기 분산 처리
  데이터 복제에 기반을 둔 로컬 처리

10.3 | 매치메이킹의 분산 처리

- 매치메이킹을 위한 의사 코드 - 비동기 분산 처리

```
Match(player)
{
        var list;
        for each (p in players)                                    //❶
        {
                if (p != player
                        && p.waitingForGame
                        && SimilarTo(player.elo, p.elo)            //❷
                {
                        list.add(p);
                }
        }
        if (list.count = = 9)
        {
                StartGameRoom(list);
        }
}
```

매치메이킹을 하려면 ❶ 의 루프를 보면서 ❷ 의 조건문을 만족해야 gks다. ❷ 의 조건문은 데이터 읽기를 하므로 비동기 분산 처리를 할 수 없다.

❷ 의 if 구문은 구현 자체가 불가능 하며 이렇게 되면 데이터 복제에 기반을 둔 로컬 처리가 불가피하다.

10.3 | 매치메이킹의 분산 처리

- 로비서버의 작동

1. 플레이어 정보 목록을 서로 동기화한다. 플레이어 정보에는 플레이어 실력에 대한 정보와 '매칭을 기다리는 중' 여부를 나타내는 불형 값이 있다.
2. 매치메이킹을 할 때 조건을 만족하는 플레이어를 이미 동기화받은 플레이어 정보 목록에서 찾아낸다.
3. 매치메이킹이 끝난 플레이어를 배틀 서버로 이동시킨다.

- 로비서버의 작동방식

1. 로비 서버를 여러 대 둔다.
2. 클라이언트는 로비 서버 중 아무것이나 하나 골라 그 안에서 접속하게 한다.
3. 모든 로비 서버 간 통신을 통해 플레이어 목록을 동기화한다. 예를 들어 플레이어 A가 매칭 시작 버튼을 누르면, 플레이어 A가 접속된 로비 서버 1은 나머지 로비 서버한테 '플레이어 A가 매칭 대기 중 = TRUE'라고 알려 준다. 나머지 로비 서버는 이를 자기가 가진 플레이어 목록에 업데이트한다. (전부는 하지 말고 플레이어 목록 중 게임 시작을 눌러 매치메이킹을 기다리는 것들만 동기화하는 것이 경제적이다.)

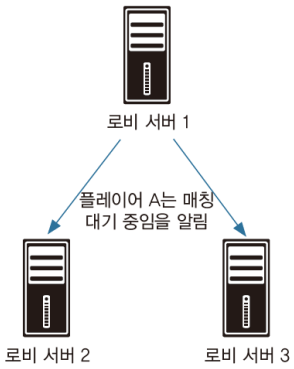


그림 10-28 로비 서버 1이 플레이어 A의 상태 변화를 알림

10.3 | 매치메이킹의 분산 처리

- 로비서버의 작동

1. 플레이어 정보 목록을 서로 동기화한다. 플레이어 정보에는 플레이어 실력에 대한 정보와 '매칭을 기다리는 중' 여부를 나타내는 불형 값이 있다.
2. 매치메이킹을 할 때 조건을 만족하는 플레이어를 이미 동기화받은 플레이어 정보 목록에서 찾아낸다.
3. 매치메이킹이 끝난 플레이어를 배틀 서버로 이동시킨다.

- 로비서버의 작동방식

1. 로비 서버를 여러 대 둔다.
2. 클라이언트는 로비 서버 중 아무것이나 하나 골라 그 안에서 접속하게 한다.
3. 모든 로비 서버 간 통신을 통해 플레이어 목록을 동기화한다. 예를 들어 플레이어 A가 매칭 시작 버튼을 누르면, 플레이어 A가 접속된 로비 서버 1은 나머지 로비 서버한테 '플레이어 A가 매칭 대기 중 = TRUE'라고 알려 준다. 나머지 로비 서버는 이를 자기가 가진 플레이어 목록에 업데이트한다. (전부는 하지 말고 플레이어 목록 중 게임 시작을 눌러 매치메이킹을 기다리는 것들만 동기화하는 것이 경제적이다.)
4. 반대로 매칭 대기 취소 버튼을 누르면 3과 마찬가지로 로비 서버 1은 다른 서버한테 '매칭 대기 중 = FALSE'라고 알려줌.
5. 플레이어 목록에는 플레이어가 어느 로비 서버에 있는지 정보도 같이 있어야 한다. 그래야 게임방 시작을 할 때 해당 플레이어의 로비 서버에 메시지를 보낼 수 있기 때문.

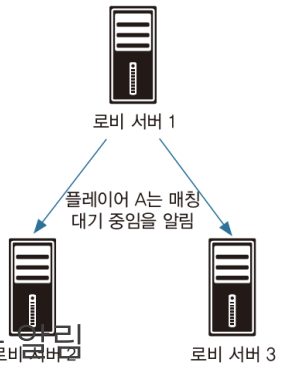


그림 10-28 로비 서버 1이 플레이어 A의 상태 변화를 알림

10.3 | 매치메이킹의 분산 처리

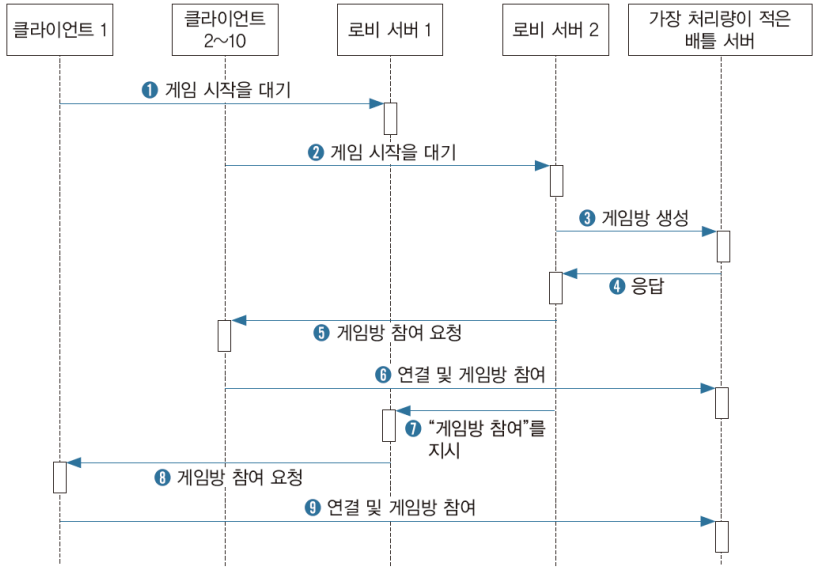


그림 10-29 매치메이킹 분산 처리의 시퀀스 다이어그램

클라이언트 1은 로비 서버 1에 접속해 있고, 클라이언트 2~10은 로비 서버 2에 접속해 있다.

❶ 클라이언트 1이 매칭 대기 버튼을 누른다. 그리고 로비 서버 1은 이를 안다. 그림 10-29에서는 생략되었지만, 이것으로 플레이어 정보가 변했다는 것을 다른 로비 서버에도 알려 준다.

❷ 클라이언트 2~10이 각각 매칭 대기 버튼을 누른다. 그리고 로비 서버 2는 이를 알게 되며, 마찬가지로 다른 로비 서버에 이 변화를 알려 준다.

❸ 로비 서버 2에서 클라이언트 1~10을 모두 매칭시켰고, 이 열 명이 모여 게임방에 들어가게 한다. 이 로비 서버는 가장 처리량이 적은 배틀 서버를 골라서 해당 배틀 서버에 "방을 만들어라."라고 지시한다.

❹ ❸ 에 대한 응답을 보낸다. 이제 로비 서버 2는 게임방이 확실히 존재한다는 것을 안다.

❺ 로비 서버 2는 클라이언트 2~10에 이 게임방에 접속하라고 알려 주면서, 게임방의 서버 주소와 게임방에 들어갈 수 있는 징표, 즉 인증 정보(크리덴셜)도 알려 준다.

❻ 클라이언트 2~10은 각각 지시에 따라 배틀 서버에 접속하고 인증한다.

❼ 로비 서버 2는 로비 서버 1에 있으므로 클라이언트 1을 직접 지시할 수 없다. 따라서 로비서버 2는 로비 서버 1에 "네가 가진 클라이언트 1에 배틀 서버로 가라고 말해라."라고 간접적으로 지시한다.

❽ 로비 서버 1은 클라이언트 1에 ❺ 와 동일한 지시를 한다.

10.3 | 매치메이킹의 분산 처리

- 로비서버 개수와 서버 간 통신량이 매우 많은 경우

  매치메이킹 대기 중인 플레이어 목록 중에서 응집력이 강한 것과 낮은 것을 구별한다.

  실력이 비슷한 플레이어끼리 묶으면 그들끼리는 높은 응집력을 지닌다.

  실력 차이가 크게 나는 플레이어 데이터 간 응집력은 낮으므로 그들끼리 매칭할 일은 별로 없다.

  → 각 로비 서버는 서로 다른 범위의 실력을 가진 플레이어들을 받아들이면 된다.

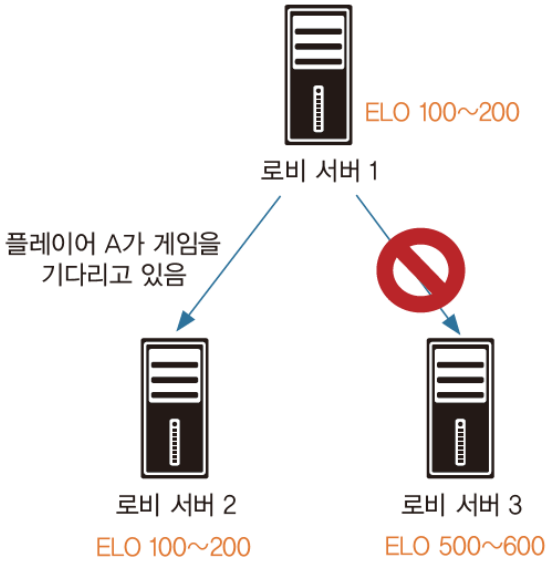


그림 10-30 플레이어의 실력 점수에 따라 서버 분산

로비 서버 1과 로비 서버 2는 플레이어 실력 점수(ELO)가 100~200인 것만 다루고, 로비 서버 3은 플레이어 실력 점수가 500~600인 것만 다룬다.

이때 로비 서버 1에 접속한 플레이어 정보('매칭 대기 중 = TRUE 또는 FALSE')가 변경되더라도, 로비 서버 3에 이를 동기화할 필요가 없다.

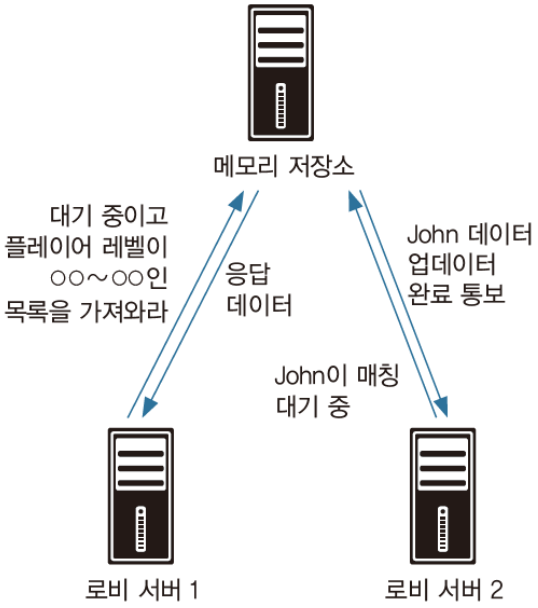
이렇게 하면 비슷한 실력의 플레이어끼리만 동기화를 하므로 결과적으로 각 로비 서버가 수신하거나 송신해야 할 데이터의 양이 적어진다.

로비 서버 간 플레이어 데이터를 공유할 경우 간발의 차이로 옛날 데이터를 액세스하는 불상사인 데이터 스테일 문제가 발생한다.

이를 근본적으로 해결할 방법은 없으며, 매치메이킹 재시작을 자동으로 하는 등 수습 처리를 해야 한다. 플레이어 입장에서는 그저 매치메이킹에 좀더 시간이 걸리는 것으로만 인식한다.

10.3 | 매치메이킹의 분산 처리

- 플레이어 목록을 로비 서버 간 복제하지 않고, 플레이어 목록을 한군데 별도 서버(메모리 저장소)에 보관하기



각 로비 서버는 플레이 목록을 직접 가지고 있지 않으며, 플레이어 목록의 조건부 검색을 메모리 저장소 서버에 요청하여 시행한다.

로비 서버는 플레이어 목록을 메모리 저장소에서 가져오고,
변경도 메모리 저장소에서 한다.

이 방식은 로비 서버 안에 연산량이 많고, 메모리 저장소 서버 내부 연산량이 적을 때 효과적이다.

메모리 저장소 서버에 몰리는 메시지 통신량이 매우 많을 때는 메모리 저장소 서버를 수평 확장(파티셔닝)하는 방법으로 해결할 수 있다.

그림 10-31
플레이어 목록을 메모리 저장소 역할을 하는 서버 기기에 따로 보관

10.4 | 몬스터 NPC 처리의 분산 처리

- 서버에서 하는 몬스터 관련 처리

  일정 시간마다 몬스터의 움직임을 델타 타임(delta time, 1/30초 혹은 1/10초)만큼 시뮬레이션(위치 이동, 모션 변화)한다.

  일정 조건(시간)마다 몬스터의 움직임 결과를 가시자 클라이언트에 보낸다.

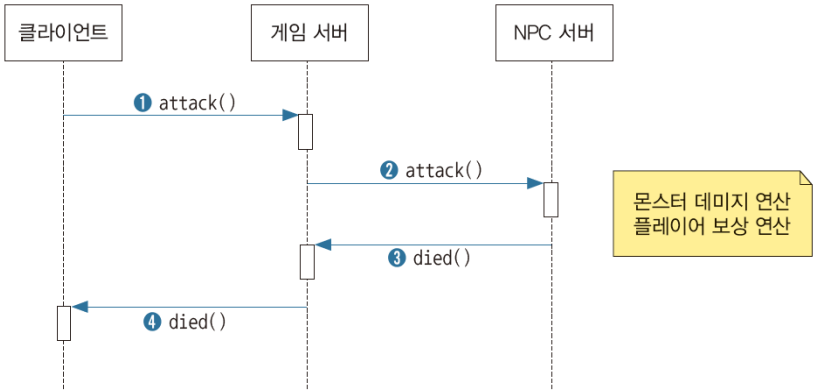
  플레이어가 몬스터와 상호 작용(공격)할 경우, 이에 대한 연산을 처리한 후 몬스터 및 플레이어 상태를 갱신(몬스터 사망 또는 플레이어 보상)한다.

- 성능상 고려해야 할 것

  보통 몬스터 수가 플레이어 수보다 훨씬 많다. 게임 플레이어가 사냥할 수 있는 몬스터가 주변에 많아야 게임도 재미있기 때문.

  몬스터 움직임을 위한 연산량이 많이 필요할 수도 있다. 예를 들어 몬스터가 플레이어를 추격하려면 길찾기 알고리즘을 연산해야 할 수 있다. 길찾기 알고리즘은 A* 알고리즘 등을 쓰기에 연산량이 많이 필요하다.

  몬스터의 연산 로직에는 게임 기획자의 아이디어가 많이 들어가기 때문에 프로그램 구조가 복잡해지거나 버그가 있을 수 있다.



1. 게임 클라이언트는 게임 서버에 "플레이어가 몬스터를 공격했다."라는 메시지를 보낸다.
2. 게임 서버는 몬스터를 담당하는 서버에 이러한 내용이 담긴 메시지를 보낸다. 몬스터를 담당하는 서버는 몬스터가 피해를 입는 처리와 플레이어에 보상 아이템을 주는 처리를 한다.
3. 몬스터를 담당하는 서버는 결과를 게임 서버에 알려 준다.
4. 게임 서버는 이 결과를 클라이언트에 알려 준다.

그림 10-32 몬스터 처리를 담당하는 서버 분리

10.4 | 몬스터 NPC 처리의 분산 처리

특별한 이유가 없는 한 몬스터 처리 평균 연산량은 1000클록 사이클 이하이다. 그런데 정작 이보다는 몬스터 관련 메시지를 주고받는 처리량이 더 많을 수 있다.

결과적으로 게임 서버 입장에서는 그림 10-33과 같이 실질적인 처리에 해당하는 300클록 사이클을 위해 4000클록 사이클이 운영체제에서 추가로 소모되며, 이렇게 되면 분산 처리를 하는 의미가 없어진다.

이 문제가 해결된다고 하더라도 분산 처리된 게임에서 플레이어가 광역 버프나 광역 공격 스킬을 쓸 때 순간적으로 지연 현상이 발생하거나, 최악의 경우 서비스가 중단되기도 한다.

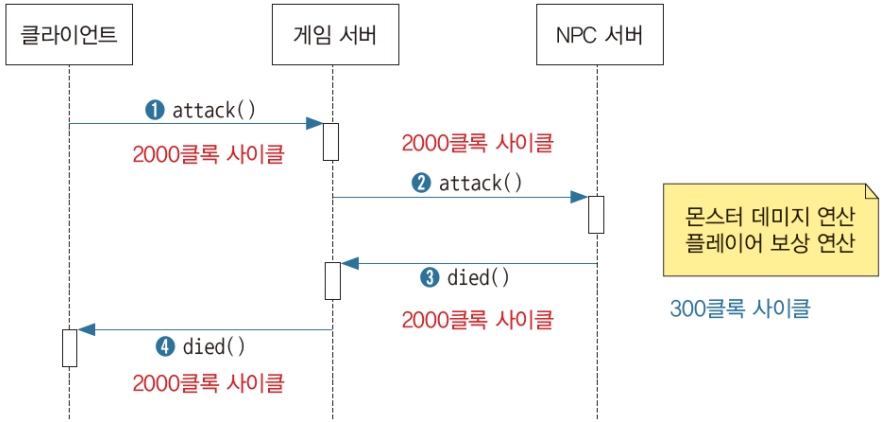


그림 10-33 해야 하는 연산량보다 메시지 송수신에 더 많은 처리량이 낭비됨

10.4 | 몬스터 NPC 처리의 분산 처리

- 의미 있는 분산 처리를 하려면

의미 있는 분산 처리를 하려면 먼저 성능 분석을 해야 한다.

코드 프로파일러 같은 도구를 이용 해서 분산 처리를 하기 전, 게임 서버 안에서 NPC 처리 코드가 실행 시간을 얼마나 점유하는지 분석하기

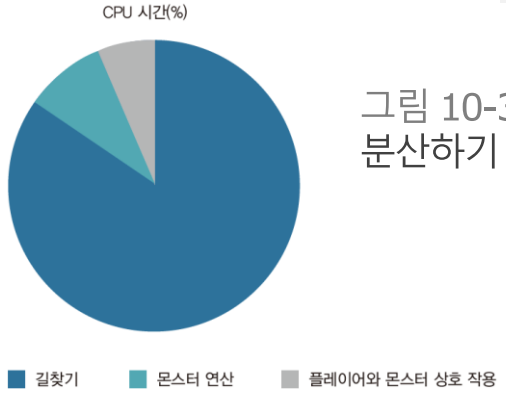


그림 10-34
분산하기 전 서버의 성능 분석 결과

몬스터의 상태 정보는 게임 서버가 직접 가지며, 플레이어와 몬스터 간 상호 작용은 게임 서버가 모두 처리한다(❶, ❷, ❼~❿). 몬스터가 어딘가로 이동해야 한다면, 길찾기 서버에 "내가 가야 하는 경로를 알려 줘."라고 요청한다(❺).
그러면 길찾기 전담 서버는 이에 대한 계산을 수행하여 그 결과를 게임 서버에 응답한다(❻).

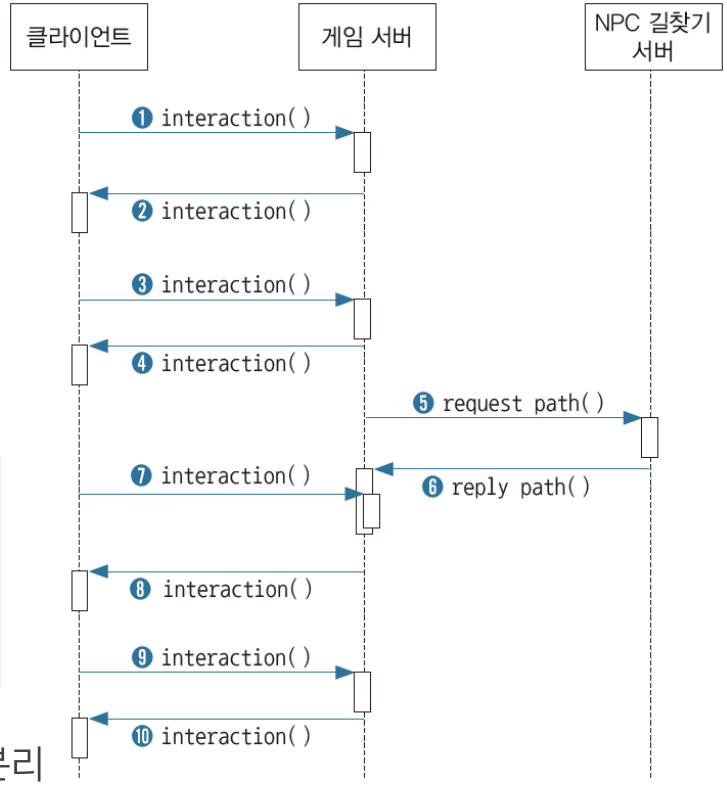


그림 10-35
길찾기 전담 서버 분리

10.5 | 플레이어 간 상호 작용 분산 처리

- 플레이어 간 상호 작용의 특징

  매우 정확해야 한다(간발의 차이로 PvP 승부 내기).

  높은 성능 수준이 요구된다.

  횟수가 매우 잦을 때가 있다. 예를 들어 두 플레이어가 서로에게 기관총을 난사하고 있다면 플레이어 간 상호 작용 메시지가 초당 5회 이상 발생한다.

  원자성이 있어야 한다. 상호 작용의 결과가 두 플레이어에게 모두 반영되거나 어느 쪽도 반영되지 않아야 한다. 예를 들어 플레이어 간 물물 교환을 할 경우 각각이 상대방에게 주려는 물품이 동시에 교환되거나 전혀 교환되지 않아야 한다.

  간발의 지연 시간도 문젯거리이다. 예를 들어 헤드샷을 쐈는데 상대방에게 1/20초 오차가 날 경우, 충분히 머리를 피할 수 있는 시간이므로 이 때문에 두 플레이어 간에 오차가 발생한다.

이렇게 까다로운 상황이다 보니 플레이어 간 상호 작용 처리는 분산하지 않는 것이 일반적이다.

굳이 분산해야 한다면, 상호 작용할 일이 없는 플레이어들을 분산하는 것이 최선이다. 즉, 지역적 분산 처리를 하는 것.

10.6 | 로그 및 통계 분석의 분산 처리

- 사용 시나리오

1. 게임 서버 안에서 일어나는 이벤트들을 로그로 남겨야 한다. 그 종류는 다음과 같다..
   - 10초마다 모든 플레이어 위치
   - 플레이어가 몬스터를 사냥할 때의 플레이어 종류와 레벨, 몬스터 종류와 레벨, 보상된 아이템
   - 플레이어가 죽을 때마다 가해자 캐릭터 종류와 레벨, 죽은 플레이어 레벨
   - 플레이어가 고급 아이템을 사용할 때마다 사용한 위치, 사용할 당시 가장 가까운 위치의 적
   - 캐릭터 레벨과 종류

2. 게임 제작자가 게임의 밸런싱을 맞추기 위해 다음 내용을 통계 수집할 수 있어야 한다.
   - 게임 월드 맵 위에 그려진 플레이어가 많이 죽는 지역 분포도
   - 플레이어의 경험치 증가 곡선
   - 몬스터 종류별 플레이어 보상량 분포 그래프

그림 10-36
플레이어가 얼마나 많이 머무르는지를 색으로 나타낸 모습

10.6 | 로그 및 통계 분석의 분산 처리

- 성능상 요구 사항

  로그를 통계 분석해서(Data Warehousing, DW) 그 결과를 게임 제작자에게 제공해야 한다.

  이 과정에서 효율적인 데이터 검색 기능이 필요하다.(아래는 코드 예시)

```
SELECT COUNT(*) FROM LOG1 WHERE V>10 AND V<=20
SELECT COUNT(*) FROM LOG1 WHERE V>20 AND V<=30
SELECT COUNT(*) FROM LOG1 WHERE V>30 AND V<=40
SELECT COUNT(*) FROM LOG1 WHERE V>40 AND V<=50
...
```

10.6 | 로그 및 통계 분석의 분산 처리

- 게임 서버에서 자신의 DB에 로그를 기록하는 경우

  DB에 기록하는 시간은 파일에 직접 기록하는 시간보다 훨씬 많이 걸림.

  DB에 기록하는 양이 순간적으로 많으면 처리 시간이 오래 걸리는 일이 대량으로 쌓이고, 당연히 DB에서 값을 가져오거나 기록할 때 걸리는 시간도 길어진다. 결국 게임 플레이 도중 지연 시간이 간헐적으로 발생한다.

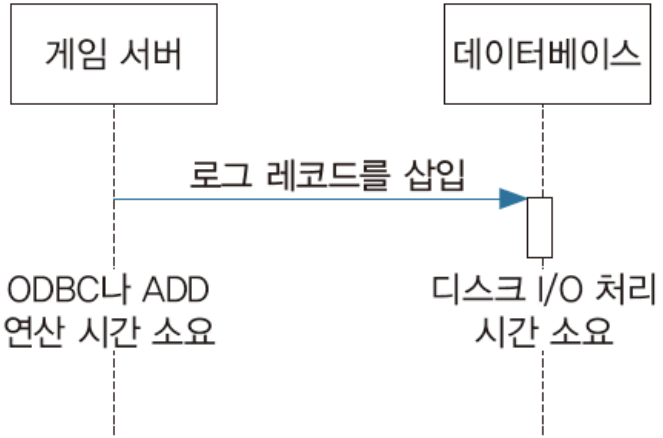


그림 10-37 DB에 레코드를 추가하는 시간에서 디스크 I/O는 큰 비중을 차지

10.6 | 로그 및 통계 분석의 분산 처리

- DB에 쌓인 데이터에서 통계 정보를 추출할 때

  데이터베이스에서 로그 분석을 수행하는 동안 DB는 굉장히 오랜 시간(10분)이 소요된다. 이 시간 동안 데이터베이스 부하는 최고조를 유지하기 때문에, 이 시간 동안 데이터베이스에 로그를 기록하는 시간은 평소보다 많이 느리다.

  DB에 게임 플레이어 정보가 있는 경우, 게임 서버가 DB에 플레이어 정보를 읽거나 쓸 때 평소보다 많이 느려지므로 로그를 담는 DB에 순간적으로 디스크 I/O가 많이 발생한다.

  DB를 액세스하는 프로그램 모듈(ODBC 또는 ADO) 자체가 연산량이 많으므로, DB에 로그 쌓기 질의를 수행하는 로그 서버는 따로 두면 좋다. 즉, 게임 서버와 로그 DB 서버는 분산해야 한다.

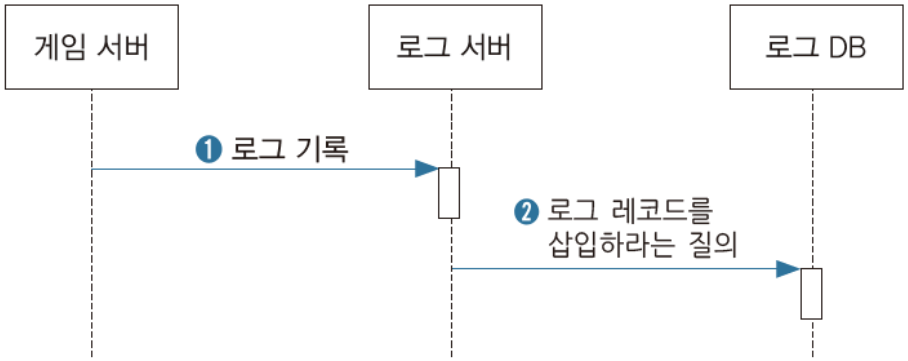


그림 10-38 게임 서버와 로그 서버 분리

1 게임 서버에서 로그를 쌓을 일이 있으면, 로그 서버에 로그를 기록하라고 지시한다.

2 로그 서버는 로그 DB에 로그를 쌓는 질의 구문을 실행한다.

10.6 | 로그 및 통계 분석의 분산 처리

- 로그 DB가 중간에 죽더라도 신뢰성을 가지게 하려면

  로그 서버는 게임 서버에서 '로그를 남겨라'는 지시를 받는다.

  로그 서버는 이 지시 내용을 로컬 파일에 일단 저장하고, 로그 DB에 질의를 던진다.

  질의 실행 후 로컬 파일에 있는 해당 내용을 삭제한다.

  로그 서버가 중간에 죽었다가 다시 켜졌을 때, 해당 로컬 파일에서 아직 기록하지 않은 로그를 로그 DB에 채워 넣으면 된다.

10.7 | 게임 장르별 분산 서버 형태

- MMORPG 게임

  게임 월드를 구성하는 소규모 지역별로 담당 서버를 둔다.
  각 서버가 담당하는 지역은 동시접속자를 수천 명 규모로 유지한다.
  경매장 같은 특수한 목적의 서버를 별도로 분리하기도 한다.

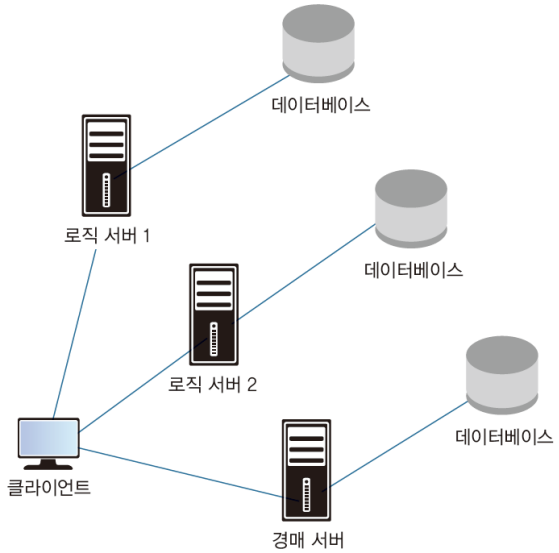


그림 10-39 지역별 서버 분리, 특수 목적의 서버도 분리

소규모 지역 하나도 게임 서버 한 대에서 다 처리하기 힘들 때는

- NPC 관련 로직에 과부하가 걸리면 NPC 관련 로직을 다른 기기로 분산한다. 플레이어 이동 연산을 하는 데 과부하가 걸리면 플레이어 이동 연산만 담당하는 서버로 분산한다.
- 소켓 I/O에서 과부하가 걸릴 때는 게임 서버와 서버를 두어 릴레이

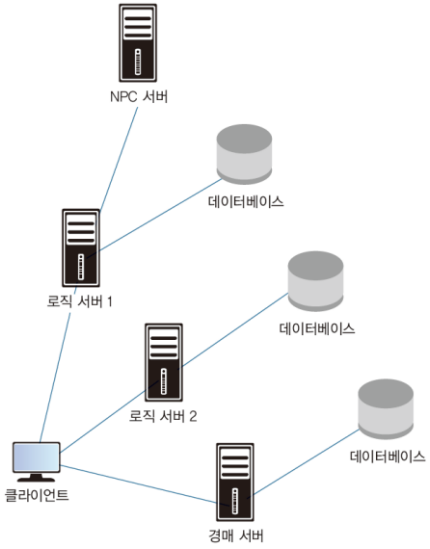


그림 10-40 NPC 관련 로직을 따로 분리

10.7 | 게임 장르별 분산 서버 형태

어떤 MMO 게임에서는 플레이어 수만 명에서 수백만 명이 매우 넓은 하나의 지역에서 플레이를 한다.

월드 자체가 이음새가 없거나 있더라도 너무 많을 때는 이를 소규모 지역으로 가르기도 어렵다.

이때는 클라이언트 메시지를 적절한 서버로 전달해 주는 릴레이 서버와 넓은 지역을 작은 단위로 나누어 처리하는 로직 서버로 구별한다.

그림 10-41
이음새 없이 넓은 월드 하나에 플레이어 수만 명 이상을 수용하는 게임

그림 10-42
이음새가 너무 많은 게임

10.7 | 게임 장르별 분산 서버 형태

- 심리스(seamless) MMO 서버 구조

로직 서버는 일정 지역 안에 있는 캐릭터의 게임 로직 처리를 담당하고
게임 로직 처리를 하면서 변경되는 정보(위치, 체력 등)는 인접한 지역을 담당하는
다른 로직 서버에 지속적으로 복제한다.

이러한 서버 구조에서는 서버당 통신량과 OS 레벨 처리량이 많이 증가한다.
따라서 서버 하나가 처리할 수 있는 클라이언트 개수가 상대적으로 적어지므로
서버 운영 효율성이 떨어진다.

심리스 MMO 서버는 같은 로직을 개발하더라도 구현이 더 복잡할 수밖에 없고
인접한로직 서버의 캐릭터가 다른 서버로 전달될 때 스테일 문제 또는 블로킹 문제가 발생하기도 한다.
실제 상용화된 서비스에서는 서비스 안정성에 관련된 문제가 될 수도 있다.

하지만 이음새 없는 거대한 월드 안에 동시접속자가 수만 명 이상 있어야 하는
온라인 게임이라면 심리스 MMO 서버 구조가 유일한 해결책이다.

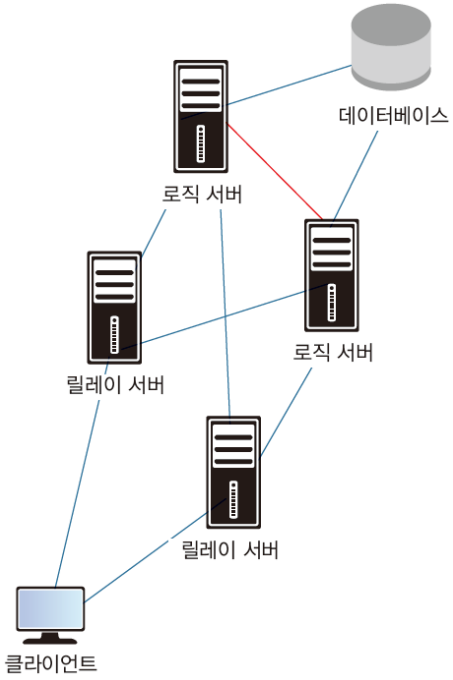


그림 10-43 서버와 로직 서버로 구별

10.7 | 게임 장르별 분산 서버 형태

- 방 만드는 게임(MOBA, FPS, MMORPG)에서 서버를 구성하기

  수평 확장된 로비 서버가 있고, 여기서 매치메이킹을 담당한다.
  그리고 다수의 게임 플레이방을 담당하는 배틀 서버를 구성한다.

  각 배틀 서버는 서로 상호 작용하는 일이 거의 없다. 게임 서버뿐만 아니라 DB 서버도 수평 확장한다.

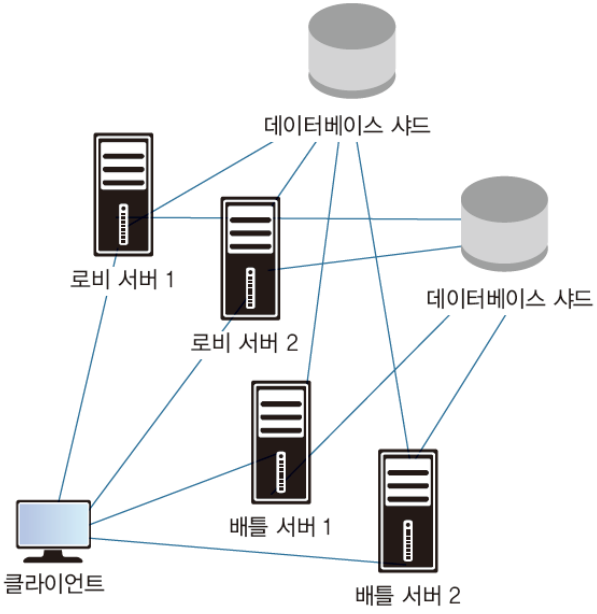


그림 10-44 MOBA, FPS, MMORPG 게임에서 서버 구성 예

10.7 | 게임 장르별 분산 서버 형태

- SNG(소셜 게임)

  게임 플레이 로직을 담당하는 서버와 게임 플레이어의 메모리 내부 정보를 가진 서버가 분리됨.

  게임 플레이 로직만 담당하는 서버는 수평 확장된 로직 서버라고 하고 수평 확장된 메모리 캐시가 로직 서버 뒤에 있다.

  게임 클라이언트는 메모리 캐시에 직접 액세스할 수 없다.

  로직 서버들 뒤에는 수평 확장된 데이터베이스도 있다.

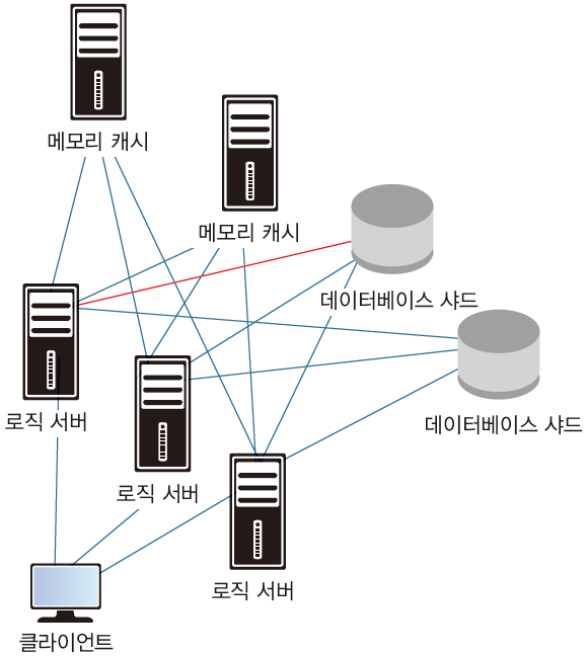


그림 10-45 소셜 게임에서 서버 구성 예

SNG뿐만 아니라 실시간 멀티플레이 대신 비동기 멀티플레이를 하는 모바일 RPG 게임에서도 종종 이렇게 구성한다.
(비동기 멀티플레이 : 플레이어 간 상호 작용이 즉시 일어나지 않는것을 의미.)

경우에 따라 이를 분산 서버 구성이 조합하기도 한다.

SNG나 비동기 멀티플레이 모바일 게임이라고 하더라도 게임 기획 내용에 실시간 대전이나 실시간 공성전이 있다면, SNG 분산 서버 구성과 MMORPG
분산 서버 구성이 뒤섞이기도 한다.

10.8 | 요약 및 결론

게임 장르가 같다는 이유만으로 분산 서버의 형태가 반드시 하나로만 나오지 않는다.

똑같은 게임 장르라고 하더라도 플레이어가 전투를 하는 방식 또는 플레이어와 플레이어 간 커뮤니티 관련 게임 기획 내용에 따라, 이상적인 분산 서버의 형태는 달라질 수 있다.

이를 무시할 경우 자칫하면 분산할 필요가 없는 부분을 분산하거나 정작 분산해야 하는 것을 분산하지 않는 실수를 할 수 있다.

이 장에서 알아본 것은 일종의 모범 답안으로 이 모범 답안을 기초로 삼되, 9장에서 익힌 분산서버 설계 지침을 응용하여 게임 서버의 분산 설계를 하기 바람.