보드 게임 서버 모델

보드 게임은 멀티플레이어 게임 구조를 갖는 게임으로 캐주얼 게임의 서버 특성을 갖는다. 캐주얼 게임은 플레이 타임이 잛고 적은 사용자와 매칭된 공간 (보통, 방이라 부름)에서 플레이를 진행한다. 방은 게임에 따라 유지될 수도 있고 게임 완료 후 자동 소멸될 수도 있다. 게임의 특성에 따라 다르게 기획된다.

보드 게임은 서버에서 판단하고 게임을 진행시켜야 하므로 서버 주도형 게임이며, 대부분 사용자가 교대로 플레이하므로 턴 방식 게임이다.

요약하면, 방에서 턴방식으로 서버 주도로 진행하는 멀티플레이어 게임이다.

주요 과제

보드 게임은 다수의 사용자들이 함께 플레이 해야 한다. 이는 MMORPG처럼 월드 단위의 구분된 공간에서 플레이를 하지 않고 서로 만날 수 있는 공간이 매우 넓다는 뜻이다. 따라서, 확장성이 중요하다. 사용자 연결을 처리하는 서비스, 매칭과 방을 관리하는 서비스, DB 서비스의 구조에 따라 제약을 받는다. 따라서, 이 부분에 대한 고려가 있어야 한다.

매칭

함께 플레이를 할만한 사용자들을 모아주는 기능이 시작할 때 중요하다. 서로 함께 할만한 사용자들을 판단하는 방법은 다양하다. 이를 매칭 기능이라고 한다.

모바일 게임은 "피망, 가지노 로열"이 대표적인 형태로 보유 돈을 기준으로 배팅할 수 있는 구간별로 매칭하는 방식을 사용한다. 도박이 아닌 게임들은 사용자들의 그룹을 좀 더 크게 나눠서 빠른 매칭이 되도록 해도 괜찮다.

매칭은 기획에 따라 다르게 구현된다. 그래도 몇 개의 외부 / 내부적인 그룹을 나누고 해당 그룹의 매칭을 서비스로 제공하고 여러 개의 매칭 서비스를 두는 방식으로 처리 가능하다. 방에 다시 진입이 가능한 게임일 경우 매칭 서비스에서 방 목록을 함께 관리한다.

DB 샤딩

다른 멀티플레이어 게임에 비해 기술적인 난이도가 높지는 않으나 DB 처리는 매칭과 함께 예외적인 부분이다. 사용자가 많으면 단일 DB로는 서비스를 제공하기 어렵기 때문에 DB를 여러 개 쓸 수 있어야 한다. 주로 사용하는 방법은 계정이나 캐릭터를 기준으로 해시 함수를 통해 DB를 선택하는 방식을 사용한다.

해시를 사용할 경우 DB 확장 시 DB를 변경하는 과정이 필요하기 때문에 계정 DB에 할당된 DB를 명시적으로 갖고 처리하는 방법도 있다.

MMORPG와 같이 DB 처리 전용 서버를 두는 방식은 장애 범위가 크고 병목이 될 가능성이 있으므로 개별 서버에서 DB에 직접 연결하는 방식을 사용하는 것이 낫다.

DB 샤딩을 하면 사용자간 트랜잭션 처리가 어려우므로 사용자간 트랜잭션은 메일이나 선물과 같은 기능을 사용하여 처리하도록 기획하는 것이 좋다.

단선 / 재접속

캐주얼 게임이나 전략 게임들은 질 것 같으면 도망 가능 경우가 많다. 모바일에서는 망 전환으로 인해 단선이 빈번하게 일어난다. 이런 상황을 고려하여 재접속 시간을 주고, 단선 시 패널티도 어느 정도 부여해야 선의의 피해를 막을 수 있다.

구현은 단선 시 로그아웃 기능이 있다면 제거하고, 빠른 인증과 현재 방 진입으로 구현하며, 단선 후 일정 시간이 지날 경우 게임에 대한 처리를 포함하여 로그아웃 시킨다.

아키텍처

프로그래밍은 아트의 성격을 갖는다. 80%가 아트고 20%가 과학이라고 얘기할 수도 있다. 그만큼 사람마다 구조와 구현 방법의 선택이 다양하다.

프론트 / 백엔드 서비스 구조

프론트에서 인증과 정보 전달을 담당한다. 이를 로비 서비스라고 한다. 백엔드에 매칭, 게임 서비스를 둔다. 프론트 서버의 목록은 클라이언트에서 웹을 통해 받거나 패치 시 전달 받는다. 서버 목록에서 임의로 선택하여 연결하고, 연결이 안 될 경우 다음 서버로 넘어가도록 하여 분산 시킨다.

백엔드에 둘 서비스는 선택할 수 있다. 매칭만 둘 수도 있고 게임을 실행하는 게임 서비스들도 백엔드에 둘 수 있다. 상점, 이벤트 / 미션과 같이 자잘한 작은 서비스는 프론트 / 백엔드를 선택할 수 있다.

장애 처리는 로비 서비스, 게임 서비스에서 매칭 서비스의 상태 변화를 감지하여 처리한다. 진행 중인 매칭은 다시 시작하게 한다. 진행 중인 게임은 유지되는 기획일 경우 상태를 알려준다.

게임의 진행 구조

Room, Game을 기반 클래스로 하고 각 하위 클래스를 만들어 처리 가능하도록 한다. 게임마다 세부적인 부분이 다를 수 있어 확장 가능하게 구현해야 한다. Room은 매치 상태를 받아 시작하고 사용자들이 들어오면 게임을 생성하여 진행시킨다. Game은 여러 가지 다른 종류를 선택 가능해야 한다. Room가 Game은 이벤트로만 통신하는 것이 좋다.

플레이어 정보를 게임 실행 시마다 DB에서 얻어 오기 보다는 게임 진입 시 로비 서비스에서 사용자 정보를 갖고 진입하는 것이 좋다. DB 캐시를 redis 등을 사용하는 경우도 있다. DB 캐싱은 여러 가지 단점이 있기 때문에 잘 생각하고 판단해야 한다.

보드 게임은 턴 방식이고 턴이 다양한 조건을 따라 변경되며 각 턴에 할 수 있는 일도 다양하다. 턴은 턴 실행의 결과에 따라 선택된다. 포커 같이 항상 다음 사람에게 턴이 넘어가는 경우도 있고, 마작이나 블루마블처럼 턴이 건너뛰거나 유지되는 경우도 있다.

특정 상태에서 실행한 동작은 이벤트를 발생 시키며 이벤트 결과에 따라 다음 턴이 결정된다. 이 구조가 설계에 반영되어야 한다. State를 몇 개로 나누고, Action을 클래스화 하고 하위 클래스들에서 동작을 만들고, Action은 State에 이벤트를 보내고, 이 이벤트에 따라 Turn과 Action을 결정한다.

포커, 마작, 블루마블, 고스톱 모두 보드 상의 위치 개념이 있다. 포커는 딜러, 플레이어의 순서, 베팅 머니를 놓는 곳 등이 있다. 블루마블도 플레이어의 위치, 보드의 국가 위치, 말의 위치를 갖는다. 각 위치와 상태에 따라 여러 가지 이벤트가 발생한다.

Turn, 위치 모두 이벤트 / 행위를 중요한 개념으로 추출했다. 상태와 행위에 따라 이벤트가 발생하며 이벤트는 다음 동작과 상태를 결정한다.

턴방식 보드 게임에서는 타이머의 처리가 중요하다. 플레이를 하지 않는 경우 강제로 주사위를 굴린다던지 하는 행위는 서버 주도로 진행되어야 한다. 행위는 클라이언트에 표현하는 시간을 갖는다. 이들 시간 값을 데이터화 하여 (코드 상의 데이터라도) 타이머로 처리해야 한다. 클라이언트와 서버 간 시간 차이가 있기 때문에 어색해지지 않도록 맞춰야 한다.

게임의 상태는 다음과 같이 볼 수 있다. 기획에 따라 추가 단계가 있을 수 있다.

```
Setup,
Play,
Pause,
End
```

Pause 상태는 특정 플레이어가 단선이 발생할 경우 일정 시간 유지할 수 있다. 재진입 시 현재 게임 상태를 전달하고 다시 진행할 수 있다. 기획에 따라 그냥 내보내고 사용자를 AI로 교체하고 진행할 수 있다.

게임은 Turn, Action, Event로 진행된다. Event는 클라이언트 이벤트를 포함한다. Action과 Event 흐름의 예시이다.

```
On: turn start
    Do: send turn start to clients

On: dice rolled from client,
    Do: send dice result to client,

On: player arrived to a block from client,
    When: block is empty
        Do: Ask building options
    When: block is occupied
        Do: Ask purchase of the building

On: reply from client
    Do: process the result
    When: dice is double.
        Do: emit turn start
    Else:
        Do: change current player
        Do: emit turn start
```

emit은 서버 내 이벤트 발생으로 본다. On은 이벤트에 대한 반응 액션이다. When은 조건 체크이다. Else는 When이 아닌 경우이다.

더 상세한 구조는 기획에 많이 의존한다. 위의 추상화로 기획 내용을 잘 수용할 수 있는 지 확인한다.

DB 처리

게임의 결과 처리 등 DB에 저장할 경우 각 서버에서 진행하고 로비의 사용자에게도 알려준다. 게임 서비스가 백엔드에 있을 경우 필수적으로 로비 서비스를 통해 전달된다. 게임 서비스도 프론트에 있을 경우 게임 서버를 통해 전달하고 로비는 갱신만 받을 수 있다.

DB 처리는 비동기로 실행되어야 하고, 로드가 높을 경우 쓰레드를 늘려서 처리하는 방법이 구비되어 있어야 한다.

비동기 처리

DB, 로깅, 웹 호출 등은 블록킹 되는 처리들이다. 블록킹 되는 처리는 해당 서버나 서비스의 요청이 완료할 때까지 대기를 하는 요청들로 그냥 사용하면 해당 쓰레드가 요청이 완료될 때까지 아무런 처리를 하지 못 한다. 따라서, 이들 호출은 비동기로 만들어야 한다.

비동기로 구현하는 방법은 별도의 쓰레드 풀을 만들고 해당 쓰레드 풀로 큐를 통해 요청을 전달하는 것이다. 요청이 완료되면 비동기 처리 쓰레드 풀에서 완료 시키거나 다른 쓰레드들로 메세지를 전달하면 된다. 이렇게 큐와 메시지를 통한 처리를 메시징 처리라고 한다. 대표적인 모델이 MPI(OpenMPI 구현 참조)와 Actor 모델 (AKKA 참조)이 있다.

프라우드넷 사례

프라우드넷은 여러 프로젝트에서 오랫동안 사용하여 안정성이 검증된 윈도우 상의 RPC 기반 서버이다. 서버로 오는 RPC 요청은 IOCP 쓰레드 상에서 처리된다. 보통 코어 갯수만큼 쓰레드를 생성하여 실행한다.

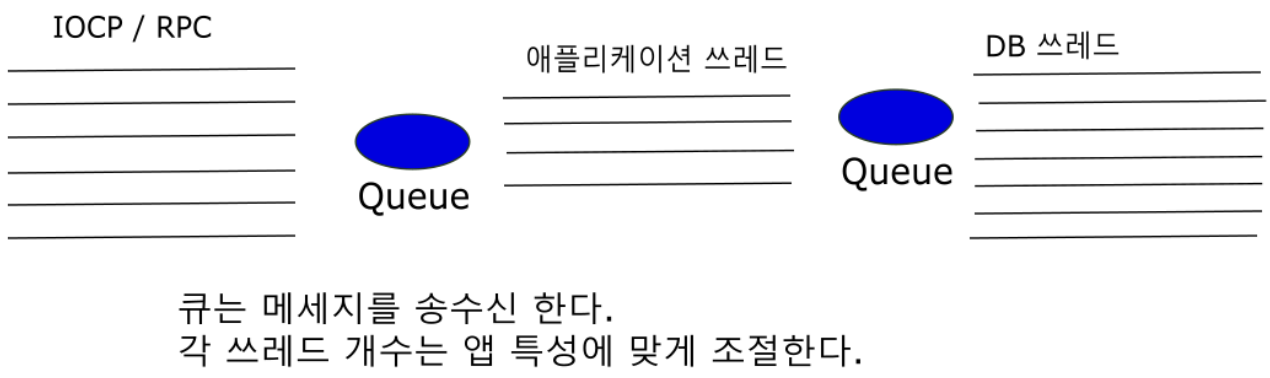
두가지 유의해야 할 점이 있다.

- 쓰레드 개수가 제한되어 있으므로 블럭킹 처리를 비동기 처리로 분리해야 한다.
- 호출 쓰레드가 매번 바뀌므로 락 처리가 필요하다.

블럭킹 처리는 C# 서버의 경우 Task를 사용하여 분리할 수 있다. 호출 쓰레드가 바뀌는 문제는 처리하는 Task나 쓰레드를 애플리케이션에서 생성하여 하나의 쓰레드에서만 처리하게 하는 방법이 있다.

락을 사용하면 블럭킹 처리와 동일한 문제가 발생하는데 주의 깊게 락 처리를 구현하면 서버로 사용 가능한 정도의 성능을 얻을 수도 있으나 매우 불편하므로 메세징 모델을 사용하는 것이 좋다.

대략의 모델



프라우드넷의 경우 RPC 함수 호출 전에 메세지 형태로 있으므로 이를 사용해서 큐를 통해 전송하거나 람다 함수를 메세지에 포함해서 전달해도 된다.

DB 쓰레드에서 처리 요청 / 결과를 애플리케이션 쓰레드와 함께 동작할 메세지 클래스를 설계하고 진행하는 것이 좋다.

애플리케이션 쓰레드는 기능별 / 데이터 별로 구분해서 처리하는 것이 좋다. 락을 거는 것보다 락 없이 큐를 통하는 것이 좋다.

큐는 ConcurrentQueue를 사용하면 된다. Lockless로 구현된 큐로 .NET의 C# / Visual C++에서 모두 사용 가능하다.