Architecture of Instance Based Games

YoungHand

2017년 8월 12일

차 례

1 인스턴스 게임 종류와 특징

플레이 하는 공간이 동적으로 생성 / 소멸되는 것이 가장 큰 특징이다. 전형적인 인스턴스 기반의 게임들은 다음과 같다.

- 액션 RPG
- 전략 RPG
- FPS 게임
- 보드 게임
- 캐주얼 게임
- 전략 게임

고정된 공간으로 보이는 마을과 같은 공간도 서버 시작 시 생성되고 서버 종료 시 소멸되는 인스턴스 공간으로 보고 동적인 인스턴스와 동일하게 구현할 수 있다.

인스턴스 기반의 게임들은 MMORPG나 대규모 모바일 전략 게임을 제외하면 대부분 여기에 해당된다고 할 정도로 다양한 게임이 사용하는 모델이다.

1.1 인스턴스 생명 주기

인스턴스는 생성 / 사용 / 소멸의 과정을 밟는다. 생성된 인스턴스의 관리 서버와 실행 서버가 다를 경우 타이밍 관련 이슈가 많다. 분산된 인스턴스 실행이 필수적인 경우가 대부분으로 서버 간의 동기화와 진입 / 진출 프로토콜이 잘 구현되어야 한다.

서버 N_1의 I_1 인스턴스가 상태 $Active$에서 $Destroying$으로 전환되고 정리 절차를 밟을 때 클라이언트에서 진입 이벤트 요청이 I_1으로 전달된다면 예외 처리 코드가 추가 되어야 한다.

이런 상황에 일일이 대처하기 보다는 생명 주기를 관리하는 곳에서 가장 먼저 상태 변경이 이루어지도록 전체 구조와 프로토콜을 설계 해야 한다.

이와 같은 미묘한 인스턴스 생명 주기 관리 문제를 분산 처리와 함께 잘 고려해서 아키텍처를 결정해야 한다.

1.2 목록 관리

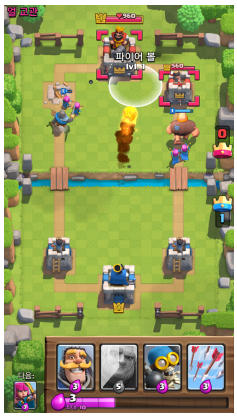

그림 1: 클래시로얄

클래시 로얄은 모바일 메가히트 게임으로 실시간 대전 방식의 게임 진행을 갖는 전형적인 인스턴스 기반 게임이다. 클래시 로얄과 같이 한번 매칭되고 게임이 끝나면 해당 인스턴스를 참조할 경우가 없다면 바로 소멸시킬 수 있지만 대부분의 게임은 해당 인스턴스에서 플레이가 지속되는 경우가 많다.

이와 다른 게임의 예로 포커 게임들이 있는데 해당 방이 유지되면서 사용자들이 진출입을 선택해서 진행할 수 있다. 매칭할 때도 적합한 조건의 인스턴스를 찾아서 들어가는 방식이다.

목록 관리는 분산 디렉토리 서비스에 필요한 기능을 부분적으로 요구한다. 분산 디렉토리는 데이터 자체만 갖고 있지만 목록 관리는 기능 처리를 요구한다.

목록 관리는 목록에 대한 게임 내 기능과 연관된다. 중요한 용례는 다음과 같다.

- 목록 조회
- 진입과 진출
- 매칭

목록 조회는 최근의 모바일 게임에는 불필요할 수도 있는데 매칭과 진출입을 위해 필요한 검색 서비스의 일종으로 본다면 여전히 중요한 요구 사항이다.

1.3 진입과 진출

분산된 시스템이 아니라면 단순한 문제지만 서버들이 기능별로 나누어지거나 서버 자체를 추가하거나 내리거나 할 수 있는 구조라면 매우 세심하게 구현해야 한다.

클라우드와 컨테이너 구조를 활용하여 동적으로 확장하고 불필요할 경우 축소하는 게 자유롭다면 비용도 매우 효율적으로 관리가 가능하기 때문에 앞으로 더 중요해질 것이다.

1.4 매칭

매칭은 두 가지 형태로 주로 진행된다. 하나는 사용자 전체 집합에 대한 조건 검색 후 인스턴스를 생성하고 진입 시키고 게임을 진행 시키는 형태가 있다. 다른 경우는 인스턴스 집합에 대해 조건 검색하고 진입 시키는 방법이 있다.

프로토콜을 일련의 작업을 달성하기 위한 연산의 요청 관점으로 보고 연산의 그룹을 단계로 나눈다면 매칭은 검색, 생성, 진입 단계로 나눌 수 있다.

2 서비스 특징

서비스 플래폼과 게임마다 차이가 크지만 대부분의 게임에 필요한 것들을 살펴보고 수용 가능하도록 구조를 잡아야 한다.

2.1 단일 월드

먼저 거의 모든 인스턴스 기반 게임들은 하나의 월드로 서비스 된다. 하나의 월드라는 의미는 사용자들이 모두 만나서 플레이가 가능한 게임이라는 뜻이다. 내부적으로 어느 정도 구분된 구조가 있다고 하더라도 절대적으로 넘을 수 없는 경계가 없는 경우가 대부분이다.

단일 월드 구조는 서비스 규모가 매우 커질 수 있다. 2명이 주로 플레이하는 게임이라고 하면 1백만 동접을 지원할 때 50만개의 인스턴스가 생긴다. DB 규모도 훨씬 커질 수 있다.

2.2 장기간 실행, 짧은 정기점검

인스턴스 기반 게임들은 게임성에 있어 항상 플레수를 하고 있지는 않지만 원할 때 플레이가 되어야 한다. 따라서, 긴 시간의 유지보수 시간이나 다운 타임은 게임에 부정적인 영향을 미친다. 서비스 유지보수를 위한 정기점검 시간도 짧을수록 좋다.

2.3 서비스 관리

서비스 관리는 인스턴스 기반 게임에만 필요한 건 아니지만 위의 목표를 달성하기 위해 중요하다. 서비스 관리 구조는 **lax** 상에서 처음이기 때문에 여기서 정리하도록 한다.

서비스 관리에 필요한 기능은 상당히 많지만 필수적인 기능은 다음과 같다.

- **모니터링** 서버와 액터들의 상태를 조회 가능하다.
- **상태수정** 액터들의 상태를 변경할 수 있다.
- **설치/패치** 서버의 바이너리와 데이터를 배포할 수 있다.
- **로그 수집과 분석** 서버의 시스템 로그와 데이터 로그를 모으고 분석한다.

서비스 관리 기능은 lax service controller인 laxcon과 remote shell 기능을 담당할 laxshell에 기반하여 구현할 계획이다. laxcon은 파일 패치와 바이너리 실행과 재실행을 하는 프로세스 모니터이자 패치 프로그램이다. laxshell은 actor의 계층 구조를 따라 json으로 조회와 상태 수정을 하는 shell이다.

로그 수집과 분석은 elasti stack이 유용하고 대규모 데이터도 spark과 연계하면 분석이 가능하다.

3 요구사항

용례 중심으로 요구 사항을 정리한다.

게임

- **인증** : 로그인과 로그아웃. 중복 로그인 처리. 단선시 빠른 재접속.
- **매칭** : 사용자 기반의 매칭.
- **참관** : 참관을 위해 인스턴스 목록 검색 가능.
- **플레이** : 어떤 게임인지 모르나 일정 시간 플레이 진행.

서비스

- **설치/패치** : laxcon만 있으면 자동으로 설치 / 패치
- **모니터링** : laxshell을 통해 시스템 상태와 액터들 상태 조회. 사용자 / 인스턴스 목록 조회.
- **상태수정** : json 명령으로 laxshell을 통해 상태 수정.
- **장애복구** : 특정 서버가 크래시 났을 때 해당 서비스를 복구.

4 구성 요소

lax는 액터 단위로 구성되고 cluster actor 들이 서비스의 뼈대를 구성하므로 cluster actor들을 파악하고 이들의 그래프 구조를 만드는 것으로 시작한다.

4.1 설계 iteration 1

게임 기능을 처리하기 위해 필수적인 요소들을 정리하면 아래와 같다.

클러스터 액터들

- **user keeper** 사용자들에 대한 북 키핑을 한다.
- **gate** 사용자들에 대한 연결을 처리하고 인스턴스 실행도 담당한다.
- **instance keeper** 인스턴스 목록을 관리한다.
- **matcher** 사용자 기반으로 매칭을 처리한다.

이와 같이 정리하고 보면 몇 가지 의문이 든다. gate는 여러 개를 두고 분산 처리가 가능하지만 각 keeper와 matcher는 수백만에서 수천만을 처리한다고 하면 성능이 일단 문제가 될 수 있고 단일 장애 지점이 된다. 성능을 개선하고 장애에 어떻게 대응할 것인지 고려해야 한다.

4.2 설계 iteration 2

성능을 개선하기 위해 각 keeper들을 분산 시킨다. 기본 아이디어는 특정 키 값에 따라 분할 하는 것이다.

사용자는 주로 검색이 이루어지는 이름(닉네임/별명/캐릭터명)으로 분산시킨다. 해시 값에 대한 분산으로 처리하면 된다. 중복 로그인 처리도 된다.

user keeper는 hash 값을 int로 하고 범위에 따라 분산 시킨다. (min, max) 값의 범위를 정적으로 구성하여 할당할 수 있다.

instance keeper는 좀 더 복잡하다. 용례를 명확하게 해야 좀 더 단순한 구조를 택할 수 있다. 가장 복잡한 구조는 안전하게 분산된 검색 저장소로 만드는 것인데 elasticsearch에서 잘 구현하고 있는 구조이다. 게임에서는 배보다 배꼽이 커지는 것으로 볼 수 있다.

instance keeper 용례

1. 특정 사용자가 플레이하는 곳으로 가서 게임을 보고 싶다.
2. 랭킹전 진행 중인 인스턴스들을 보고 선택해서 들어가고 싶다.
3. 접속이 끊기고 다시 들어왔는데 어느 인스턴스로 가야할 지 판단해야 한다.

1, 3은 user keeper에서 처리할 수 있다. 2번과 함께 특정 조건의 인스턴스 전체를 빠르게 검색해야 할 경우가 아니라면 특별히 인스턴스를 따로 관리할 필요는 없다.

조건에 따라 그룹을 만들고 각 instance keeper에 그룹들을 할당해서 검색이 해당 그룹을 시작으로 이루어지는 걸로 한정할 수 있다면 그룹 할당이 괜찮아 보인다. 주로 채널이라고 말하므로 채널로 용어는 사용한다.

instance keeper 별로 채널을 할당하고 채널을 배정 받아 keeper를 선택하도록 한다. instance keeper를 선택할 때 최상위 조건은 channel이 된다.

matcher도 그룹별 할당을 한다. 그룹의 구분은 매칭하고자 하는 대상군으로 나눈다. 결국은 한 그룹으로 몰릴 수 있는데 그룹의 동적 분할과 matcher의 추가 생성을 시도한다. 동적 분할 시 클러스터 전체에 안정적으로 알리고 이후 진행이 될 수 있도록 해야 한다. 하위 matcher를 두고 분할하는 방식으로 진행한다. 억 단위 요청이 오더라도 몇 개의 matcher면 충분할 것으로 보인다.

이제 확장성은 분할을 통해 해결된 것으로 보인다. 정적 분할과 동적 분할을 적절하게 활용해야 하고 동적 분할은 프로토콜 설계가 잘 돼야 한다.

gate도 잠시 살펴보야 한다. 게임 인스턴스를 실행하므로 다른 gate에 있는 사용자들이 통신을 위해 새로 연결을 맺어야 한다. 먼저 연결을 맺고 해당 인스턴스에 진입을 진행한다. 연결을 맺는데 실패하거나 클라이언트 오류가 나거나 하면 모바일 게임을 지원해야 하므로 timeout으로 실패 처리한다.

4.3 설계 iteration 3

이제 기본 기능에 필요한 액터들은 찾았고 정적/동적 분할의 개념으로 확장성도 확보했다.

각 액터가 있는 장비에 문제가 생기거나 버그로 인해 크래시가 나거나 네트워크 연결이 끊어지는 장애가 발생했을 때 견딜 수 있게 해야 한다.

각 장애 지점에 대해 살펴보고 어떤 방법이 장애를 극복할 수 있는 지 살펴본다.

user keeper 특정 keeper가 사용 불가능하게 되었을 경우 해당 hash 값을 처리하는 새로운 user keeper를 실행하면 새로 진입하는 사용자들은 처리 가능하게 된다. 기존 사용자들은 필요한 시점에 인증을 추가로 받는 것으로 해결 가능해 보인다. 인증 정보를 DB에 저장해 두고 처리해야 기존 사용자들의 인증 정보를 복원할 수 있다.

instance keeper 새로운 instance keeper를 실행하고 각 gate를 통해서 해당 인스턴스의 정보를 복원한다. 복원 자체는 선택적일 수 있다. 복원 과정은 instance keeper 쪽에서 gate들에 요청할 수도 있고 gate 들에서 보내 주는 방식을 택할 수도 있다.

matcher 해당 matcher를 다시 실행 시킨다. 모든 gate에 알려서 해당 matcher를 통해 대기중 매칭은 모두 취소해야 한다.

gate gate의 장애는 해당 gate에 접속한 사용자들을 일정시간 대기하고 instance keeper에서는 정보를 삭제한다. 그걸로 대부분 처리된다.

장애의 판단 이론적으로 여러 가지 복잡한 문제들이 관여한다. Split brain 문제는 cluster actor가 중요한 액터들의 상태 동기화를 받으므로 여기에 기초해서 장애로 판단하도록 구현한다.

4.4 설계 iteration 4

4.4.1 분산 디렉토리 애플리케션 서버

user keeper, instance keeper는 지금까지 따라온 바와 같이 정적/동작 분할이 필요하고 (분산 디렉토리) 동작을 수행할 수 있는 서버이다. 분산 디렉토리는 LDAP이나 Active Directory 같은 것들이 있고 분산 데이터베이스는 NoSQL들이 다양하게 있다. NoSQL의 분산 기능이 더 유사할 수 있는데 NoSQL DB들은 일반 RDBMS와 달리 로직을 추가하는 기능이 없다.

일반화 시켜서 구현하려면 노력이 많이 필요하겠지만 replication을 추가로 포함해서 구현해두면 완벽한 서버를 만들 수 있을 것 같다.

4.4.2 Database

별도의 캐싱이 없는 database 접근이 필요하며 각 액터가 직접 접근하는 방식을 택한다. 별도의 DB 연결을 처리하는 노드와 액터들을 두는 것만으로도 persistency 관련 여러 가지 논리적인 불가능성에 직면하게 된다.

예를 들어, 해당 에이전트의 연결이 끊어질 경우 기존에 보낸 요청들의 성공과 실패를 구분할 수 없게 된다. 이런 문제는 대부분의 게임에서 치명적이다.

SSD를 디스크로 사용하고 RAID 6로 묶으면 빠르고 안정적인 RDBMS를 얻을 수 있다. 사용자별로 정적인 sharding을 구현하고 트랜잭션은 큐 테이블을 두거나 애플리케이션에서 처리하는 방식으로 구현한다.

4.5 정리

keeper에서 dir로 이름을 변경한다. user dir, instance dir, gate, matcher를 핵심 요소로 하고 각 노드에 데이터베이스를 처리하는 액터를 둔다.

5 Deployment

중요한 목표 품질이 있다.

- 장애 범위를 최소화 한다.
- 확장성이 보장 되어야 하고 서비스 중 확장이 가능해야 한다.
- 구성 단위로 유지보수가 가능해야 한다.
- more to add here

5.1 성능

플래폼이 윈도우가 익숙하여 윈도우를 대상으로 진행하려고 했으나 클라우드나 사설망 모두 VM이나 컨테이너 상에서 동작하고 윈도우는 컨테이너 지원이 성숙하지 않아 Linux를 서비스 플래폼으로 한다.

PC에서 개발할 때 보다 VM 상에서 성능이 더 안 나올 때가 많아 어쩔 수 없는 선택이다.

5.2 장애 범위의 최소화

장애를 최소화 하고 유지보수 단위를 작게 만들려면 사용자 세션을 처리하는 부분과 서비스를 제공하는 단위가 분리 되는 것이 더 낫다.

사용자 연결은 외부에서 들어오고 다양한 공격도 사용자 연결을 통해 이루어진다. 따라서, 해당 서버는 장애가 좀 더 빈번하게 발생한다. 중요한 서비스는 뒷단의 좀 더 안전한 환경에 두면 장애를 줄일 수 있다.

상태 공간이 큰 기능은 버그와 장애가 더 많이 있을 수 있는데 이들은 장애가 발생하더라도 복구 가능하게 하고 단위 추가도 쉬워야 한다.

5.2.1 Gate와 HAProxy

Gate들은 기능이 동일하기 때문에 로드밸런싱이 session 수준에서 가능하다. 하프록시가 오랜 기간 사용되고 안정성도 높고 성능도 좋아 적합한 선택이 된다.

Gate 앞 단에 HAProxy들을 두고 Gate와 통신을 중계하도록 해서 특정 Gate가 유실되더라도 적합한 노드를 찾아 서비스가 유지되도록 한다.

Gate들은 기능이 동일하고 프로토콜 상의 비대칭성이 없도록 해야 한다.

5.3 확장성

반응 속도가 중요한 서비스와 사용자 규모에 따라 확장이 필요한 기능 모두 확장성을 고려해야 한다. 확장은 scale-out으로 가능해야 한다.

앞단의 서비스들은 접속한 사용자들과 연관된 상태만 갖고 장애 범위가 접속된 사용자들로 제한 되어야 한다.

따라서, 한 프로세스에서 많은 사용자를 처리하기 보다는 다수 프로세스에서 사용자들을 나누어 처리하는 것이 낫고 필요시 서비스 중에 추가가 가능해야 한다.

뒷단의 중요 서비스들도 동적으로 분할이 가능하면 좋은데 아직 적합한 알고리즘과 구현이 확정되지는 않은 것 같다.

active-standby 형태의 복제가 최소한의 기능이고 active-active 구성이 자유롭게 확장 가능한 것이 이상적이다. 백엔드 서비스들은 대규모 데이터에 대한 서비스를 제공하므로 이들 데이터를 동적으로 분할해서 각 노드에 할당하는 것이 핵심이다. 안정적인 분산 알고리즘을 찾고 복제와 함께 잘 구현하면 가장 큰 경쟁력이 될 수 있다.

이름을 replicated and dynamically partitioned data and logic service로 지으면 어떤 방향인 지 명확해 보인다. 동적으로 분할되고 복제된 분산 데이터 서비스는 카우치베이스와 같은 NoSQL에서 먼저 지원하고 있다. 이 데이터 상의 애플리케이션 논리를 구현하는 기능은 아직 없다.

분산된 데이터 상에 논리 구현은 생각보다 여러 가지 어려운 문제들을 포함할 것 같고 이들을 살펴서 어디까지 가능할 지를 보고 알고리즘을 정해서 단계적으로 쌓아 가면서 구현한다.

이들 개념을 dir<user>, dir<instance> 형태의 generic container로 만들 수 있다면 최상이다.

5.4 구성 단위 유지보수

문제가 발생한 컴포넌트만 유지보수 할 수 있다면 서비스 다운타임을 최소화 할 수 있다.

프로토콜 변경과 같이 어려 노드의 여러 컴포넌트에 영향을 미치는 것까지 나눌 수는 없겠지만 이 부분도 어디까지 가능한 지는 살펴볼 필요가 있따.

분리해서 유지보수가 가능하려면 **확장성**에서 언급한 기능들이 동시에 구현되어야 한다. 모든 것이 서로 연결된 측면을 갖고 있어 개념화가 더 이루어져야 한다.

5.5 보안

5.6 운영

6 구현의 설계

구현이 쉬운 순서에서 어려운 순서로 진행한다.

6.1 gate

노드 당 하나만 실행. 다른 액터들에서는 $(node, gateid, userid)$로 통신을 요청.

사용자 op

- 게이트 목록 : 다음 번 게이트 확장이나 축소를 위해 전달. 클라 저장.
- 인증 요청 : 로그인 요청
- 로그아웃 : 로그아웃 요청. 타임아웃 / 재진입 처리 효율을 위해 사용.
- 매칭 : 조건에 따른 매칭 요청
- 인스턴스 목록 : 채널 + 조건으로 요청
- 참관 : 특정 인스턴스 진입 요청

- 진입 / 진출 : 요청을 받아 dir<instance>를 통해 처리. 통로만 된다.

최초 접속한 게이트와 이동하는 게이트의 차이가 없도록 구현해야 장애 범위를 최소화 할 수 있다.

매칭은 어느 게이트 액터인지와 누구랑 매칭되는 지만 알려주고 실제 인스턴스 생성과 진입은 해당 게이트를 통해서 분리해서 처리한다.

6.1.1 runner<instance>

gate에서 부하 분산을 위해 선택해서 요청한다. instance runner는 cluster actor가 아닌 것으로 한다. instance runner는 instance들을 갖는다.

generic class는 아닐 수도 있지만 일반화를 지속적으로 추구한다는 의미로 runner<instance>를 사용한다.

6.2 matcher

6.3 dir<user>

replication, 동적 / 정적 분할을 갖춘 directory가 되도록 최대한 구현. 개별 구현하고 추상화 가능한 지 판단한 후 진행한다.

정적 분할로 하고 사용자 nick(name)의 hash 값을 사용하여 분할한다. DB에 인증 정보를 저장한다.

operations

- 로그인
- 인증 키 조회
- 위치 조회
- 로그아웃

6.4 dir<instance>

7 과제들

7.1 분산 상태와 동작의 일관성

이론적으로 어떤 용어가 있고 어떻게 정리되었는 지 모른다.

gate에 있는 사용자가 매칭을 요청한다. gate에 있는 사용자가 인스턴스 진입을 요청한다. 이미 매칭을 요청한 상태이므로 인스턴스 진입은 허용되면 안 된다. 이를 어떻게 관리할 것인가?

좀 더 추상화 하면 특정 연산은 두 개 노드 상의 상태를 변경할 수 있다. 두 노드의 상태 변화는 일관성이 유지되어야 한다. 로그처럼 모든 곳에 상태 점검이 들어가면 구현이 어려워진다. 어떤 구현 방법이 적절한 선택일까?

7.2 dir의 구현 가능성

dir<user>, dir<inst>와 같이 일반화된 분산 컨테이너를 구현할 수 있을까? 안정적으로 구현할 수 있을까?

7.2.1 키 기반 데이터 분산

std::hash function

7.2.2 키 기반 복제

mirroring algorithm

7.2.3 키 기반 동작

user / instance의 키로 노드/액터 선

7.2.4 조건 기반 동작

7.2.5 노드의 추가와 삭제

삭제를 견디기

노드 추가

7.2.6 하나의 구조

dir master, dir segs.

배보다 배꼽이 커짐.

클라 처리

8 데이터 분산 리서치

8.1 분산 스토리지

Greenplum, couchbase의 키 기반 분산 알고리즘 정리. 실 구현 조사. C++ / actor 기반의 구현 정리

그린플럼의 세그먼트 복제 참조.

마스터를 가짐