마비노기 영웅전
비동기 통신 및 서버 구조

Jinuk Kim

rein01@gmail.com

마비노기 영웅전 서버

C# 기반의 MMORPG 게임 서버

현재까지 .Net 혹은 JVM 언어로 작성된 가장 큰 규모의 게임 서버

단일 서버 인터페이스

게임 서버 군 전체를 묶어서 하나의 서버처럼 동작

주 게임 요소

1 인에서 최대 8 명까지 하나의 인스턴스 던전에 들어가 일반적인 PvE 전투

네트워크 비동기 통신

목표

- 네트워크 코딩 스트레스를 모든 프로그래머에게 주지 말자
- 하위 레이어의 지식을 지나치게 요구하지 말자

문제

- 네트워크 통신 $\neq$ 비지니스 로직
- 정보가 필요한 시점 $\neq$ 계산이 끝나는 시점
- C++과 C#을 모두 지원해야 한다

방향

시스템 프로그래머와 응용 프로그램 프로그래머의 설계 목표가 상충

적당히 돌아가는 수준에서 끝나게되는 경우가 흔하다

- 네트워크 코드 (자동) 생성
- C#/C++ 상호 운용성 확보
- 논리 흐름에 따라 코드를 작성할 수 있게한다
- 저수준 네트워크 기술 지식을 숨긴다

두 개의 주체

서비스

비동기 네트워크 메시지를 처리하는 쪽

클라이언트

비동기 네트워크 메시지를 보내는 쪽

대략적으로는, 클라이언트가 비동기 메시지(operation)를 보내고, 서비스의 응답을 콜백으로 받아처리한다.

메시지 생성

C# 2.0

Lightweight code generation 사용 (.NET IL 직접 생성)

C# 4.0 이라면 ExpressionTree 로 대체 가능함

C++

VislauStudio 매크로 혹은 하드 코딩으로 작성

후반에는 주로 하드 코딩하게 되었다고 한다

Pros & Cons

Pros

- C# 쪽에서 데이터 포맷만 정의하면 메시지가 생성된다.

Pros & Cons

Pros

- C# 쪽에서 데이터 포맷만 정의하면 메시지가 생성된다.

Cons

- C++ 코드 생산성이 떨어진다.
- C# 형식의 가변성 혹은 generic 형식의 처리 문제

프로토콜 반복기

Client

```
var op = new Operation();
op.onComplete += () => {...};
op.onFail += () => {...};
RequestOperation(op);
```

Server

```
IEnumerable<Object> Run() {
  yield return info.ID;
  yield return info.Name;
}
```

Pros & Cons

Pros

- 클라이언트 코드를 작성할 때, 연관된 코드가 한 자리에 모인다

Pros & Cons

Pros

- 클라이언트 코드를 작성할 때, 연관된 코드가 한 자리에 모인다

Cons

- 개발자들이 enumerator 를 배우는데 드는 비용
- C++ 에서 사용할 수 없다
- 중간 처리가 필요한 비동기 작업이 적기 때문에, 굳이 enumerator를 쓸 이유가 없다

delegate 기반 코드 생성

- **마비노기 영웅전에서 사용한 방법이라고 한다**
- C# delegate 를 IDL 로 사용했다
- Client 용 코드는 C++ 로, Service 용 코드는 CIL 을 생성하게 했다
- Hindely–Milner 타입 추론을 쓴다. 불변성은 ref, 가변성의 경우 공변성은 out, return 을, 반공변성은 in 을 이용한다

Pros & Cons

Pros

- C# IDL 만 가지고 클래스를 자동 생성한다

Pros & Cons

Pros

- C# IDL 만 가지고 클래스를 자동 생성한다

Cons

- 프로토콜의 연관성을 기술할 수 없다
- 클라이언트 / 서버 코드가 대칭성이 없다
- 배경지식이 많이 요구된다
- 유지 보수가 좀 불편하다 (C++ 및 CIL)

C# interface로 코드 생성

1. Client/Service 용 code DOM을 생성한다
2. C++과 CIL 코드를 대칭적으로 생성한다
3. MS asynchronous programming design patterns[1]의 내용을 따라 네이밍 및 제약사항을 지킨다
4. 뻔한 serialize 형식은 C++ 쪽에도 복제해서 기타 작업을 간소화 한다

[1] http://msdn.microsoft.com/en-us/library/ms228969.aspx

Pros & Cons

Pros

- 대부분의 코드를 자동 생성한다
- C++/Common language spec. 양쪽에서 일관성있게 작업한다

Pros & Cons

Pros

- 대부분의 코드를 자동 생성한다
- C++/Common language spec. 양쪽에서 일관성있게 작업한다

Cons

- 상속을 써서 생기는 제약 사항

Issues I

추상화 수준

- '일반화 가능한' 추상화 수준을 선택하지 않았다
- 개략적으로 서버 간 RPC 프레임워크에 해당
- 마비노기 영웅전처럼 전체 서버 군을 작성하는 경우가 아니면 사용하기 힘들 것 같다

Issues II

C# enumerator의 학습 곡선?

- Enumerator는 이미 오래된 추상화 개념. 오히려 언어를 바꾸는 오버헤드는?
- Operation에서 사용한 delegate 역시 다른 언어 – C++0x lambda 등 closure 류의 구현체 – 에서 쉽게 쓸 수 있다
- 2차 시도의 비동기 메시지 처리 형태는 C#이 아니어도 가능

Issues III

C#을 IDL로 고집하는 이유?

- C#은 데이터를 기술하기에는 너무 비대한 언어
- 연산간의 연관성을 기술하기 위해서 너무 복잡한 방식 (Property 류의) 을 쓴다
- 물론 C++ 같은 경우 애초에 boost::spirit 같은걸 쓰면 이해하기 힘들었겠지만...

자이언트 서버 구조

왜 ?

- 보통의 MMORPG 서버보다 훨씬 많은 수의 유저를 받을 수 있는 서버
- 단일 서버[2]
- 큰 게임 내 커뮤니티 + 신규 유저 장벽 해결
- 운영 비용 줄이기
- *기술적 도전 과제*

[2]개발 초기 기획에는 MMO 요소가 없었다

Goal

- 추가 구현이나 디자인 변경없이
- 서버를 추가할 때 마다 성능이 올라가게 (horizontal scaling)
- 개별 서버가 유저에게 노출되지 않는다
- 특정 서비스, 서버가 병목이 되지 않는다

Architecture

- 가능한한 작은 기능 단위를 쪼개서 이를 서비스로 구현
- 서비스를 연결하는 통합 네트워크
- 이 서비스들을 연결하는 통합 네트워크
- 서비스의 위치를 추상화하기 위한 위치 서비스
- Operation: 성공 / 실패에 대한 콜백이 달린 서비스의 기능 단위를 제공

통합 네트워크

통합 네트워크

- TCP 위에 메시지 파이프 구현

통합 네트워크

- TCP 위에 메시지 파이프 구현
- 메시지 계층 위에 Operation을 처리하는 계층 구현

통합 네트워크

- TCP 위에 메시지 파이프 구현
- 메시지 계층 위에 Operation을 처리하는 계층 구현
- 위치 서비스를 통해 올바른 서비스에게 메시지 전송

사내 테스트

서비스들

Frontend, Character, Item, Quest, Party, Story,

Microplay, Postal

사내 테스트

서비스들

Frontend, Character, Item, Quest, Party, Story,

Microplay, Postal

성능 문제

- 물리 서버 1대로 서비스 1종의 부하를 못 견딤

사내 테스트

서비스들

Frontend, Character, Item, Quest, Party, Story,

Microplay, Postal

성능 문제

- 물리 서버 1대로 서비스 1종의 부하를 못 견딤
- 통합 네트워크 튜닝
- 한 서비스를 여러 서버에 분산하는 메커니즘 구현

분산 서비스 구현 I

분산 트랜잭션

한 operation을 하나의 서비스 인스턴스만 처리하는 메커니즘 필요

분산 서비스 구현 II

엔티티

- 한 캐릭터의 인벤토리, 우편함, 하나의 전투 같은 분산 처리의 단위를 묶음
- 엔티티를 찾는 연산이 복잡하니 엔티티 간 연결을 계속 유지

중복 로드 방지

모든 동기화는 DB 에서

Grand Open – 2010. 01 I

서버 설정

- 30 대의 물리서버
- 2 대의 DB 서버
- 그리고 웹 서버

Grand Open – 2010. 01 II

경과

- 최대 약 5만 동시 접속자 수
- *정기 점검과 긴급 점검과 긴급 점검과 · · ·*
- DB 서버에 log insert 속도를 따라가지 못했다고 한다
- 물론 분산 트랜잭션 동기화도 문제

사례 I

엔티티가 정확히 언로드 되는 시점을 정할 수가 없다

- 엔티티 간 커넥션 카운팅 : 순환 참조 문제
- 언로드 조건 변경 : Dangling reference 문제

(GC 언어 위에) 분산 GC를 만들어야 할 판?

사례 II

랙이 퍼진다

1. 오퍼레이션이 다른 오퍼레이션을 기다린다
2. 한 서비스가 느려지면 연쇄적으로 다른 서비스의 대기 큐도 증가
3. 일부 오퍼레이션은 구조적으로 다른 많은 오퍼레이션을 기다림

사례 III

오퍼레이션 폭발

1. 한 엔티티에 너무 많은 옵저버가 있으면
2. 엔티티의 변경이 많은 서비스에 영향을 준다
3. 엔티티가 자주 바뀌면 ? (스스로) DoS 공격하는 효과

사례 IV

오퍼레이션 오버헤드

- 메시지 시리얼라이즈가 부담
- 실제 처리 하는 시간보다 오퍼레이션이 늘어나는게 성능에 악영향
- 서비스를 세분화해서 오히려 역효과

사례 V

로드 밸런싱

- 같은 종류의 서비스 간 로드 밸런싱
- 랜덤에 의존해서 했더니 잘 안 되더라 (머신 간 성능 차이 고려 X)

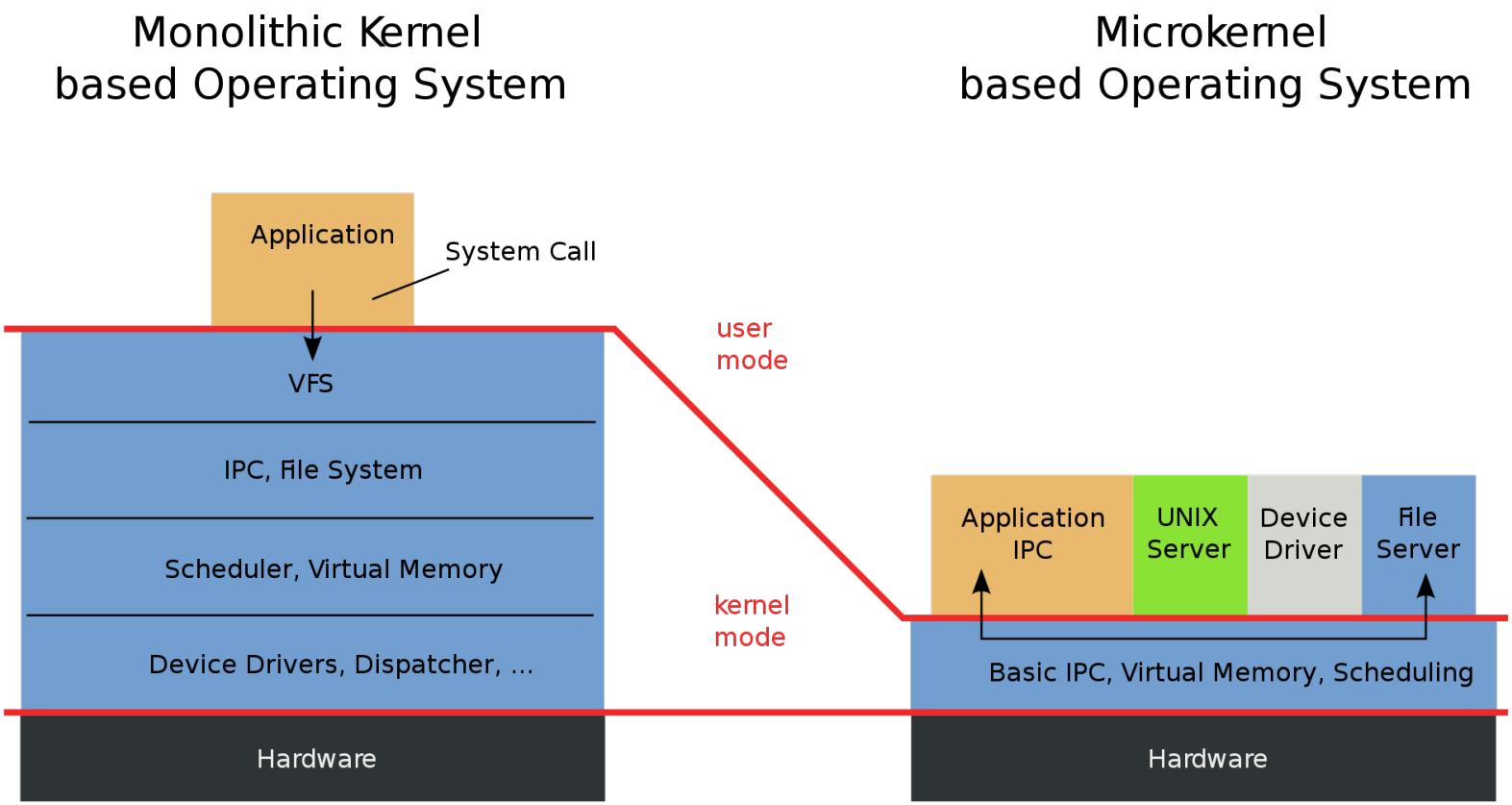
사례 VI

프런트엔드 서비스 비대화

클라이언트 요청을 오퍼레이션으로 바꾸고, 그 결과를 처리하는 서비스

- 거의 모든 로직 내장
- I/O 및 CPU 로드 모두 과다
- 가장 많이 떠 있는 서비스 (물리 머신마다 1개 이상)

마이크로 커널?



http://en.wikipedia.org/wiki/File:OS-structure.svg, public domain에 존재

분산 서비스 구조 I

마이크로 커널과 유사한 구조

- IPC 같은 '통합 네트워크' 제공
- 데몬 간 통신으로 동작하는 것처럼, '서비스' 간 메시지 교환으로 동작한다
- IPC를 네트워크 수준으로 확장해도 잘 동작한다

분산 서비스 구조 II

예상되는 문제도 동일하다

- IPC 통신 부하가 크다: 메시지 시리얼라이즈에 걸리는 부하 (I/O, CPU)
- 복잡한 라우팅 단: 비대한 프런트엔드 서비스

분산 서비스 구조 III

Fault Tolerance

- 한 서비스가 크래시하면 그 서비스에 연결된 엔티티 때문에 전체 서버군을 재시작 해야 할 수도
- 일정 기간 동안 서버가 아무 종류나 문제를 일으킬 확률이 p 일 때 서버가 N 대라면, 해당 기간 동안 무사히 서비스할 확률은 $(1-p)^N$ 이다
- 1 년에 대해 좀 큰 $p = 0.01$, $N = 32$ 일 때, 무사히 서비스할 확률은 겨우 72% 밖에 안된다

구현 문제

- .NET Garbage collector: 개발 초기에 서버가 정지하는 문제의 원인
- GC때문에 상대적으로 큰 캐시 풋 프린트
- 게임 구현에서 '채널'의 역할이 모호하다
- 분산 트랜잭션 구현 방식 때문에 DB 서버가 피할 수 없는 병목
- 현재 수준이 아니라 10만 명, 20만 명 수준이 된다며?

Q & A