게릴라 멀티플레이어 개발

(Guerrilla Multiplayer Development)

작성자: 어니스트 우(Ernest Woo)

작성일: 2012년 9월 13일

스마트폰에서 네트워크 게임을 하게 하는 가장 좋은 방법은 무엇일까? 어니스트 우(Ernest Woo)는 우주 배틀 게임 <ErnCon>을 위해서 기존의 네트워크 기술을 살피고 통합시켰다. 그 결과로 여기에 iOS와 안드로이드 개발자들에게 유용한 솔루션을 공유한다.

인디 게임 개발자의 가장 큰 이점 중 하나는 만들고 싶은 게임을 만들 자유가 있다는 것이다. 아이디어를 막는 회사의 관료들 없이 어떤 말도 안 되는 아이디어라도 시도할 자유가 있다 – 스스로 뭘 하고 있는지 전혀 이해하지 못한다 해도 말이다!

나의 첫 안드로이드 게임 <FRG>에 대한 사용자들의 피드백과 <프로젝트 INF(Project INF)> 같은 초기 멀티플레이어 안드로이드 게임에서 받은 영감을 바탕으로, 나는 후속편을 3G에서도 플레이 가능한 리얼타임 멀티플레이어 총격 게임(real time multiplayer shoot 'em up)으로 만들고 싶었다. 문제는 **내가 멀티플레이어 게임을 어떻게 만드는지 모른다**는 데 있었다.

결론적으로 나는 이 장애물을 극복했고 <ErnCon>을 릴리즈할 수 있었다. 그 과정에서 나는 멀티플레이어 게임을 만드는 많은 기술적인 세부 사항들을 배웠다. 여기에 공유하는 내 경험이 모바일 리얼타임 멀티플레이어 게임 개발 방식에 실마리가 되었으면 좋겠다.

어디부터 시작했는가?

나는 직업상 자바 웹 어플리케이션(Java web applications)들과 네이티브 어플리케이션들(J2ME, 안드로이드, iOS)을 개발해온 자바 개발자(Java developer)이다. 웹

앱과 네이티브 앱 간의 통신은 항상 HTTP를 이용해서 해왔다 – 리얼 타임 게임에는 적합하지 않은 기술이다! 처음에는 내가 멀티플레이어 게임 개발에 관해 걱정되는 핵심 문제의 답을 찾는 데에 초점을 맞췄다.

1. 네트워크 주소 변환 (NAT). <ErnCon>은 3G에서도 플레이 할 수 있어야 했기 때문에, 나는 휴대 전화의 전형적인 대칭형 NAT(Symmetric NAT)를 선회할 방법을 알아내야 했다. 대칭형 NAT는 내부의 주소/포트(휴대폰)에 다른 외부의 주소/포트 (통신회사/ISP의 NAT)를 매핑(mapping)함으로써 모든 목적지 주소/포트(게임 서버)에 연결한다.

소비자용 라우터(consumer-grade router) 뒤에 디지털 가입자 회선(DSL)이나 케이블 모뎀을 연결해서 실행하는 게임에서는 비교적 자유로운 NAT 유형(Full-cone, Restricted Cone, Port-restricted Cone)을 통과하는 방식이다. 하지만 이 중 어떤 것도 실제로 대칭형 NAT(Symmetric NAT)에 적용할 수 없다.

UDP 홀 펀칭(UDP Hole Punching)과 다른 NAT 통과 기술들을 알아본 뒤, 클라이언트 서버 아키텍처의 비교적 단순한 솔루션으로 정착했다. 서버가 항상 퍼블릭 IP 주소를 가진, 다시 말해 서버 자체가 NAT가 아닌 방식이다. 퍼블릭 서버와의 통신을 위해서는 특별한 NAT 통과 트릭이 필요 없다. 더 자세한 내용을 알고 싶으면 NAT에 대한 위키피디아(Wikipedia) 항목 [1]을 읽어 보기 바란다.

2. 단순 네트워크 모델(Simple network model). 리얼타임 멀티플레이어 게임 개발 경험이 없는 상태로, 나는 다른 게임들이 네트워크 레이어를 어떻게 구성했는지에 대한 정보를 찾았다. 에픽 게임즈(Epic Games)는 <언리얼 토너먼트(Unreal Tournament)>의 네트워크 아키텍처에 대한 정보를 공개[2]했다. <언리얼 엔진(Unreal Engine)>의 컨셉으로 하기엔 너무 빡빡하다는 것을 깨닫긴 했지만 말이다 – 나는 <언리얼 엔진3>를 만들고 싶지는 않았다. 내가 최종적으로 선택한 <퀘이크 3(Quake Ⅲ)>의 네트워크 모델[3]은 마지막 받은 상태에서 전체 게임 상태 업데이트를 델타 압축해서 보내는 개념으로, 훨씬 이해하기 쉬운 방식이었다.

3. 활용 기회의 축소. 나는 인디 개발자이다 보니 치트(cheat)를 강력히 감시할 자원이 없다. 게임의 아키텍처 자체가 해커를 막아야 하고, 클라이언트가 통제하는 데이터는 모두 이용될 수 있었다. 클라이언트 서버 아키텍처는 서버가 모든 게임 데이터의 유일한 권한을 갖게 만듦으로써 이러한 우려를 잠재울 수 있다.

---

[1] 참조 링크: http://en.wikipedia.org/wiki/Network_address_translation

[2] 참조 링크: http://udn.epicgames.com/Three/NetworkingOverview.html

[3] 참조 링크: http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/Quake3Networking

4. 전세계에 서버 배치. 만약 세상의 모든 플레이어들이 한 장소에 위치한 서버에 연결한다면, 클라이언트 서버 아키텍처를 이용한 리얼타임 멀티플레이어 게임은 잘 작동하지 않을 것이다. 랙(lag)을 줄이고 게임 플레이 성능을 향상시키기 위해서는 서버를 전략적으로 세계 곳곳에 배치해야만 한다. 플레이어를 지역으로 분류하여 지역적으로 가까운 플레이어들이 언제나 같은 서버에서 플레이 할 수 있어야 한다. 나는 아마존 웹 서비스(AWS)[4] 덕분에 미국, 유럽, 동남 아시아, 남아메리카를 포함한 여러 지역에 서버를 마련할 수 있었다.

실행 세부 사항

속담에도 있듯이, "문제는 사소한 데에 있다." 적절한 아키텍처를 찾고 네트워크 모델을 결정하고, 고심 끝에 프로바이더(provider)를 정하는 노력이 실제 모든 것을 시행하는 데에는 무색했다. 너무 세세한 부분까지 들어가면 지루할 테니, 여기서는 내가 해결한 몇 가지 장애물에 대해서만 이야기하기로 하자.

1. 기존의 게임으로 시작하라. 나는 당장 멀티플레이어의 장점을 모두 가진 <ErnCon>을 시작하고 싶었지만, 기존에 만들었던 <FRG>에서 멀티플레이어 프로토타입을 만들기로

[4] 참조 링크: http://aws.amazon.com/ko/

했다. 그 덕분에 네트워크 코드와 클라이언트 서버 개발에만 집중할 수 있었다. <FRG>의 멀티플레이어 실행 수준에 만족하고 난 뒤에야 이어서 새로운 게임을 개발할 수 있었다.

2. 자바(Java)를 이용하고 남용하라. 안드로이드 앱은 자바에서 개발할 수 있기 때문에 나도 역시 자바를 이용해 서버 개발을 간소화했다. 그렇게 해서 톰캣(Tomcat)에서 실행할 수 있는 웹 어플리케이션을 금방 만들었고 그래픽과 음향 없이 게임을 가동할 수 있게 됐다. 클라이언트와 서버 간의 코드를 공유함으로써 양 쪽 모두 같은 게임을 시뮬레이팅했는지 확인하는 수고를 덜 수 있었다.

3. 자바를 더더욱 이용하고 남용하라. <퀘이크3>의 네트워크 모델을 빨리 시행하기 위해서, 나는 자바의 실시간 어노테이션(annotations)과 리플렉션(reflection)을 광범위하게 사용했다. 어노테이션은 게임 엔티티(game-entity) 클래스에서 네트워크를 피할 수 있는 필드를 표시하는 데 사용한다. 클라이언트와 서버는 어노테이션이 달려있는 필드의 게임 엔티티 목록에 근거해 UDP 패킷(UDP packet)을 조합한다. 클라이언트와 서버 간에 코드가 공유되기 때문에, 네트워크 프로토콜(network protocol)을 한 군데에서만 바꾸면 되었다.

4. 매치메이커 서비스 (Matchmaker service). 게임에서 플레이어를 모으고, 게임 서버와 통신하고, XP, 캐시, 인벤토리 같은 플레이어 상태를 기록하기 위해선 추가 서비스가 필요하다. 아마존 웹 서비스(Amazon Web Services)는 내가 다음과 같은 매치메이커 서비스를 개발하고 배포하는데 필요한 것을 모두 제공했다.

- **EC2 로드밸런서(load balancer)**
- **엘라스틱캐시(Elasticache) 및 관계형 데이터베이스 서비스(RDS)**를 퍼시스턴스 레이어에 제공. (근본적으로 아마존이 운영하는 맘캐시드(Mamcached)와 MySQL) Memcached.org [5] 에 따르면, 맘캐시드는 "무료&오픈 소스, 고성능, 광범위 메모리 오브젝트 캐싱 시스템"이다.
- **심플 스토리지 서비스(S3)**는 ErnCon 스토어에서 보여지는 아이템 이미지를 비롯한 게임 애셋과 WAR과 JAR 파일 서버를 호스팅한다.

5. 서버 배치의 용이함. 아마존이 <ErnCon>을 개발하고 배치하는 백엔드(backend)에서 도움이 되는 툴을 많이 제공했지만, 그래도 최대한 힘들지 않게 새로운 서버를 프로비저닝하기 위해서는 할 일이 여전히 많았다.

---

[5] 참조 링크: http://memcached.org/

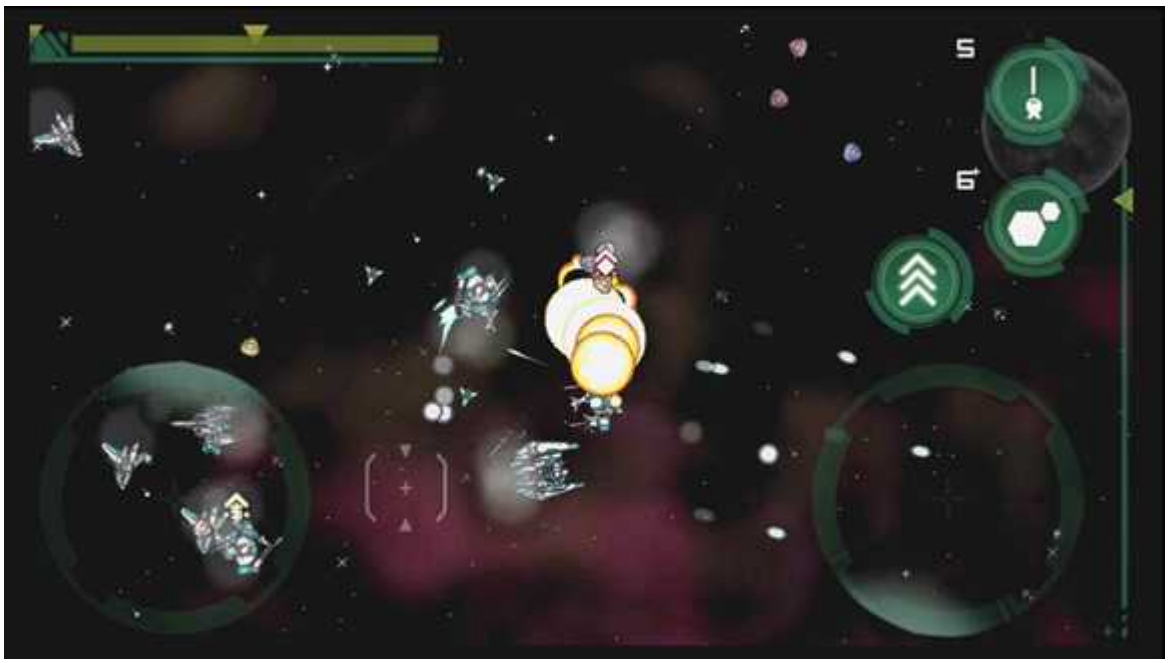
현재 나는 주문형 아마존 머신 이미지(AMI)를 보유하고 있는데 S3의 최신 매치 메이커 WAR를 가져온다. 게임서버도 마찬가지로 최신 게임 서버 WAR에서 가져온다.

게임 서버들은 자동적으로 매치메이커에 등록되어서 매치메이커가 게임을 만들고 플레이어를 게임 서버에 보내기 시작할 수 있다. 서버 코드를 업데이트할 때의 배포 절차는 다음과 같다.

1. 업데이트된 매치메이커와 게임 서버 WAR 그리고 JAR 파일들을 S3에 업로드한다.
2. 나의 주문형 AMI에서 EC2 Instance Launch Wizard를 실행해서 새로운 EC2 인스턴스(EC2 instance)를 만든다.
3. 인스턴스가 동작해서 플레이어와 게임을 받을 때까지 기다린다.
4. 모든 지역에서 반복한다.

이렇게 해서 새로운 게임 서버를 프로비저닝하는 데 전체적으로 5분도 안 걸린다.

<ErnCon>의 멀티플레이어 아키텍처는 대부분 여가시간에 개발했기 때문에 특히 자랑스럽게 생각하는 부분이다. 다음 순서는 네트워크 프로토콜이다.



3G 고려사항들

<ErnCon>의 멀티플레이어 아키텍처를 디자인할 때, 게임이 3G에서 플레이 할 수

있어야 한다는 게 주요 요건이었다 – 와이파이(Wi-Fi)가 필수 요건이어서는 곤란했다. 게임 플레이는 가능한 한 마찰이 없어야 한다 - <ErnCon>은 플레이어에게 와이파이 연결을 확인하기 위해 게임에서 나가라고 할 수 없다. 또 플레이어에게 홈 와이파이에 연결하라고 강요하는 것은 모바일 게임의 자세가 아니다! <ErnCon>을 3G에서도 가능하게 하기 위해서는 개발 과정에서 다음의 것들을 고려해야 했다.

1. 일반적인 3G 통신 속도에서 플레이 할 수 있어야 한다. 3G에 특화된 최소, 평균 통신 속도가 정립되어 있지는 않지만, 조사결과 알아낸 데이터 전송속도는 최소 384Kbit/s였다.

2. 일반적인 3G 지연 시간(latency)에서 플레이 할 수 있어야 한다. 일반적인 지연 시간에 대해 명백한 자료를 찾기는 힘들다. 내가 포럼과 기사들을 통해 찾아낸 정보에 의하면 <핑(Ping)> 앱을 이용해 T-모바일(T-mobile)과 버라이즌(Verizon)에서 테스트해 보면 왕복 300ms 수준에서 500ms나 그 이상까지 급증하는 것으로 나타났다. 300ms는 좋지는 않지만 클라이언트 사이드에서 잘 예상한다면 플레이가 가능한 수준이다.

<퀘이크 3> 네트워크 모델

이 글을 계속 읽기 전에 부디 <퀘이크 3>의 네트워크 모델[6]을 살펴보기 바란다. 이 부분은 UDP 네트워크 같은 컨셉을 이미 알고 있다고 간주하고 작성하겠다. <퀘이크 3> 네트워크 모델의 몇 가지 핵심 포인트는 다음과 같다.

1. 서버는 어떤 데이터가 누구에게 보내졌는지 추적한다. 서버는 어떤 게임상태가 모든 클라이언트에 보내졌는지 추적하는 작업을 많이 한다. 이는 서버가 데이터를 델터로 압축하면서 대역폭을 절약할 수 있게 한다.

2. 데이터의 델터 압축. 이는 서버가 마지막 보낸 상태에서 업데이트된 클라이언트 데이터만 보낼 것임을 고급스럽게 말하는 방법이다. 예를 들어, 우주선의 x, y 좌표와 방향에 대해서 트래킹한다고 할 때, 플레이어가 (y축을 따라) 올라간다면 서버는 플레이어의 y 좌표 업데이트만 포함한 패킷을 보내게 된다.

3. 랙 다루기(Implicit handling of lag). 랙 스파이크(Lag spike)나 다른 UDP 데이터의 손실은 서버에서 처리한다. 클라이언트에서 받은 최종 상태에 기준해서 서버가 계속 네트워크 업데이트를 하는 것이다. 이렇게 되면 응답 채널을 따로 가져갈 필요가 없어진다.

---

[6] 참조 링크: http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/Quake3Networking

게임 엔티티 직렬화 (Serializing Game Entities)

네트워크 코드에 대해 쓰기 전에, 나는 내 기존 게임 엔티티(우주선, 총알, 장애물 등) 중에서 어떤 데이터를 시리얼라이징하고 네트워크에 전송해야 하는지 결정할 방법이 필요했다. 이전 글에서도 언급했듯이 나는 이 "흥미로운" 필드를 점검하기 위해 실시간 자바 어노테이션(runtime Java annotations)을 사용하기로 결정했다. 네트워크 코드는 클래스 정의(class definition)를 스캔 할 수 있고, 네트워크에 보내고 받는 데이터를 직렬화하거나 역 직렬화하는데 사용하는 직렬변환기를 만들 수 있다.

어노테이션은 단순하다. 두 개가 서로 다른 방법으로 필드를 검사하기만 하면 된다.

1. 넷데이터 델타(NetData DELTA). 이 필드들은 필드의 가치가 클라이언트가 받은 최종 상태와 다를 때마다 네트워크로 보내진다. 대표적인 델타 필드들에는 게임 엔티티의 위치(x, y 좌표들), 방향, 생명력(life)이 있다.

2. 넷데이터 풀(NetData FULL). 이 필드들은 게임 엔티티가 정해진 클라이언트에 새로운 것으로 확인될 때 보내진다. 위에서 언급한 바와 같이, 서버는 클라이언트로 보내지거나 보내지고 있는 모든 데이터를 추적한다.

풀 필드(FULL field)의 값은 게임 엔티티가 살아있는 동안 절대 변하지 않는다. 따라서 풀 필드는 절대 이니셜 페이로드(initial payload) 이후에 서버로부터 게임상태 업데이트를 나타내지 않는다. 대표적인 풀 필드들에는 게임 엔티티의 오브젝트 타입(object type/ 이 엔티티가 대표하는 총알, 우주선 혹은 장애물의 유형이 무엇인가)이 있다.

다음 코드는 <ErnCon>에서 플레이어와 인공지능으로 움직이는 발사체를 대표하는 로봇(Robot) 계급의 주석 필드들(annotated fields)이다.

```
@NetData(type = NetData.Type.DELTA) public float x, y, rotation, scaleX, scaleY;
@NetData(type = NetData.Type.FULL, subfield = "id", subfieldType = RobotDef.class) public RobotDef def;
@NetData(type = NetData.Type.DELTA) public int item1Ammo, item2Ammo;
@NetData(type = NetData.Type.DELTA) public boolean firing, boosting;
```

"def" 필드가 풀 필드라는 점에 주목하기 바란다. 풀 어노테이션(FULL annotation) 위의 추가 속성(extra attribute)은 논 프리미티브 필드(non-primitive field)를 직렬화 하는

직렬변환기를 말할 때 사용되며, 이들 필드는 네트워크를 통해 보내야 한다.

자바 리플렉션은 실시간 어노테이션을 위한 클래스의 필드 정의를 검사하도록 한다. 리플렉션은 또한 이 필드 정의들이 각각 게임 엔티티들의 인스턴스에 값을 받아 놓는 데 사용되는 직렬변환기에 보관되도록 한다. 결국, 직렬변환기는 게임 엔티티의 스냅샷 두 개를 어떻게 찍어야 하는지, (만약 있다면) 차이점을 어떻게 써내야 하는지를 아는 것이다. 결과물은 네트워크에서 클라이언트들로 보내진다. 직렬변환기는 또한 네트워크에서 서버 쪽의 상태(server-side state)에 맞추도록 만드는 랜덤 인스턴스로 받은 데이터를 적용하는 방법도 알고 있다.

다음의 코드 정보는 DATA_GRAPHS에 보관된 그들의 직렬변환기를 찾고, 최종 인지 상태에서 델타 오브젝트 혹은 만약 오브젝트가 새것일 경우 풀 오브젝트를 쓰는 toObjs 내 각 게임 엔티티의 반복을 보여준다.

```
short numObjectsWritten = 0;
HashSet<Integer> ids = new HashSet<Integer>(toObjs.size());
for ( int i = 0; i < toObjs.size(); i++ ) {
    tmpObj = toObjs.valueAt(i);
    if ( tmpObj != null ) {
        graph = DATA_GRAPHS.get(tmpObj.getClass());
        if ( graph != null ) {
            GameObject fromObj = objects.get(tmpObj.id);
            ids.add(tmpObj.id);
            if ( fromObj != null ) {
                // Calculate delta
                buffer.mark();
                buffer.put(DELTA_OBJECT);
                if (graph.writeDelta(fromObj, tmpObj, buffer)) {
                    numObjectsWritten++;
                } else {
                    buffer.reset();
                }
            } else {
                // Store full object
                buffer.put(FULL_OBJECT);
                graph.writeFull(tmpObj, buffer);
                numObjectsWritten++;
            }
        }
    }
}
```

곧 나올 <ErnCon> iOS 버전에서는 익스포터를 만들어서 모든 직렬변환기와 필드 어노테이션 정보를 프로젝트에 장착된 JSON 파일에 넘기게 했다.

게임 상태 기록하기와 보내기

직렬변환기가 개별 게임 엔티티들을 받고 두 상태 간의 차이점을 쓰는 방법을 알고 있어서 전체 게임 상태의 델타를 네트워크에 쓸 수 있었다. 게임 상태를 보관하는 것은 매우 간단하다 – 특정한 상태를 위해 활동적인 모든 게임 엔티티들은 게임스테이트 오브젝트(GameState object)에 의해 운영되는 목록에 복사되고 보관된다. 이 게임스테이트 오브젝트들은 <퀘이크 3> 네트워크 모델에 묘사된 대로 정확히 사용된다.

1. 게임스테이트는 모든 클라이언트들에게 가장 빨리 인정받은 게임스테이트에 저장된다.

2. 다음의 코드 정보에서 보여주듯이, 각 클라이언트에 대해 최근 게임스테이트와 클라이언트의 최종 게임 상태 사이의 델타를 개산함으로써 데이터 패킷을 만든다.

```
if ( client.lastAckState == Client.INVALID_STATE ) {
    buffer.put(NetworkUtils.FULL_STATE);
    latestState.writeFullState(buffer);
} else if (client.lastAckState != latestStateId ) {
    lastAckState = gsList.get(client.lastAckState);
    if ( lastAckState != null ) {
        buffer.put(NetworkUtils.DELTA_STATE);
        lastAckState.writeDeltaState(latestState, buffer);
    } else {
        buffer.put(NetworkUtils.FULL_STATE);
        latestState.writeFullState(buffer);
    }
}
```

<퀘이크 3> 네트워크 모델의 약점 중 하나가 이 점에서 명백해진다는 점에 주목하라. 게임 상태의 임의적인 번호를 위해 관련된 모든 게임 엔티티들을 복사해 저장하다 보면 매우 비싸질 수 있다. <ErnCon>에선 다음과 같은 이유 때문에 이게 즉각적인 큰 문제는 아니었다.

1. 플레이어 최대 인원 수가 8로 제한된 것은 저장해야 하는 상태(와 클라이언트들과 통신할 때 대역폭)의 양을 줄이기 위한 것이었다

2. EC2 인스턴스에서 실행되는 게임 서버들은 필요할 때 새로운 서버들의 공급을 쉽게 해주었다. 다수의 게임 관리에 책임이 있는 EC2 인스턴스는 실행중인 게임의 로드(load)를 분산시켰다.

3. 게임 상태를 모든 프레임에서 기록하지 않고 **정기적으로 기록하기**. <ErnCon>은 메모리를 보호하기 위해 50 미리세컨트(milliseconds[7])에 한 번씩만 새로운 게임스테이트 오브젝트를 만든다. 이는 아직 최적화 수치를 내기엔 초기단계로 내가 나중에 정리할 것 중 하나다.

네트워크 데이터 다루기

네트워크에 데이터를 보내고 직렬화하는 것은 중요하지만 <ErnCon> 네트워크 코드의 전부는 아니다. <퀘이크 3> 네트워크 모델 시행 이후, 나는 어떤 데이터가 보내져야 하고 어떻게 다루어야 하는가를 알아내는 데 주력했다. 비록 이는 게임 유형에 따라 엄청 달라지겠지만, <ErnCon>에 적용된 것 중 꼭 알아야 하는 몇몇 일반적인 경우들이 있다.

1. 클라이언트는 로컬에서 시뮬레이션을 실행하지만 게임을 가이드 하기 위해 네트워크 데이터를 이용한다. <ErnCon>에서의 데드 레커닝(dead-reckoning [8])은 단순하게 서버로부터 오는 데이터가 없을 때 게임 시뮬레이션이 클라이언트에서 실행할 수 있게 하는 것을 의미한다. 컴퓨터가 제어하는 적(enemy)의 인공지능 논리(AI logic) 또한 인공지능 행동(AI behavior)이 어떤 정해진 게임 상태에도 결정론적이기 때문에 클라이언트 사이드에서 실행한다. 부상 같은 중요한 사건은 총알이 우주선을 맞춰 폭발 애니메이션이 보여진다 하더라도 클라이언트 사이드에 적용되지 않는다.

2. 클라이언트와 서버 사이의 핑 타임(ping time). 작업을 위한 데드 리커닝 같은 클라이언트 사이드 예측을 위해, 클라이언트는 패킷이 서버에 도달하는 데 걸린 시간을 알아야 한다. 각각의 <ErnCon> 클라이언트는 매 초 핑 패킷(ping packet)을 보낸다. 핑

---

[7] 1000분의 1초

[8] 참조 링크: http://gamasutra.com/view/feature/3230/dead_reckoning_latency_hiding_for_.php

패킷을 받으면, 서버는 즉각 응답한다. 각각의 핑 패킷은 클라이언트에 보내는 고유의 ID를 가지고 있으며, 이는 왕복 시간을 알아내는 데 사용된다. 마지막 3개의 핑의 이동 평균은 이용되는 모든 데드 리커닝 계산의 기반으로 사용된다. 경험상, 3개 이상의 핑들을 추적하는 것은 가끔 데드 리커닝에 부정적인 영향을 미쳐서 네트워크 지연이 발생하기도 한다.

3. 네트워크 데이터와 클라이언트 시뮬레이션 융합하기. 클라이언트가 서버로부터 게임 상태 업데이트를 받을 때, 최종 상태를 거쳐 새로운 상태를 생성하기 위한 델타 압축 데이터(delta-compressed data)를 적용해야만 한다. 이 새로운 상태는 다시 실제 지역 시뮬레이션에 적용되어야 한다. <ErnCon>의 각 게임 엔티티는 적절한 데이터를 촉발하고 핑 타임에 기반해 로컬 엔티티(local entity)를 새로운 서버로 밀어냄으로써 새로운 상태로부터 데이터를 머지하는 방법을 알고 있다. 지역 위치와 업데이트된 서버 위치에 삽입하기 위해 복잡한 공식들[9]을 사용하지는 않는다 – 새로운 위치의 방향에서 벡터(vector)는 그대로 오브젝트에 적용된다. 오브젝트가 서버 위치에서 너무 멀리 벗어나 있으면, 클라이언트가 오브젝트를 최신 서버 위치로 빠르게 옮긴다.

4. 네트워크 데이터로부터 새로운 게임 엔티티 만들기. 서버에서의 네트워크 업데이트는 클라이언트에 알려지지 않은 – 예를 들어, 총알이 발사되었을 때 - 오브젝트에 데이터를 포함시킬 수 있다. 클라이언트들은 이 알려지지 않은 게임 엔티티들의 새로운 로컬 인스턴스를 만들 수 있다.

아직 남은 과제

<ErnCon>의 멀티플레이어 시행의 주요 요점에 대해 논의했지만, 여전히 논할 것이 더 많이 남아 있다. 100 페이지짜리 논문을 여기 올려서 여러 사람을 괴롭히기보단 아래의 코멘트나 이메일(contact@woogames.com)을 통해서 질문을 받아 논의를 지속하도록 하겠다. 기술적 문의에 대해 최선을 다해 코드/의사 코드(pseudo-code)를 제공하겠다.

---

[9] 참조 링크: http://www.gamedev.net/page/resources/_/technical/multiplayer-and-network-programming/defeating-lag-with-cubic-splines-r914