※ 본 기사는 CMP MEDIA LLC와의 라이선스 계약에 의해 국문으로 제공됩니다

무한한 공간 : MMO의 Single-Sharded 구조에 관한 논쟁
(Infinite Space: An Argument for Single-Sharded Architecture in MMOs)

크자르탄 에밀슨, CCT 팀 (Kjartan Emilsson, CCP Team)

가마수트라 등록일(2010. 08. 09)

http://www.gamasutra.com/view/feature/4175/infinite_space_an_argument_for_.php

대부분의 MMORPG 는 플레이어 밀집도와 서버 문제를 해결하기 위하여 게임 월드를 각각의 사례나 조각으로 만든다. 싱글 샤드(single shard)는 MMO 개발의 자연스러운 선택 사항이 된 것 같고, EVE Online 에서도 이와 같은 방법을 사용했다.

"왜 싱글 샤드 구조 인가?"라고 물음에, "왜 샤드를 구성하는가?"라는 좀더 심오한 질문을 고려하는 것이 더 도움이 될 것이다. 개발자가 샤드 실행을 선택하는지에 대하여 두 가지 이유가 있다. 내용과 기술적 도전의 부족 때문이다. 이 두 가지는 실제로 밀접하게 연결되어 있다.

내용

대부분의 MMO 는 물리적 제약에 의해 제한된 환경에서 발생한다. 지구처럼 생긴 공간이나 건물과 같은 공간적 제약이 있는 곳에서 움직이는 아바타가 그렇다. 게다가, 이러한 구체적인 환경에서, 플레이어는 퀘스트와 NPC 를 만나는 것과 같은 수 많은 스크립트화 된 활동과 맞닥뜨린다.

가장 제한적인 물리적 제한은 아바타의 밀도를 걱정스럽게 만든다. 이것은 기술적인 문제인 동시에 유용성 문제이기도 하다. 플레이어는 끊임없이 과밀화된 환경에서 게임 하기를 원하지 않는다. 적당한 수준으로 아바타 밀도를 유지하기 위해서, 광대한 플레잉 필드가 있어야 하거나 주어진 공간 내에 수용할 수 있는 플레이어의 수를 제한하는 것이 필요하다. 이 두 가지 선택 모두 당신이 디자인 할 수 있는 내용의 양에 의해 제한되고, 내용은 게임 개발 비용에 가장 큰 부분이기 때문에, 이는 재정적인 제한과 결부된다.

분명한 해결책은 순차적으로 발생하는 내용을 만드는 것인데, 이를 통해 당신이 원하는 만큼의 플레잉 필드를 가질 수 있게 된다. 이러한 접근의 단점은 수작업화 된 환경에 보여지는 미적인 수준에 절대 접근할 수 없을 것이라는 것과 스크립트화 된 활동이 반복적이거나 상황적 요소가 부족해 진다는 것이다.

이 문제에 대한 진정한 해결책은 다른 소셜 네트워크와 마찬가지로 MMO 에도 플레이어가 내용이 된다는 생각을 가지는 것이다. 일단 이 생각이 기본적인 디자인 가이드라인으로 받아들여지면, 싱글 샤드 구조를 만들고 유지하는데 결부된 도전은 더 쉬워진다. 실제로 디자인 모델에 여러 장점을 부여할 수 있게 된다.

이 가정을 좀더 면밀히 살펴 보면, 사람들에 의해 발생되는 두 가지 종류의 내용을 알 수 있다. 게임 월드 안에서 반복되는 유저가 생산한 자산으로 묘사될 수 있는 물질 내용(material content)과 소셜 인터랙션의 패턴으로 간주되는 사회성 내용(social content)이다.

전자는 이해하기 쉽다. 반복되는 플레이어 생산 내용이 플레잉 필드에 덧붙여질 수 있고, 다른 플레이어의 그룹에게 월드는 좀더 “의미있는” 공간이 된다. RTS 게임이 이와 같은 경우인데, 배경이 상대적으로 재미없고 자동적으로 발생될 수도 있다.

예를 들어, EVE 에서, 많은 게임 플레이는 규정되지 않은 지역에서 영역을 정복하거나 통치하는 것을 수반한다. 최우선적 우주 정거장을 위치해 둘 장소를 선택함으로써, 플레이어는 전략적인 전장터의 지형에 대비한다. 최우선 우주 정거장의 지원을 받는 스타베이스의 위치를 선정하고, 공격 및 방어 시스템을 환경 설정하여, 플레이어는 결정적인 싸움이 발생하는 상황에서 전략을 구사할 수 있게 된다.

두번째 형태인, 사회성 내용은 매우 강력하지만 주의 깊은 디자인을 필요로 한다. 소셜 인터랙션의 필드는 광범위한 활동과 컨셉을 아우른다. :

- 대화와 메시지와 같은 순수한 소셜라이제이션
- 플레이어 간의 전투나 환경에 대항하는 협동조합식의 전투. 이것은 1:1 전투에서 수천명의 플레이어의 순번으로 나누어진 파벌 간의 전투로 크기를 변경한다.
- 경제적인 활동

소셜라이제이션의 개념에서, 주요한 "내용"은 친구 리스트, 플레이어 멤버십이나 길드 및 포럼에서 실현되는 사회적 태피스트리[1] 이다. 싱글 샤드는 내용에 풍부함을 덧붙이는데, 플레이어가 서버 별로 나누어질 필요가 없기 때문이다. 플레이어는 나누어진 서버라기 보다는 전체 플레이어를 기반으로, 경험을 공유하거나 이슈를 논의할 수 있다. 작은 서버 단위라기 보다는 전체의 사회로 인식될 수 있도록 할 수 있다.

게다가, 명성을 얻는 것은 큰 규모의 이용자일 때 더 큰 보상이 될 수 있으며, 이 때문에 명성을 얻는 것에 더 노력하게 된다. 싱글 샤드 커뮤니케이션 인프라를 구축하는 것에 대한 기술적인 도전이 저평가되어서는 안 된다. 그리고 이 글에서 이를 더 논의할 것이다.

EVE 플레이어에 의해 형성된 기능적인 사회를 개발하고자 한 CCP 의 노력으로, 인구통계학적으로 선별된 플레이어 의회가 개발 과정에서 플레이어의 관심을 대변하는 역할을 맡았다.

싱글 샤드 게임은 하나로 연결된 사회를 형성할 수 있게 해 주었고, 선별된 플레이어가 전체 유권자의 관심을 대변하여 이를 훨씬 더 잘 할 수 있게 해 주었다. 모든 사람들이 서버를 공유하였고, 이를 통해 하나의 사회적 맥락이 형성되었기 때문에 커뮤니티는 논의, 논쟁에 대한 공통된 기준점을 가지며, 유명한 인물이 전체의 플레이어에게 알려 질 수 있게 되었다.

또한 싱글 샤드의 이점은 시간과 공간적인 측면에서 전투의 복잡성을 이루어낼 수 있다는 점에서 장점을 가진다는 것이다. 이 복잡성으로 인해 다양한 역할이 생성되고, 되풀이되는 일상적인 전투가 줄어든다. EVE 에서, 수천명이 함께 싸우는 전쟁을 만들어냈다.

플레이어가 전장의 맨 앞에 서든지 중간 역할을 담당하든지 간에, 전략적인 계획은 오롯이 그들에게 달려있다. 이러한 전투의 규모와 시간은 그들은 진정한 내용물을 형성하도록 해 준다. 수명이 짧고 곧 잊혀지는 소규모 접전 대신, 이 전투는 플레이어를 매료시키는 이야기가 되고, 게임 파워와 명성을 쌓을 수 있게 해 준다. 이러한 전투를 제공하는 것은 서버 및 클라이언트 측면의 기술적인 도전을 수반한다.

[1] 여러 가지 색실로 그림을 짜 넣은 직물. 또는 그런 직물을 제작하는 기술

예를 들어, EVE("거대한 전쟁"이 없다고 일컫어지는)의 중심이 되는 전쟁은 2005 년에 시작되었는데, 정치적이고 이념적인 이유로, 기존의 연합 세력이 새로운 세력을 제압하고자 한 것에서 연유되었다. 이 싸움은 눈덩이처럼 불어나서 전쟁이 되었고, 몇 그룹이 빠지고 항복하고 붕괴되거나 새로운 그룹이 참여하게 되는 양상이 되었다.

이 단계에서 직접적으로 싸움에 임한 조직은 30,000 명 이상의 플레이어였다. 이전의 전쟁에서 진 빚을 청산하기 위한 기회를 가지기 위해 많은 그룹이 싸움에 임했던 것 처럼, 3 년 동안이나 지속된 이 전투는 미래의 전투를 예견하는 장치가 될 것 같다.

마지막으로 경제가 있다. 많은 사람들이 깨닫지 못했다고 하더라도, 경제는 소셜 인터랙션의 정점이다. 디자이너에게 영향을 받기 보다는 플레이어에게 영향을 받는 것이다. 오직 플레이어 주도의 경제 환경만 있을 수 있고, 아이템과 소모품의 가격 변동은 실제의 경제 이치를 반영하기 시작한다. 시장은 게임에 참가하는 모든 사람들의 활동을 비추는 거울이 되고, 플레이의 활동을 바꾸는 역할도 하게 된다.

이러한 플레이어 중심의 시스템은 싱글 샤드에 필요한 것은 아니다. 그러나 이 시스템은 싱글 샤드에 의한 확장된 규모로 인해 촉매작용을 받게 된다. 작은 경제는 몇 몇의 강한 플레이어에 의해 조정될 것이고 가격 변동이 커지며 불안정화가 이루어질 것이다. 경제의 규모가 커질수록 탄력적으로 조정된다. 일단 불안정화를 넘어서면, 게임 월드의 거시 경제적인 요소를 반영하기 시작한다. 그리고 어떤 디자이너도 수작업으로 만들어 낼 수 있다고 생각할 수 없는 사회적 콘텐츠가 된다.

EVE 구조 개요

EVE 에 싱글 샤드를 가져야 한다는 디자이너의 목표에 따라, 우리는 게임 우주의 물리적 배경을 위해 순차적으로 발생하는 콘텐츠를 가지고 가기로 결정했다. 전형적인 갤럭시에서의 자연스러운 거리는 집합점을 자연스럽게 드러나게 해 주었고, 그 결과 태양계의 전체 구조는 하나의 절차 내에서 쉽게 운영될 수 있다. 클라이언트 측면에서, 이것은 더 세밀한 입성도를 가지게 해 주었고, 물리적으로 가장 가까운 환경을 시뮬레이션 하도록 해 주었다.

그러나 이것과 별개로, 백엔드 로직 서버에 대한 개념은 컨텍스트에 의한 다른 노드에 맵팽되된다. 우리는 노드 클러스트의 상위 단계의 로직을 가지고 있다. 이 노드에 의해 조정되는 모든 데이터는 모든 월드를 함께 엮어주는 하나의 데이터 베이스에 읽히고 쓰여진다. (그림 1 참조)

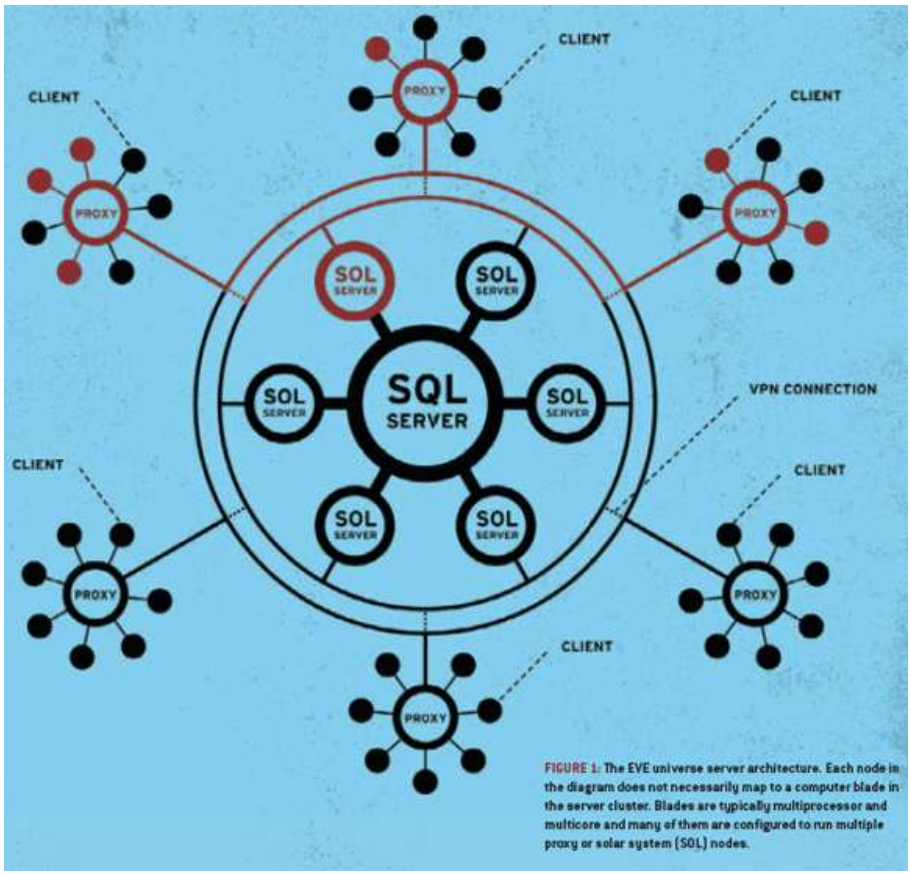


그림 1.

혹자는 하나의 태양계가 샤드처럼 역할할지도 모르나 이것이 올바르지는 않다고 주장할지도 모른다. 글로벌 플레이어 중 어떤 플레이어가 어떤 태양계로도 들어올 수 있고, 모든 경제적 활동이 전체 경제에 간접적인 영향을 미칠 수 있을 것이다.

게다가, 기 언급된 모든 사회적 구조는 시스템 영역을 넘어선 국제적이고 명백한 것이다. 태양계는 “밀집”의 문제를 보이고 있으나 이것들은 게임 디자인에 의해서만 해결될 수 있는 문제이다.

실질적인 목적으로, 전체의 클러스터는 하나의 절차를 가진 우주처럼 움직인다. 그러나 이 접근이 도전을 받아 왔다는 것을 말하지는 못한다.

게임 디자인 도전

플레이어 밀집은 게임 이용성 관점에서 뿐만 아니라 기술적인 관점에서도 도전이 될 수 있다. EVE 에서, 디자인 적 고민을 하게 한 두 가지 종류의 골치 아픈 플레이어 무리를 발견했다. 첫 번째는 "Yulai 문제"라고 불렀던 것이다.

몇 명의 똑똑한 플레이어는 Yulai 태양계가 특히 다른 지역의 스타 클러스터와 잘 연결되었다고 판단했다. 그리고는 그들의 의상을 벌크로 판매하기 시작했다. 그러한 자신의 물품을 판매하는 약삭빠른 판매자는 문제가 아니었지만 그 상태로 오래 머무르지 않았던 것이 문제였다. 구매자가 많이 방문할수록 더 많은 가게들이 생겨났다.

구매자들은 가게를 방문하여 거기서 물건을 사기를 바랬다. 얼마 지나지 않아 플레이어 전부가 Yulai 에서 쇼핑하고 있는 것처럼 보였다. 그리고 그 시스템의 인구밀도는 전체 온라인 플레이어 기준으로 주목할 만한 퍼센트를 차지하였다.

심각한 서버 문제를 야기 이전에, 이 거래 허브의 성장을 억제하기 위한 노력으로, 우리는 Yulai 주변에 점프 루트를 놓아서 맵 상의 변화를 꾀하였다. 몇 달 안에 "Yulai 문제"는 "Jita 문제"로 바뀌었다. 플레이어가 최상의 지점을 알아내어 그 곳으로 비즈니스를 옮겼기 때문이다.

이러한 허브의 형성은 인간의 본능적 행위인 것 같다. 그리고 우리가 시장을 좀더 균등하게 배분시키기 위한 효율적인 방법에 대하여 디자인을 논의하였음에도, 그것은 결국은 하드웨어와 소프트웨어 문제를 해결한 어떤 것이었을 뿐이었다.

두번째 클러스터링 문제는 시장 허브의 문제와는 정반대였다. 전략적인 목적을 위한 거대한 전투였던 것이다. 허브가 영구적이고 예측 가능한 것인 반면, 함대 규모의 전투는 일시적이고 예측 불가능했다. 그러한 전투를 야기하는 급작스러운 게임 플레이는 적군을 이기기 위해 정신적으로 무장한 수천 명의 플레이어를 무리 짓게 하여 정치적인 세력화 되도록 하였다.

특정한 시스템의 인구밀집에 대한 경고 없이, 10 명의 플레이어에서 1200 명의 플레이어로 급속히 자랐고, 이로 인해 기본 수준으로 되돌릴 겨를도 없이 서버 로드가 발생하였다. 이것에 대한 추정적인 디자인 해결책은 동시에 멀티플 시스템으로 전투를 퍼뜨리는 것이었다. 그러나 이것은 "우리가 가진 모든 것으로 적을 공격하라"는 영원한 전투 지침에 직접적으로 반대되는 것이었다. 사령관이 그들의 병력을 실질적으로 이득이 되는 방식으로 쪼개어 두는 그런 방법을 실행할 수 있게 하는 것은 지금도 해결중인 도전 과제 이다.

프로그래밍 도전

큰 규모의 싱글 샤드 환경은 코드 측면에서도 독특한 도전이다. 여기에 우리가 직면했던 몇 가지의 문제점이 있다.

메모리 고갈. 수년 동안, EVE 의 태양계를 운영하는 노드의 최대 메모리 사용량은 게임의 이용자 수와 확정성이 증가하면서 꾸준히 함께 증가하였다. 특히, Jita 와 같은 태양계는 메모리 사용량을 매우 증가시켰다. EVE 가 론칭되었을 때, 32 비트 윈도의 서버 2003 의 환경이었고, 가상 메모리가 2GB 의 제한점을 가지고 있었다. 우리가 64 비트 OS 로 업그레이드 했을 때 가상 메모리는 3GB 로 제한되었다. 그러나 때때로 이 조차도 충분하지 않았고, 그래서 우리는 서버를 64 비트로 바꿀 시점이 왔다고 결정했다.

처음에, 우리는 약간 걱정했다. 클라이언트와 서버에서의 시뮬레이션이 맞지 않았기 때문이었다. 복잡한 수학적 문제를 야기시키는 코드 상이함 때문에 두 사이가 멀어진 것인지 아닌지 확신하지 못했다. 결국 주의 깊은 테스트를 통해서 이것이 그러한 환경에서 알고리즘이 수와 관련된 안전화 문제가 아니라는 것을 확인하였다.

64 비트 바이너리의 출시는 놀라운 사건이었다. 사실, 64 비트 모드에서 큰 수를 등록하는 것은 최적화 가능성을 좋게 해 주었기 때문에 성능이 꽤 증가하였다. 물론 모든 포인터가 2 배의 사이즈가 되었기 때문에 기본 메모리 소비가 증가하였다. 그러나 우리는 3GB 의 가상 메모리 제약에서 벗어날 수 있었다. 이제 각각의 프로세스는 원하는 만큼 할당할 수 있고, 어드레스 스페이스의 고갈도 없었다.

물리적인 메모리에서도, 전통적으로 Sol 노드 프로세스에서 운영되었던 각각의 머신에 4GB 의 피지컬 메모리가 충분하다는 것인 판명되었다(그림 1 참조). 대부분의 할당된 가상 메모리는 활동성이 중단되었고, 페이징도 거의 없었다. 지금까지 페이징 때문에 노드가 죽는 것을 보지 못했다.

비동기 시스템 프로그래밍 및 실행 분배하기. 최상위 단계의 분배 노드의 로직은 전형적인 기능이 프로세스 바운더리로 가서 되돌아 오기 이전에 퍼블릭 인터넷으로 넘어간다는 것을 의미한다. 이러한 비동기식 프로그래밍에 대한 요청은 악명높은 다루기 힘든 문제 이다. Stackless Python 이 이를 해결하기 위해 나왔다.

CCP 에서, 우리는 가능한 한 게임 프로그래에게 직관적이고 간단한 프로그래밍을 만드는 것이 필요하다고 생각했다. EVE 와 같은 복잡한 게임을 만들기 위해서, 멀티스레드 프로그래밍이 필요하다고 생각했다. 우리는 또한 게임 레벨 코드를 위한 스키립트 사용이 필요하다는 것을 알았다. 게임 개발 매우 초기 단계에서 Stackless Python 을 우연히 접한 것은 매우 운이 좋았던 것이며, 이것을 게임 로직에 사용할 수 있었던 것도 운이 좋았다.

Stackless Python 은 "tasklets"와 "channels"의 개념을 소개한 Python 프로그래밍 언어의 변형이다. Tasklet 은 독립적인 OS 스레드가 있는 스레드의 실행이고, tasklets 은 채널의 도움으로 싱크로나이즈하거나 커뮤니케이션하는 것이다. 이들은 실행 스택 보다 더 많은 메모리를 소비하고, 빠르게 하기 위한 커널 모드 스위칭을 필요로 하지 않는다.

무사히, 프로그래머는 그나 그녀가 필요로 하는 어떤 것을 갈기 위하여 새로운 tasklets 을 만들 수 있다. EVE 에서 tasklets 은 협동적인 스케줄링을 사용한다. 특정한 스위칭 지점에서 tasklets 을 스위치 할 수 있다는 것이다. 이것은 멀티스레드 프로그래밍에서 자주 눈의 띄는 복잡한 라킹이나 싱크로나이제이션의 필요롤 제거해 준다.

이 프로그래밍 모델이 빛나는 곳은 I/O 를 가진 곳이다. 효율적인 I/O 는 OS 에 논블로킹 인터페이스를 활용한다. 윈도우에서, 이것은 소켓과 I/O 컴플레션 포트의 형태이다. 스레드는 I/O 작용으로 시작하고 I/O 컴플레션을 조사한다. 그러나 프로그래머의 시각으로, 이것은 극도로 복잡하고 에러를 발생하기 쉬운 것이다. 프로그래머는 단지 보내고 받기를 원할 뿐이고, 이벤트 루프와 같은 것을 걱정하지 않는다.

이것을 가능하게 하기 위해서, 우리는 Python 프로그래머가 I/O 인터페이스를 블로킹 하도록 하였다. 실제로는 오직 taskletbloking 이다. Tasklet 이 *socket.recv( )* 기능을 실행할 때, 우리는 비동기의 I/O 리퀘스트를 WinSock 에 실행한다. 그리고 나서 우리는 tasklet 을 유예하고, 다른 tasklet 이 실행하도록 한다. 나중에, I/O 리퀘스트가 완성된 것을 알았을 때, 막혀 있는 tasklet 의 결과를 준비하고 다시 실행될 수 있도록 한다.

실시간 데이터 베이스 유지하기 및 디자인하기. 우리는 서버 아키텍쳐의 중심부에 있는 데이터 베이스를 선택했다. 우리 데이터베이스 디자인의 핵심은 유지하기 쉽도록 놀랍도록 간단하게 두는 것이다. 모든 중요한 DB 이용은 가능한 한 효율적으로 절차에 따라 일이 진행되도록 했다. 데이터 베이스에 싱글 트리거도 캐스캐이딩 포린 키도 없다. 우리는 소스 코드로부터 보이도록 했다.

우리는 또한 수년 동안 발생하는 DB 코드를 위해 엄격한 코딩 가이드라인을 만들었다. DB 코드 체크를 위한 전문가를 두었고, 올바르지 않은 코드를 고치게 하였다. 데이터 베이스 이용을 모니터하는 것은 매우 중요한 일이고, 우리는 모든 종류의 데이터 베이스 사용에 대한 통계를 모두 저장한다. 우리는 각각의 저장된 절차가 매일 얼마나 불러지는 지고, 인덱스가 얼마나 자주 스캔 되는지 등을 조사한다.

이러한 통계를 살펴 봄으로써, 우리는 만약 개발자가 데이터 베이스를 이상한 곳으로 보내는 코드를 만든다면, 이를 즉시 알아차린다. 우리는 거대한 규모로 실시간 게임을 운영하고 있으며, 이는 스피드가 최우선이라는 것을 의미한다. 그래서 우리는 은행의 교환수와 비슷한 묘책을 사용한다.

이것의 한 예는 태양계 콘피규레이션을 로딩할 때, 우리가 "read uncommitted"를 읽을 때 일 것이다. 우리가 읽고 있는 데이터에 어떠한 인서트와 업데이트도 없다는 것을 알고서, 우리는 어떤 잠김도 없이 읽을 수 있게 해 준다. 또 다른 비책은 이용자가 매일 데이터를 필터 하도록 해 준다는 것이다. 예를 들어, 이용자가 오후 2 시 사이에서 3 시 35 분 사이의 플레이어 저널의 모든 기록을 선택하도록 할 필요가 없다. 날짜로 필터링 하도록 해 준 것만으로도 충분하다. 이 경우에 우리는 매일 자정에 클러스터 키를 따라가도록 해 주면 된다. 이것은 우리가 날짜/시간 칼럼을 인덱스 할 필요가 없다는 것을 의미한다.

오직 하나의 데이터 베이스를 가지고 수행 도전을 개발하는 동안, 어떤 일은 더 쉬워진다. 나누어진 데이터 베이스 시스템을 가지고 있다면, 하나의 데이터 베이스가 모든 캐릭터를 저장하고 다른 데이터 베이스가 각각의 샤드를 저장하여 우리가 처리할 필요 없는 모든 종류의 복잡한 문제가 야기될 수 있다. 데이터 베이스 간의 데이터 복제가 필요 없다는 것 뿐 만 아니라, 멀티플 데이터베이스를 부를 필요도 없고, 샤드 데이터베이스 간에 캐릭터나 소유물을 움직일 필요도 없다.

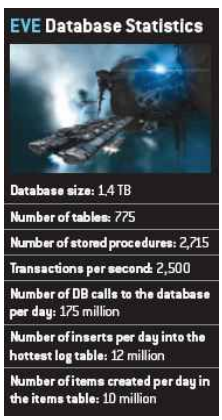
조작적 도전

기 언급했던 요구를 수행하고 데이터 베이스 하드웨어의 우수한 실행 능력을 유지하는 것은 꽤 큰 임무가 될 수 있다. 이것이 우리의 가상 현실의 중심점이고, 그래서 어떠한 잠복이나 느림이 EVE 의 전체 우주에 영향을 미칠 수 있다.

핵심적인 실행을 위해 데이터 베이스 서버에 헤드룸을 두어야 하는 것은 매우 중요하다. 우리가 극복해야만 했던 주요한 병목현상은 데이터 베이스 스토리지의 I/O 실행이다.

시간이 지남에 따라, 우리는 성공적으로 전통적인 화이버 채널의 disk array storage 훨씬 빠른 Solid State Storage 디바이스로 옮겨 왔다. 처음에는 가장 중요한 테이블에 한해서만 이루어졌지만, 최근에 전체 데이터베이스를 모두 SSD 로 옮겼다. 이로 인해 우리는 데이터베이스의 지연 없이 가상 현실을 유지하는데 도움을 얻었고, 여전히 거대한 스케일을 조정할 수 있는 능력을 갖추고 있다.

지속적인 관심을 필요로 하고, 우리 팀과 일하고 있는 분야는 게임 월드를 얼마나 작게 나누어 게임 월드 실행을 위한 무거운 짐을 여러 단계의 노드로 분할할 수 있는 지를 알게 해 주었다. 지금은, 하나의 서버에 전체 태양계를 위한 힘을 할당할 수 있을 정도이다. 그리고 하나의 서버 노드가 싱글 CPU 코어에서 대부분 운영이 될 수 있다. 네트워크와 비동기 방식을 활용하여 가능하게 되었다.



“Jita Problem”과 관련하여 발생한 디자인의 골치거리는 우리가 이 제한점을 해결하기 시작한 그 지점에서 발생했다. 수 천명의 플레이어가 Jita 시스템으로 가거나 전투에 관여하고 있을 때, 우리는 게임 시뮬레이션이 지연 없이 운영이 될 만큼의 프로세스를 처리할 충분한 CPU 파워를 가지고 있지 못해 고민했다. 서버 룸에서, 우리는 지타가 운영될 수 있기만을 바라면서 있었다.

SracklessIO 와 64 비트 서버 코드와 같은 소프트웨어 베이스의 발달은 용량의 큰 차이점을 만들어 냈다. 작년에 우리는 “War on Lag”에서 세 갈래로 나누어진 강습을 보았고, 거기서 우리는 StacklessIO, EVE64 및 서버 하드웨어의 일부 상위 라인을 거의 동시에 시작했다.

결과적으로 Jita 의 용량이 600 명의 플레이어에서 1,200 명으로 증가했다. 6 개월 내 100%의 수용인원의 향상을 이루어낸 것이다. 2 배의 수치를 얻기 위해서 이런 식으로 계속 일 했다.

결론

우리가 이미 보았듯이, 싱글 샤드에서 MMO 를 운영하는 것은 시스템 아키텍쳐, 느린 실행시간, 데이터베이스와 조작에 변형을 가하는 것이다. 그리고 심지어는 게임 디자인 단계에까지 영향을 끼친다. 게다가, 플레이어의 수가 증가함에 따라 변형은 때로는 전혀 기대하지 않았던 곳에서 나타나게 된다.

이런 싱글 샤드 게임의 발전은 끝이 없는 임부이고, 끊임없이 혁신을 필요로 하고, 현명한 해결책을 필요로 한다.

그러나 한계를 종용하는 것은 혁신의 하나의 원천이 되고, 전문가의 관점에서 즐겁고, 플레이어의 경험에 새로운 차원을 더해 줄 수 있는 발견에 이르도록 해 준다. “왜 싱글 샤드인가?”라는 질문에 대한 대답은 “모두에게 무언가 보답을 해 줄 수 있고, 도전적이기 때문이다”라고 답할 수 있겠다. 그리고 이것이 게임에 관한 모든 것이다. 그렇지 않은가?