Half-Push/Half-Polling

1. Introduction

아무리 잘 만들어진 소프트웨어라도 시시각각 변하는 고객들의 요구사항을 반영하기 위해 변경이 불가피 하다. 때문에, 많은 프로그램 개발 업체들은 고객의 만족도를 향상시키고 서비스를 위한 최적의 시스템을 만들기 위해 업그레이드를 고객들에게 지원한다. 이러한 업그레이드를 지원함으로써 사용자는 매번 새로 프로그램을 설치할 필요 없이 가장 최신의 서비스를 사용할 수 있다.

이러한 클라이언트/서버 모델을 적용한 프로그램을 개발하는 업체들이 업그레이드 서비스를 지원하기 위해 고려해야 할 이슈중의 한가지는 데이터 전송 방식이다. IDE 센터를 이용하여 서버의 리소스가 큰 제약사항이 아닌 경우를 제외하면, 서버의 리소스가 한정적이기 때문에 리소스를 효율적으로 사용하기 위한 데이터 전송방식이 필요하다.

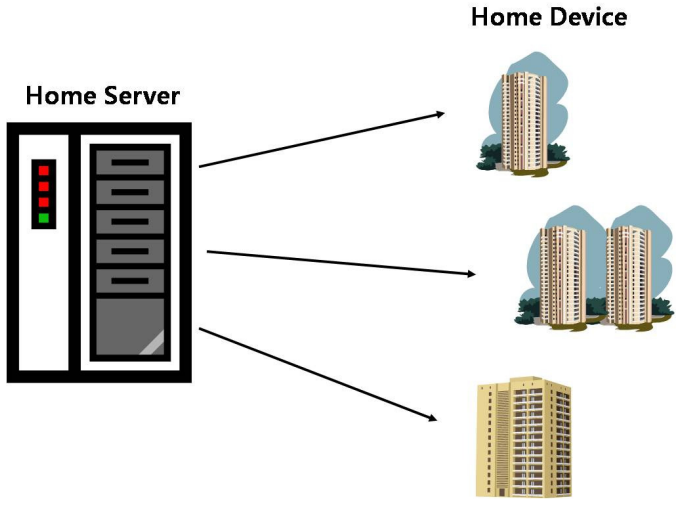
이러한 클라이언트/서버 모델에서 일반적인 데이터 전송 방식으로 polling과 push 방식을 사용하고 있다. 하지만, polling방식은 동시에 많은 클라이언트들이 업그레이드를 요청하면 서버에 과부하가 발생하고, push방식은 클라이언트가 off되어 있거나 업그레이드 도중 오류가 발생하면 업그레이드가 되지 않는 클라이언트들이 발생할 수 있는 문제점을 가지고 있다.

따라서, 이러한 문제점들을 보안하기 위한 새로운 데이터 전송 방식의 필요성이 대두된다. 본 논문에서는 기존의 데이터 전송 방식인 polling 방식과 push 방식의 단점을 보완한 Half-Push/Half-Polling 패턴을 제안한다.

Half-Push/Half-Polling 패턴은 서버의 업데이트 시간 배분과 스케줄링 기법을 이용하여 서버의 과부하를 최소화하고 모든 클라이언트에 적합한 업데이트를 지원하는 데이터 전송 방식이다.

1. Example

네트워크를 통해 지속적인 업그레이드를 해주는 홈 네트워크를 개발 중이라고 가정하자.



2. Context

클라이언트/서버 모델을 기반으로 데이터 전송 처리 시스템 Architecture를 설계해야 한다, 그리고 서버의 과부하를 줄이고 시스템 정지나 오류로 인해 업그레이드가 중지된 client도 추후에 업그레이드를 해줘야 한다.

3. Problem

고객들에게 지속적으로 최신의 서비스를 지원하기 위해 업그레이드를 지원하기 위한 데이터 전송하는 시스템을 구축한다고 상상해보자. 서버의 리소스가 큰 제약사항이 아닌 경우를 제외하면, 서버의 리소스가 한정적이기 때문에 리소스를 효율적으로 사용하기 위해 일반적인 데이터 전송 방식인 polling이나 push 방식을 시스템에 적용할 것이다. 하지만, polling 방식을 적용하게 되면 주기적으로 항상 server 버전을 체크하고 동시에 많은 클라이언트들이 업그레이드를 요청하면 server와 client 모두 과부하가 발생한다. 또한, 모든 클라이언트들이 동일한 버전을 유지해야만 하는 단점이 있다. 만약, push 방식을 적용하면 스케줄링을 이용하여 과부하를 피할 수 있지만 server가 업데이트를 진행할 때, 클라이언트의 전원이 꺼져있거나 업그레이드 중 오류가 발생하면 업그레이드가 불가능해지는 상황이 발생한다.

따라서, 스케줄링 기법을 이용하고 클라이언트의 상태를 관리하는 효율적인 데이터 전송 방식을 구상해야 한다. 이러한 전송 방식을 적용함으로써, server의 과부하를 줄이고 모든 클라이언트를 업그레이드를 할 수 있다. 이 시스템을 설계하기 위해 다음과 같은 forces가 고려되어야 한다.

Force (요구사항)

큰 전제 - Push방식과 Polling 방식의 단점들을 극복하고, 장점들은 유지해야 한다.

- 서버의 Throughput을 고려하여, 적절히 과부화를 분산하여 업그레이드를 할 수 있어야 한다.
- 특정 Client만을 선택해서 버전을 업그레이드 할 수 있어야 한다.
- 중지된 Client도 추후 업그레이드 가능해야 한다.
- 다양한 종류의 Client들을 업그레이드 할 수 있어야 한다.

4. Solution.

앞에서 언급한 Push와 Polling 기반의 업그레이드 방식이 가져오는 단점들을 극복하기 위해, 이 두 가지의 혼합 모델인 Half-Push/Half-Polling 패턴을 소개한다.

4.1 Structure

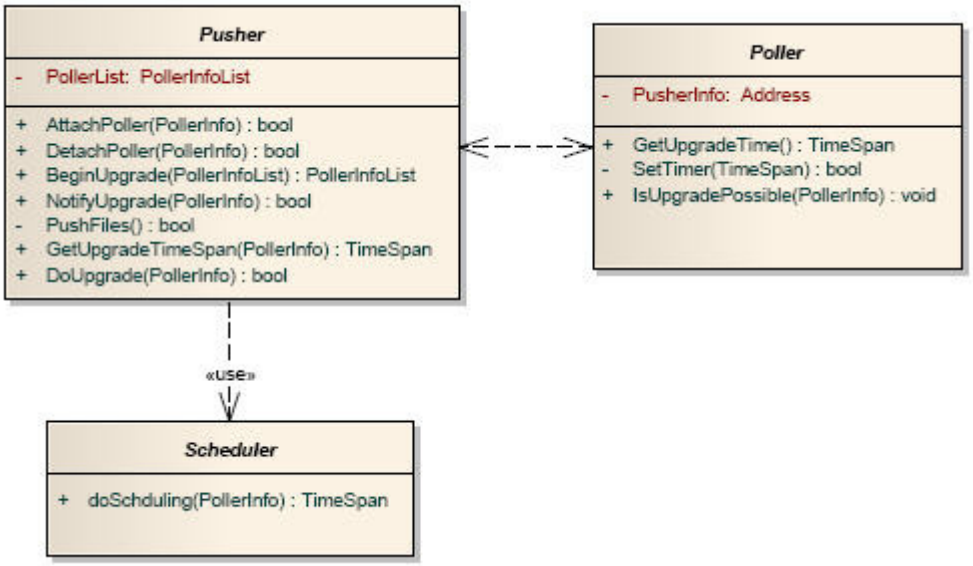


그림 1. Half-Push/Half-Polling Structure

Pusher 컴포넌트는 기존의 Push 업그레이드 방식에 사용되는 Push 기능을 확장한 컴포넌트이다. 이 컴포넌트는 업그레이드를 요청하는 Poller의 목록을 관리하고, 업그레이드 시간을 배분한다.

- AttachPoller – 새로운 Poller를 등록하는 메소드로, Scheduling 대상 목록인 PollerList에 추가한다.
- DetachPoller – 등록된 Poller를 제거하는 메소드로, Scheduling 대상 목록인 PollerList에 제거한다.
- BeginUpgrade – 업그레이드를 시작하는 Entrypoint로, 업그레이드를 요청하는 external source에 노출되는 인터페이스다.
- NotifyUpgrade – Pusher가 선택된 Poller에게 업그레이드 명령을 알린다.
- GetUpgradeTimeSpan – Pusher로부터 업그레이드 요청을 받은 Poller가 다시 Pusher에게 언제 업그레이드를 할지 시간을 배정한다
- DoUpgrade – Poller가 Scheduler를 통해 배분 받은 시간(GetUpgradeTimeSpan()의 리턴 값)이 0이면, Upgrade 하기 위해 호출되는 메소드로 내부적으로 PushFiles()를 호출한다.
- PushFiles – 실제 업그레이드할 파일 목록을 Poller에게 전달한다.

Poller는 업그레이드가 주기적으로 필요한 컴포넌트로, Pusher를 접근 할 수 있는 주소를 항상 가지고 있어야 한다.

- GetUpgradeTime - Pusher로부터 Upgrade할 시간을 받는다.
- SetTimer ~~(SetUpgradeTimer로 변경예정)~~ – Pusher로 부터 배정받은 시간을 저장한다. 타이머나 Watcher를 통해 배정받은 시간이 되면, 바로 업그레이드를 실행하지 않고 현재 서버의 처리량(Throughput)이 한계점을 도달했을 경우를 대비해 업그레이드가 가능한지 Pusher의 GetUpgradeTimeSpan()를 호출한다.

Scheduler 컴포넌트는 업그레이드 시간을 배분하여 여러 가지 스케줄링 전략을 사용할 수 있는

컴포넌트다. 업그레이드 대상이 되는

4.2 Dynamics

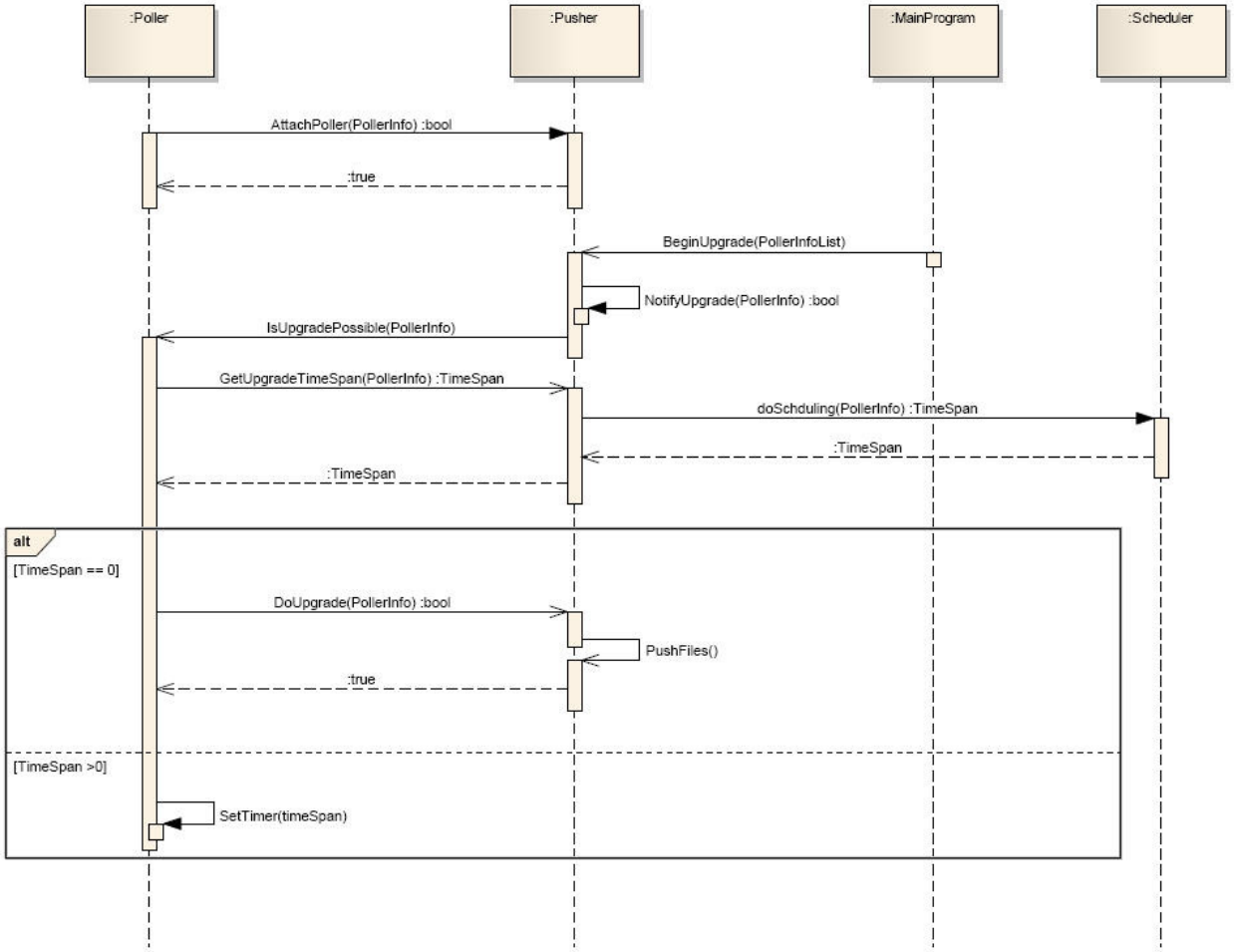


그림 2. Half-Push/Half-Polling 업그레이드 순서도

Publisher-Subscriber[7]처럼, Poller(클라이언트)는 업그레이드를 받고자 하는 Pusher(서버)에게 자신의 Reference 를 전달하여 등록/ 해지할 수 있다.

외부 Upgrade 요청 프로그램에서 특정한 디바이스나 그룹 군들을 선택하여 Upgrade 명령인 BeginUpgrade()을 Pusher 에게 전달한다.

그러면 Pusher 는 (IsUpgradePossible)를 통해 Poller 들에게 업그레이드 정보들(버전 등.)을 전달하고, Poller 는 자신의 정보와 비교하여 Upgrade 가 필요한 상위 버전 정보가 날라오면, GetUpgradeTime()을 호출한다. 그러면 내부적으로 Scheduler 를 호출하여 시간 간격을 배정받게 된다. 만약 오후 3 시 4 분과 같은 특정 시간으로 업그레이드 시간을 전달하게 되면 Pusher 와 Poller, 또한 Poller 간에 복잡한 시간 동기화[9][10] 매커니즘이 필요하게 되므로, 이러한 문제를 피하기 위해 300ms , 4000ms 와 같은 시간 간격(Timespan)을 사용한다.

리턴 받은 업그레이드 배정 시간(Timespan)이 0 이면 바로 업그레이드를 수행 (DoUpgrade())하게 된다. (이 예에서는 Push 방식으로 파일을 전송하지만, Poller 에서 Pusher 의 FTP 의 파일을 가져가도 상관없다.) 하지만 서버의 배정 시간 차가 양수이면 SetTimer()를 이용해 배정 시간 만큼 기다린다. 배정 시간이 되면, 업그레이드 (DoUpgade())를 바로 호출하지 않고, 업그레이드 시간을 다시 배정(GetTimeSpan)을 받는다. 왜냐면 서버의 Throughput(처리량)이 임계점에 도달했을 경우, 서버의 가용성을 위해 새로운 시간을 다시 배정한다.

5. Implementation

1 단계: 업그레이드 대상(Poller)의 특성을 파악해라.
항상 네트웍에 연결되어 있는 Client (Poller)들은 Push 형태(NotifyUpgrade)로 업그레이드 하고자 하는 대상을 지정하는 것이 가능하지만, Client가 네트웍에 연결된 시간이 일정하지 않거나, P2P 처럼 클라이언트가 쉽게 사라지는(departure) 도메인(예를 들어 SDR[11])의 Client(Poller)들은 Polling 형태로 서버에 접속해서 업그레이드 스케쥴링을 정보를 받아오는 것이 좋다.

2 단계: 스케쥴링 알고리즘을 정한다.
업그레이드 시 중요시 되는 QoS를 파악해야 한다. Realtime 시스템과 같이 Deadline이 중요한 스케쥴링 방법[1], 다양한 버전들을 다루는 스케쥴링[12]등, 여러분의 시스템이 추구하는 QoS에 맞는 스케쥴링 방법을 선택해야 한다.

3 단계: QoS를 고려해 메시지 교환 포멧을 결정해라.
상호 운영성을 위해 XML과 같은 표준 메시지 포멧을 사용하면, Marshaling/Unmarshaling 과 같은 변환 작업이 요구 되어지는데, 제한적인 리소스에서 빠른 응답을 요구하는 시스템에는 적합하지 않을 수 있다. 이런 경우는 Protocol 기반의 메시지 전송이나 시스템에 의존적이지만 성능을 보장하는 COM+, OLE와 같은 Binary Method Table[7]을 사용할 수 있다.

4단계: Poller가 Pusher에게 전달해야 하는 정보들을 확장 가능하게 구축해라.
어떠한 스케쥴링 알고리즘을 사용하느냐 또는 Poller의 특성에 따라, Pusher에게 전달하는 정보 역시 영향을 받게 된다. 이질적이며, 다양한 종류의 Poller가 계속 추가 될 경우에는, 주고받는 정보들을 확장과 조합이 가능하게 설계해야 한다. 이럴 경우는 주고 받는 메시지를 확장할 수 있는 Composite Message[4] 나 Parameter Object[5] 패턴을 고려할 필요가 있다.

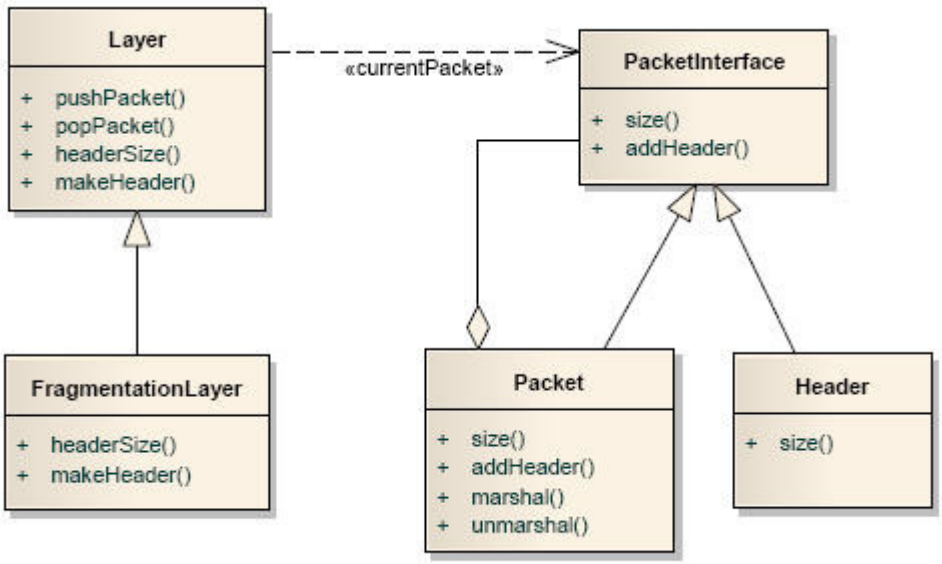


Figure 3. Composite Message 패턴

Composite Message는 Pipe나 Filter를 통해 쉽게 메시지를 추가하고 프로토콜을 교체 및 확장할 수 잇는 패턴이다. 단 런타임 시에 Pipe & Filter를 구축하여 메시지를 생성하므로, 리소스가 제한적이며 빠른 응답을 요구하는 Embedded System에서는 적합하지 않을 수도 있다.

5 단계: 종류별, 그룹별 또는 의존성을 가지는 Poller들 끼리 업그레이드 하는 경우를 고려해야 한다.

업그레이드 시 한 종류의 Poller들만을 고려할 수 없는 경우(다양한 종류의 MP3 디바이스를 가진 업체들, 또는 다양한 전자 제품의 버전을 관리해야 하는 제조 업체들 등.)가 많다. 또한 서로간에 업그레이드 의존성을 가지는 Poller들 끼리 별도로 업그레이드를 할 수 있어야 한다. 이런 경우는, Figure 4처럼 Pusher와 Poller사이에 중개자인 Event Channel [1]을 두어야 한다. 또한 더 복잡한 전달을 요구하는 상황에서는, Channel안에 다양한 Pipe와 Filter들을 두는 것도 고려해야 한다.

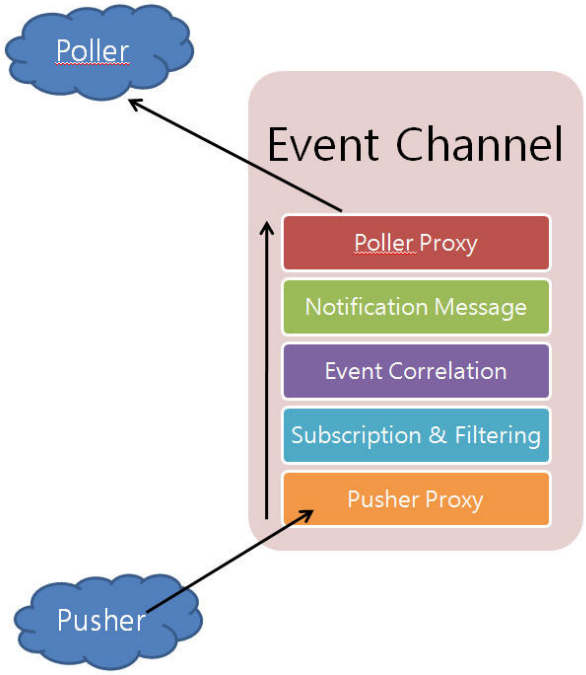


Figure 4. Event Channel를 이용한 예

```
class EventChannel
{
.....
//Poller를 관리하는 리스트
   private List ConsumerList;
   private List FilterList;

   public bool AttachConsumer();
   public bool DetachConsumer();

   public bool AttachFilter();
   public bool DetachFilter();

   //EventChannel 서비스를 시작한다.
   public bool Run();

   //Poller의 정보 즉 디바이스 고유ID, Type 정보, 상태 정보, App 버전, Framework 버전 정보 등을 전송
   public void NotifyUpgrade();
   };
```

위 소스에서 언급하지 않았지만 Pusher는 Poller를 직접 관리하는 대신 Event Channel들을 관리하게 되고, Event Channel에게 Upgrade 명령을 내리기만 하면 된다. 각각의 Event Channel들은 다음과 같은 Filter들의 조합으로 구성될 수 있다.

- 업그레이드 하고자 하는 Poller의 타입을 지정할 경우 - "NodeType:3536" , "NodeType:3836"-
- 업그레이드 하고자 하는 세대(Resident)의 정하고자 하는 경우 - "AptNum:0101"
- 업그레이드 하고자 하는 아파트 세대(Resident)의 범위를 정하고자 하는 경우 - "Range:AptNum:AptNum:01010101:AptNum:11011101"

6 단계: Pusher에게 전달받은 시간 정보를 관리하기 위해 Timer나 WatchDog을 이용해라.
만약 Pusher와 Poller가 동일한 시간대를 가져야만 동작한다면, 복잡한 시간 동기화 메커니즘이 필요할 것이다. 하지만 Half-Push/Half-Polling 패턴은 시간 차 (예를 들어 50초 후, 500초 이후)를 이용하기 때문에 동기화 매커니즘을 사용할 필요가 없다. 대신 이러한 시간 차를 저장하고 해당 시간에 호출되는 Timer나 Watcher(WatchDog)[2]를 사용해야 된다. 또한 시스템의 정지를 고려하여, 현재 시간과 시간 간격 (TimeSpan)을 파일이나 DB에 저장할 필요가 있다. 시스템이 다시 시작되면, 이 정보를 가지고 언제 업그레이드를 요청하는지를 알 수가 있다. 현재 시간이 이들 값을 더한 값보다 크면 다시 Pusher에게 Upgrade가 가능한지를 요청하면 된다.

7 단계: 상황에 맞는 파일 전송 메커니즘을 고려해라.
모든 시스템이 그렇듯이 도메인이나 상황에 맞는 적재적소의 메커니즘을 사용해야 한다. 모든 상황에 맞는 최적의 솔루션(Silver Bullet)은 존재하지 않는다. 다양한 종류의 디바이스를 업그레이드/패치 해야 할 경우, 상황에 따라 다양한 네트워크 환경을 고려해야 할 필요가 있다. 예를 든다면, Session을 유지한 채 대용량의 파일을 전송을 요구하는 Poller가 존재할 수도 있고, 소량의 데이터를 주기적으로 주고 받아야 하는 Poller가 존재할 수도 있다. 이러한 상황들을 고려하여, Poller의 타입 별로 적합한 파일 전송 전략들을 선택해야 한다. 이러한 문제의 해결 방법으로, "JAWS: A Framework for High-Performance Web Servers"[3]의 연구는 훌륭한 시각을 제공한다. 이 연구에서, IOCP와 같은 비동기 전송 메커니즘(Proactor[6])이 사이즈가 작은 파일을 전송 할 때는, 성능이 좋지 않은 것을 볼 수 있다.

6. Example Resolved. (예제 심화)

다양한 종류의 클라이언트(Poller)들을 효율적으로 관리하기 위해서는, 단순히 클라이언트들을 리스트들을 순차적으로 관리하는 것 보다, 다양한 Event Channel을 구축해야 한다. Client의 디바이스 종류 별(모델별), 종속성이 발생하는 것 등으로, Upgrade가 자주 일어나는 형태별로 Event Channel을 구축하면 더욱 쉽게 클라이언트들을 관리할 수 있다.

업그레이드 시간을 단축 하기 위해서, 업그레이드 모듈과 별개로 모든 Poller(Client)들이 네트웍에 Alive되어 있는지를 판별하는 별도의 CheckAliveManager를 구축할 필요가 있다. 업그레이드 시,

종종 꺼져있는 Poller들을 만날 때가 있다. Pusher가 Poller에게 Upgrade 전송시 Timeout 으로 Poller가 죽어 있는 것을 판단하는 것은, 업그레이드 시간을 중지시키는 좋은 주범이 된다. 업그레이드 시, 시스템의 중지 시간을 최소화 하기 위해서 Poller들이 주기적으로 Pusher에게 Alive 메시지를 전송하는 로직이 필요하다. 예를 들어, 주기 * N회 이상의 시간이 지나도 Alive가 오지 않으면, 그 Poller를 Dead 상태로 변경할 필요가 있다.

7. Variant

-OMG CORBA Event Service

RealTime CORBA의 Event 서비스는 업그레이드는 아니지만, 데이터를 전달하는 입장에서는 Push와 Pull (Polling) 이 혼재된 Event Channel 방식을 지원한다. [1]

- Hybrid Push/Pull Download Model in Software Defined Radios.

SDR은 휴대전화, PHS, 무선LAN 등, 출력이나 주파수대역, 변조방식 등이 서로 다른 여러 무선통신 수단을 한대의 무선기기의 소프트웨어를 바꿔 쓰는 것만으로 대응 시키는 기술이다. 기존의 단말기는 새로운 방식의 서비스가 시작되면 새로 구입해서 바꾸는 것이 당연시 되었고, 구세대의 서비스를 위한 인프라와 단말기는 폐기할 수 밖에 없었다. 하지만 SDR은 기존에 반도체가 수행하던 방식에 의존한 처리의 대부분을 소프트웨어로 수행한다. SDR이 제공하는 다운 모델 중 Hybrid Push and Pull[11]은 Half-Push/Half-Polling 패턴의 좋은 사례이다.

- Samsung HomeVita

삼성 전자의 홈 네트워킹 시스템인 Homevita는 다양한 시행 착오 속에서 Push와 Polling 모델을 섞은 업그레이드 방식을 채택했다. 1.0 버전에서는 Polling 방식의 업그레이드 모델을 지원했지만, 동시에 수 많은 디바이스의 업그레이드 요청으로 많은 부하가 걸렸다. 그 후 이러한 문제를 피하기 위해 2.0에서는 Polling 방식으로 업그레이드 모델을 변경하여, 서버의 과부화는 줄였지만 Dead된 디바이스는 업그레이드 되지 않는 문제가 있었다. 그래서 최신 버젼에서는 Push와 Polling 모델의 혼합 모델을 사용하여, 서버의 과부화를 줄이고 alive 상태인 디바이스가 최신 버전 상태를 유지할 수 있게 되었다.

8. Consequence

The advantages of this pattern include:

- Polling 기반의 업그레이드 방식이 가져오는 일순간에 서버에 과부하가 발생하는 문제를 해결하면서, Push 방식이 가져오는 특정 클라이언트만 업그레이드 하는 장점을 취할 수 있다.
- 클라이언트가 주기적으로 서버의 버전을 체크할 필요가 없어, 네트웍 트래픽과 서버/클라이언트의 부하를 줄일 수 있다.
- 그룹화를 하여, 클라이언트만 특별히 선택해서, 업그레이드를 할 수 있다.

Possible disadvantages are:

- 서버의 과부화를 고려한 스케쥴링 방식을 이용하다 보니, 긴급히 모든 디바이스를 업그

레이드 해야 하는 백신과 같은 시스템에서는 적합하지 않다.

- 프로그램 실행 도중 서버(Pusher)에 장애가 발생하면, 클라이언트(Client)들의 업그레이드는 불가능해 진다. Pusher 컴포넌트를 복제하거나, 여러 가지 Fault Tolerance한 기법[2]들을 사용하여, 신뢰성(reliability)를 보장해야 한다.
- P2P와 같이 수시로 서버와 클라이언트 정보가 바뀌는 곳에는 적용하기 어렵다.

9. Related Patterns

Publisher-Subscriber

Observer 패턴의 또 다른 이름으로, 두 컴포넌트 (Publisher, Subscriber)의 정보를 동기화 하기 위해 사용되는 패턴이다. 대표적인 예로 데이터베이스 Publisher-Subscriber 복제를 들 수 있다, 수다쟁이 (non-stop talker) 객체를 만났을 때 즉 끊임없는 Polling으로 과부화가 발생했을 때, 사용하는 패턴이다.

Composite Message

분산 객체에서 데이터를 marshaling/un-marshaling 하는 패턴[4]으로, Layer들을 지나가면서 전송하고자 하는 메시지를 확장/추가하는 패턴이다. Half-Push/Half-Polling 패턴에서는 이질적인 환경을 가진 디바이스 (Poller)별로 전송 Protocol을 생성하기 위해 사용되는 패턴이다. 다양한 분산 미들웨어 에서 사용된다.

Pipe&Filter

상황에 따라, 유연하게 전송하고자 하는 메세지를 추가하거나, 걸러낼 때 (Filtering) 사용되는 패턴으로 앞서 언급한 Composite Message에서 내부적으로 사용되는 패턴이다. Event Channel을 구축 할 때, Poller들을 추가하거나 걸러내기 위해 사용되는 패턴이다.

Broker

서버와 클라이언트간의 직접적인 의존성(위치 정보, 플랫폼 제약등)을 제거하는 패턴이다. Poller가 가지는 Pusher의 위치 정보를 Broker에게 위임함으로써, Poller가 Pusher에 대한 직접적인 의존성을 제거할 수 있다..

References

[1] Timothy H.Harrison, david L. Levine, and Douglas C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," Proceedings of OOPSLA '97, Atlanta, Georgia, October, 1997

[2] Robert S. Hanmer, "WatchDog", Patterns for Fault Tolerant Software, WILEY, 2007

[3] James C. Hu, Douglas Schmidt, "JAWS: A Framework for High-Performance Web Servers", Domain-Specific Application Frameworks: Frameworks Experience By Industry, John Wiley & Sons, October, 1999.

[4] Aamond Sane, Roy Campbell, "Composite Messages: A Structural Pattern for Communication between Components", OOPSLA' 95 Workshop on Design Patterns for Concurrent, Distributed, and Parallel Object-Oriented Systems, 1995.

[5] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, "Refactoring : Improving the Design of Existing Code", Addison-Wesley Professional , 1999
[6] Douglas C. Schmidt, Michael Stal, Hans Rohert, and Frank Buschmann, "Proactor", Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, WILEY, 2000.
[7] Frank Buschmann , Regine Meunier , Hans Rohnert , Peter Sommerlad , Michael Stal, "Broker", Pattern-Oriented Software Architecture Volume 1: A System of Patterns, WILEY, 1996
[9] F. Cristian, "Probabilistic Clock Synchronization", Distributed Computing, vol. 3.
[10] R. Gusella, S. Zatti: "The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3 BSD", IEEE Transactions on Software Engineering, Vol. 15, July 1989
[11] Jamadagni, Satish., Umesh M.N**.,** "A PUSH download architecture for software defined radios"**,** 2000 IEEE international symposium on personal wireless communication
[12] Sameer Ajmani, Barbara Liskov, Liuba Shrira, "Scheduling and Simulation: How to Upgrade Distributed Systems"