<http://bcho.tistory.com/860>

## [비동기 네트워크 서버 프레임웍 Vert.x 소개](http://bcho.tistory.com/860)

[클라우드 컴퓨팅 & NoSQL/Vert.x & Node.js](http://bcho.tistory.com/category/%ED%81%B4%EB%9D%BC%EC%9A%B0%EB%93%9C%20%EC%BB%B4%ED%93%A8%ED%8C%85%20%26%20NoSQL/Vert.x%20%26%20Node.js) | 2014.02.04 23:53 | Posted by 조대협

http://cfile2.uf.tistory.com/image/23255E4A52F78102175A66

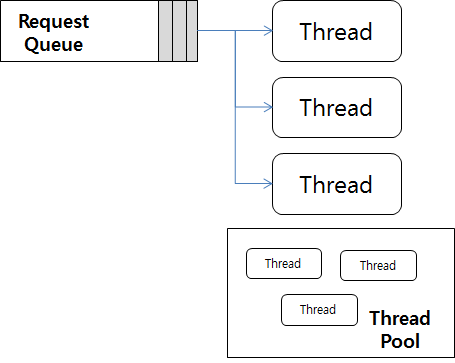
**비동기 네트워크 서버 프레임웍 Vert.x**

**조대협**

**\* 서문**

Vert.x는 NodeJS와 같은 비동기 소켓서버 프레임웍이다.

Vert.x에 대한 이해를 돕기 위해서, Tomcat과 같은 WebApplication Server(이하 WAS)에 대해서 먼저 간단하게 짚고 넘어가자.



<그림. Tomcat의 쓰레드 구조>

Tomcat의 경우에는 HTTP request가 들어오면, request가 앞의 request Queue에 쌓이게 된다.

쌓이게된 request들은 Thread Pool에 있는 Thread에게 하나씩 할당되어, request를 처리하고, 작업이 끝나면, request가 들어온 connection으로 response를 보낸후, 작업을 끝낸다. 작업이 끝난 Thread는 다시 Thread Pool로 들어간다.

이런 구조에서, Tomcat이 순간적으로 동시에 처리할 수 있는 Connection의 수는 Thread Pool의 Thread 수만큼이 되는데, 일반적으로 Tomcat은 50~500개의 Thread정도가 적절하다.

즉, Tomcat 서버 하나는 동시에 최대 500여개의 Connection을 처리할 수 있다고 보면 된다.

요즘 들어, 서비스의 규모와 용량이 커짐에 따라서 동시에 여러개의 Connection을 처리해야 하는 기능이 필요하게 되었는데, 이러한 WAS로는 수십만,수백만개의 Connection을 동시에 처리하는 것이 불가능하다.

또한, HTTP 뿐만 아니라, TCP와 같은 다른 request를 처리해야 하는 여러가지 Protocol 지원 문제도 있고

단순히 request/response 기반의 HTTP 요청 뿐만 아니라, HTTP long polling/Streaming과 같은 Push성 서비스를 구현하려면, 동시에 유지되어야 하는 Connection수가 많아야 한다. (모든 클라이언트가 Conenction을 물고 있기 때문에)

이러한 요구 사항을 반영하기 위해서 나온 서버들이 nodejs와 같은 비동기 소켓 서버이다.

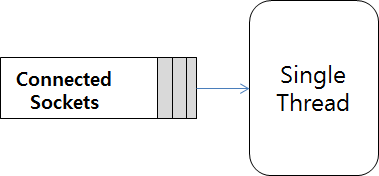
Vert.x는 nodejs의 자바 버전정도로 보면 되는데, 아주 재미있는 것이 구조에서 부터 기능까지 node.js에 비해서 발전된 모습이 매우 많다.

Vert.x는 nodejs와 마찬가지로 single thread model이다.

WAS와 같이 Thread pool을 이용하는 것이 아니라, 하나의 Thread로 모든 작업을 처리한다. Single Thread 모델을 사용할 경우 Thread의 Context Switching 오버헤드를 줄일 수 있어서 성능에 도움이 되고, 또한 Multi threading에서 고민해야 하는 locking 처리나 공유 데이타 처리에 대해서 전혀 걱정할 필요가 없다.

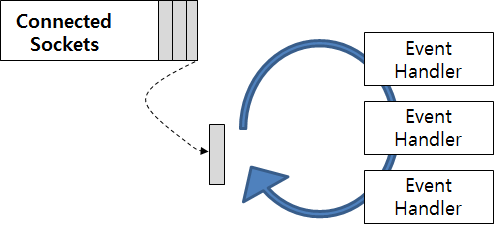
요즘 트렌드인 것 같은데, 요즘 유행하는 고속 서버들 nginx 웹서버나, redis와 같은 IMDB역시 single thread 모델을 사용한다.

Vert.x는 Single Thread로 도는데, 이를 Event Loop(EL)이라고 한다. Vert.x에 일단 클라이언트가 연결되면, EventLoop가 각 연결된 개별 socket들에 대해 event를 검사한후에, event가 있으면 그 function을 수행해준다.



<그림.Vert.x의 Single Thread 구조>

예를 들어서, 클라이언트 A,B,C와 소켓 SA,SB,SC 로 연결이 되어 있다면. Single Thread에서 Event Loop가 돌면서, 첫번째는 SA에 대한 loop를 돌고, 두번째는 SB,세번째는 SC에 대한 루프를 돈다. 루프를 돌다가 소켓에서 메세지가 들어오거나 연결이 끊기거나 기타 이벤트가 있으면, 그 이벤트를 처리한다.



< 그림. Event Loop의 개념>

모든 연결된 소켓에 대해서 Event loop가 도는데, 보통 하나의 Event Loop를 처리하는 Thread에서 10,000~20,000개 정도의 Connection을 처리하는데는 큰 무리가 없다. (백기선氏 감사)

물론, 이벤트 발생 비율이 많고, 이벤트를 처리하는 로직에 소요되는 시간이 높다면 당연히 성능은 떨어진다. 전체 클라이언트로 메세지를 Broadcast 하면, 당연히 성능이 내려가겠지만, 일반적인 P2P 형식의 이벤트 처리에는 큰 무리가 없다.

**\* Vert.x 특징 들여다 보기**

그러면 Vert.x의 몇가지 기술적인 특성을 살펴보도록 하자.

**1. 내부 컴포넌트 구성**

Vert.x를 만든 사람들이 대단하다고 느끼는 점은, 다 처음부터 개발한 것이 아니라 대부분의 모듈을 기존의 오픈소스들을 기반으로 해서 개발하였다는 것이다.

고속 네트워크 처리를 위해서는 국내 개발자 이희승씨가 만들어서 더 유명한 Apache Netty를 서버 엔진으로 사용하고 있고, Vert.x 노드간의 통신을 지원하기 위해서, 데이타그리드 솔루션인 HazelCast를 사용하고 있다.

 클러스터링과, 네트워크 고속 처리 부분을 신뢰가 가는 오픈 소스를 기반으로 했으니, 안정성이 좋은 것은 두말할 나위가 없다.

**2. 다양한 언어 지원 (Node.js는 Javascipt만 된다.)**

Vert.x에서 돌아가는 프로그램을 Verticle이라고 하는데, 쉽게 생각해서 Java의 서블릿(Servlet)으로 생각하면 된다.

흥미로운 점은 Verticle은 JVM 위에서 구동하는 여러가지 언어로 구현이 가능하다는 것이다. 현재까지 Java,JavaScript,Python,Groovy,Scala 를 지원하고 있다.

실제로 써보면 알겠지만, 정말 재미있다. Python으로 구현했다가 필요하면 Java로 구현했다가. :)

**3. 클러스터링 지원**

Vert.x는 클러스터링 지원이 가능하다. 여러개의 Vert.x 서버(instance)를 띄워서 구동이 가능하고, Vert.x 서버간에 통신이 가능하다. (메세지를 보낼 수 있다.)

단, Vert.x 인스턴스간의 데이타 공유는 아직까지는 불가능하다. 백엔드에 HazelCast 데이타 그리드 (일종의 클러스터된 공유 메모리)를 사용하고 있으니, 조만간에 Vert.x 인스턴스간 데이타 공유가 가능하지 않을까 싶다.

Vert.x의 기능을 사용하지 않더라도, 내장되 HazelCast 기능을 사용하면, 인스턴스간의 데이타 공유 구현은 가능하다.

※ 참고로, Vert.x에 embedded된, HazelCast는 Community 버전이다. (무료 버전). HazelCast는 자바 기반이기 때문에, 대용량 메모리를 사용하게 되면, Full GC가 발생할때, 시스템의 순간적인 멈춤 현상이 발생하기 때문에, 이를 감안해서 사용하거나, 또는 상용 버전을 사용하면 Direct Memory라는 개념을 사용하는데, 이는 Java Heap을 사용하지 않고, Native 메모리를 바로 접근 및 관리 함으로써, 대용량 메모리를 GC Time이 없이 사용할 수 있다. HazelCast도 기능상으로 보면 대단히 좋은 솔루션이기 때문에, 한번 살펴보기를 권장한다.

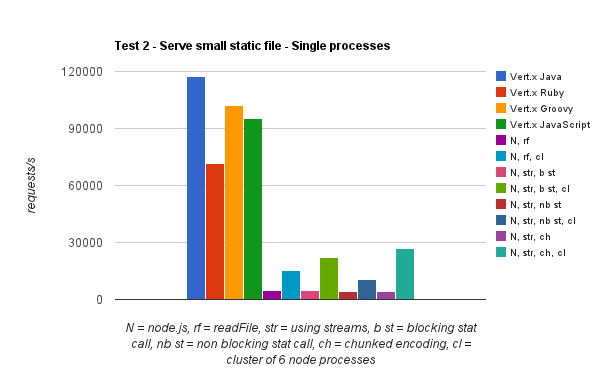
**4. 멀티 인스턴스 지원을 통한 Node.JS보다 빠른 성능**

Vert.x의 ELP는 Single Thread에서 동작하지만, 동시에 여러개의 ELP를 띄울 수 있다. Multi Thread를 띄워서, 동시에 여러개의 Verticle을 실행할 수 있다는 이야기다. 자칫하면 WAS의 Multi threading 모델과 헷갈릴 수 있는데, 여러개의 Thread를 띄우더라도, 각 Thread는 독립적은 ELP를 가지고 동작하고, WAS의 MultiThread 모델과는 다르게, 각 Thread간에 객체 공유나 자원의 공유가 없이 전혀 다르게 독립적으로 동작한다.

이렇게 하나의 Instance에서 여러개의 Thread를 띄울 수 있기 때문에, Multi Core Machine에서는 좋은 성능을 낼 수 있다. Node.js의 경우 Single Thread기반의 ELP를 하나만 띄울 수 있기 때문에, Core수가 많아서, 이를 사용할 Thread가 없다. 그래서 Core가 많은 Machine에서 성능이 크게 늘어나지 않는 반면에, Vert.x는 동시에 여러개의 Thread에서 여러개의 ELP를 수행하기 때문에 성능이 더 높게 나온다. 물론 Node.js도 여러개의 Process를 동시에 띄워서 여러개의 Core를 동시에 사용할 수 는 있지만 Process의 Context Switching 비용이 Thread의 Context Switching 비용보다 크기 때문에, 여러개의 Thread 기반으로 동작하는 Vert.x가 성능에서 유리할 수 밖에 없다.

아래는 Node.JS vs Vert.x의 성능을 비교한 자료로

http://vertxproject.wordpress.com/2012/05/09/vert-x-vs-node-js-simple-http-benchmarks/



small static file을 서비스 하는 성능 측정 결과이다.

http://vertxproject.files.wordpress.com/2012/05/chart\_3-5.png?w=640

**5. Embedded Vertx**

Vert.x는 자체가 서버로써 독립적으로 동작할 수 있을 뿐만 아니라 라이브러리 형태로도 사용이 가능하다. 즉 Tomcat같은 WAS에 붙어서 기동이 될 수 있다. 하나의 JVM에서 Tomcat 서비스와 Vert.x 서비스를 같이 수행하는 것이 가능하다. 예를 들어서 일반적인 HTTP Request는 Tomcat으로 처리하고, Socket.IO나 WebSocket과 같은 Concurrent connection이 많이 필요한 Request는 Vert.x 모듈을 이용해서 처리하게할 수 있다.

**\* Vert.x 개념 잡기**

그럼 이제 부터 Vert.x의 개념에 대해서 알아보자. Vert.x를 공부하면서 가장 힘들었던 점이 이 부분이다. 메뉴얼이나 기타 설명된 자료들이 아주 함축적이어서, 직접 테스트와, Thread dump, Class Loader등을 통해서 분석한것과 Vert.x google group를 통해서 분석한 내용들을 정리 하였다.

**1. Vert.x instance**

Vert.x instance는 하나의 Vert.x 서버 프로세스로 보면 된다. 하나의 Vert.x JVM 프로세스를 하나의 Vert.x instance라고 이해하면 된다.

**2. 일반 Verticle (aka. standard verticle, ELP verticle)**

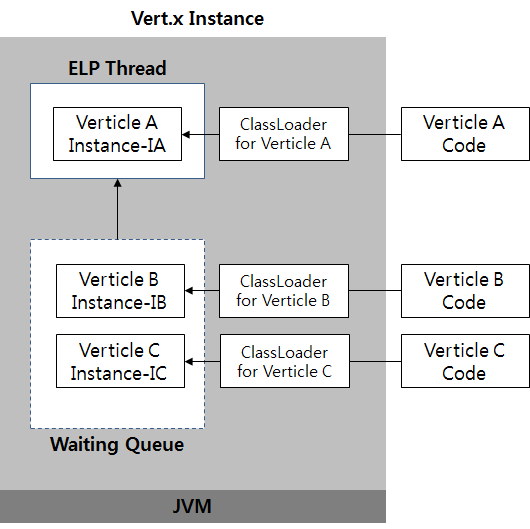
먼저 Verticle이다. Verticle은 Vert.x에서 수행되는 하나의 프로그램을 이야기 한다. 앞서 설명했듯이, 자바의 Servlet과 같은 개념으로 이해 하면 된다.

**3. Verticle instance**

Verticle 코드가 로딩되서 객체화 되면, 이를 Verticle instance(하나의 Verticle Object)라고 한다.

Verticle은 ELP안에서 수행이 되며, 항상 같은 쓰레드에서 수행이 된다. Verticle instance는 절대 multi thread로 동작하지 않고, single thread에서만 동작한다. (중요)

개념 설명을 돕기 위해서 아래 그림을 보자.



하나의 Vert.x instance는 하나의 JVM위에서 동작한다.

Verticle Code는 각 Verticle instance당 할당된 Class Loader에 의해서 로딩되고, 객체화가 된다. 위의 그림은 Verticle A,B,C 3개가 있고, 각각의 클래스 로더에 의해서 로딩된 경우이다. 그리고 ELP Thread가 하나인 경우인데, 이 경우, Verticle Instance 들은 Queue에서 기다리게 되고, 순차적으로 ELP Thread를 점유하면서 순차적으로 실행된다. 단 수행되는 순간에는 Single Thread 모델이기 때문에, 하나의 Verticle만 그 순간에는 수행된다.(즉, Verticle이 100개이고, ELP가 1개라도 모든 100개의 Verticle은 이 ELP 한개에서 순차적으로 수행된다.)

Vert.x에서는 CPU Core수에 따라서 자동으로 ELP Thread를 생성해준다.

ELP Thread 수를 지정하려면 vertx.pool.eventloop.size를 System Property로 주면 된다. 참고로 ELP Thread의 수는 CPU수 보다 작아야 Thread Context switcing이 발생하지 않아서 조금 더 높은 성능을 낼 수 있다.

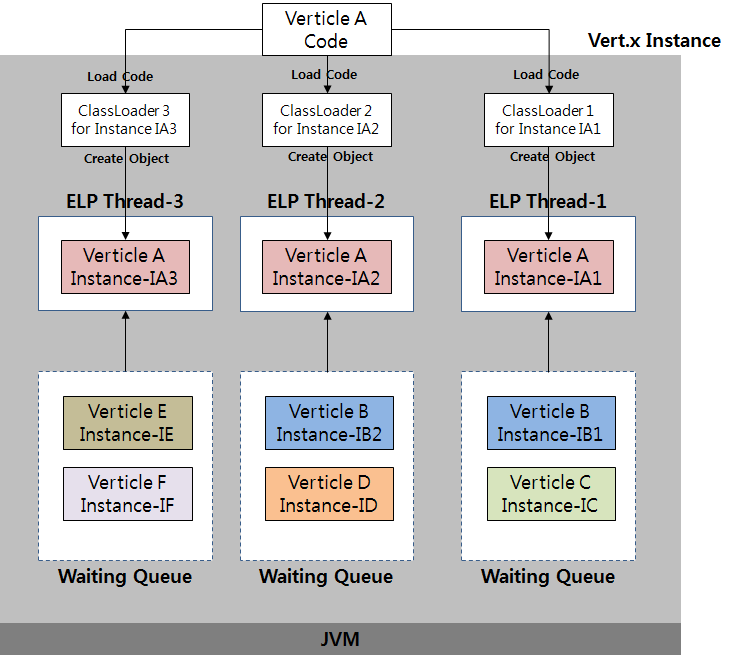


<그림. Vert.x 소스코드중 ELP 쓰레드를 생성하는 코드>

Vert.x 엔진 코드 상으로 보면 CPU Core수의 2배로 ELP Thread를 생성하는데, 실제로 테스트 해보니, 8 Core에서 ELP Thread가 2개 밖에 안생긴다. (몬가 이상한걸?)

동시에 여러개의 ELP Thead를 생성할 수 있기 때문에 같은 Verticle이라도 여러개의 Thread에서 수행할 수 있다. 여기에서 헷갈리지 말아야 하는 것이. 같은 Verticle Code일지는 몰라도, Verticle Instance는 각기 다른 인스턴스이다.

아래 그림에서와 같이 Verticle A Code에 대한 Verticle instance를 3개를 생성하게 되면, 각각 다른 Class Loader에 의해서 Verticle instance가 생성된다. (자바의 같은 Servlet코드를 다른 3개의 war 파일로 패키징되는 것과 같음). 그래서 JVM 입장에서는 이 3개의 Verticle Instance는 전혀 다른 Object가 되고, 각각 다 독립된 ELP에서 다르게 수행이 된다. 그래서 전혀 Verticle Instance간에 공유하는 것이 없고 결과적으로 Thread Safe하게 수행된다.



Verticle을 배포할 때 instance수를 정할 수 있는데, 위의 그림은 Verticle A는 3개, B는 2개, C,D,E,F는 각각 1개의 Verticle Instance를 배포한 예이다. 각 Instance들은 ELP의 Queue로 들어가서 항상 같은 Thread에서만 수행된다.

**4. Worker Verticle**

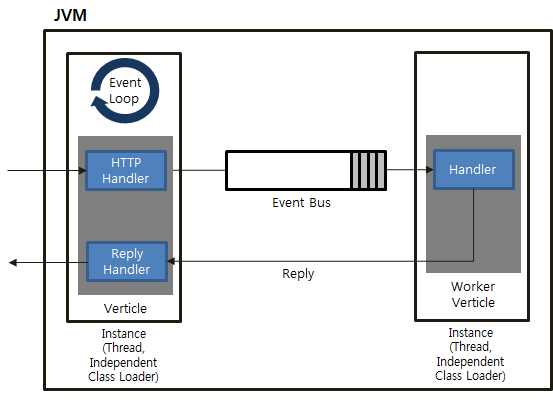
앞서 설명했듯이, 일반 Verticle은 개별 socket에 대해서 ELP를 돌면서 Event 처리를 한다. 그래서 해당 Event 처리에 오랜 시간이 걸리면 전체적으로 많은 성능이 떨어지기 때문에, 문제가 된다.

예를 들어 하나의 event 처리에 100ms 가 걸리는 작업이 있다면, 연결된 Connection이 100개가 있을 경우 전체 socket에 대해서 한번 이벤트 처리를 하는데 10초가 걸리고, 처리가된 소켓이 다음 이벤트를 받을 수 있을 때 까지 10초가 걸린다. 즉 클라이언트 입장에서 연속적으로 요청을 받았을때, 두번째 요청에 대해서 응답을 받는 것은 10초후라는 이야기가 된다. 이는 Single Thread 모델이기 때문에 발생하는 문제인데, Vert.x에서는 이런 문제를 해결하기 위해서 Worker Verticle 이라는 형태의 Verticle을 제공한다.

이 Worker Verticle은 쉽게 생각하면, Message Queue를 Listen하는 message subscriber라고 생각하면 된다. Vert.x 내부의 event bus를 이용해서 메세지를 보내면, 뒷단의 Worker Verticle이 message Queue에서 메세지를 받아서 처리를 한 후에 그 결과 값을 다시 event bus를 통해서 caller에게 보내는 형태이다. (Asynchronous call back 패턴)

event bus로 request를 보낸 Verticle은 response를 기다리지 않고, 바로 다음 로직을 진행하다가, Worker Verticle에서 작업이 끝난 이벤트 메세지가 오면 다음 ELP가 돌때 그 이벤트를 받아서 응답 메세지 처리를 한다.

DB 작업이나, 시간이 오래 걸리는 작업은 이렇게 Worker Verticle을 이용해서 구현할 수 있다.



**5. Worker Verticle instance & Thread pooling**

이 Worker Verticle의 재미있는 점은 Thread Pool을 지원한다는 것이다. ELP Verticle과 마찬가지로, Worker Verticle instance는 독립된 클래스 로드로 로딩되어 생성된 객체로 쓰레드에서 각각 다른 독립된 instance가 수행되기 때문에, Thread safe하다.

단 Worker Thread의 경우 공용 Thread Pool에서 수행된다. ELP Thread의 경우 Verticle은 항상 지정된 Thread에서 수행되는데 반해, Worker Verticle Instance는 Thread Pool의 아무 쓰레드나 유휴 쓰레드에서 수행이 된다.

**6. Event Bus**

Event Bus는 일종의 Message Queue와 같은 개념으로, Verticle 간에 통신이나, Vert.x Instance간의 통신이 가능하게 한다.

Verticle간의 통신이란 Java로 만든 Verticle에서 Python으로 만든 Verticle로 메세지 전달이 가능하다. 또한 Event Bus를 이용하면 서로 분리되어 있는 (다른 JVM 프로세스로 기동되는 또는 서로 다른 하드웨어에서 기동되고 있는) Verticle 간에 메세지를 주고 받을 수 있다.

이 메세지 통신은 1:1 (Peer to Peer) 통신 뿐만 아니라, 1:N (Publish & Subscribe) 형태의 통신까지 함께 지원한다.

여타 일반적인 Message Queue와 다른 특성은 일반적인 Message Queue의 경우 Fire & Forget 형태의 Message exchange pattern을 사용하는데 반해서, Event Bus는 Call back pattern을 사용한다.

즉, JMS나 RabbitMQ의 일반적인 경우 클라이언트가 메세지 큐에 메세지를 넣고 바로 리턴하는데 반해서, Event Bus를 이용하면, 클라이언트가 메세지를 넣고 바로 리턴이 된 다음, 큐를 사용하는 Consumer가 메세지를 처리한 후에, 메세지를 보낸 클라이언트에게 다시 메세지를 보내서, 처리를 하도록 할 수 있다.

예를 들어 하나의 HTTP request가 들어왔을때, event bus에 메세지를 넣고 해당 클라이언트는 내부적으로 리턴이 된다.(HTTP response는 보내지 않고, Connection은 물고 있다.) 그 후에 Worker Verticle에서 작업을 처리한 후 Call back rely를 보내면, 클라이언트에서 이 이벤트를 받아서 HTTP connection으로 response를 보낼 수 있다.

이 구조가 있기 때문에, Single Thread모델임에도 불구하고 뒷단에서 비동기 메세지를 처리하는 방식을 이용하여 long transaction도 핸들링할 수 있게 된다.

**7. Shared data 처리**

하나의 Vert.x instance안에서만 공유 가능하다. (다른 Vert.x 인스턴스간에는 아직까지는 불가능)

ConcurrentMap<String, Integer> map = vertx.sharedData().getMap("demo.mymap");

을 통해서 Map을 사용하거나

Set<String> set = vertx.sharedData().getSet("demo.myset");

을 통해서 Set를 사용할 수 있다.

**8. Module**

모듈은 하나의 Runnable Application으로, 자바의 일종의 WAR 파일이라고 생각하면 된다.

mod.json (WAR 파일의 web.xml 과 같은 메타 정보 description) 에 메타 정보를 정의한 후에, 클래스와 jar 파일등의 라이브러리, 그리고 기타 애플리케이션에서 사용할 파일들을 같이 묶어서 패키징 한 형태이다.

이 Module은 Maven repository 시스템등에 저장 및 배포될 수 있다. (vagrant나 docker 컨셉과 비슷한듯. 이제 Runtime application도 repository를 사용하는 추세인가 보다)

**9. Pumping**

네트워크 Flow Control을 해주는 기능으로,  읽는게 쓰는거 보다 빠르면(예를 들어 socket에서 읽고, 파일에 쓰는 ) Queuing이 많이되고, 메모리 소모가 많아져서 exhausted된다. 이를 막이 위해서 쓰다가 Q가 차면 socket read를 멈추고 있다가 write q가 여유가 생기면 다시 읽는 것과 같은 flow control이 필요한데, 이를 pump로고 한다.

**10. HA (High Availibility)**

Vert.x는 HA 개념을 지원한다. Vert.x에서 HA란, HA mode로 Vert.x instance를 구성했을때, 해당 Vert.x instance가 비정상 종료 (kill등) 되면, 해당 instance에서 돌고 있던 Vert.x Module을 다른 Vert.x Instance로 자동으로 옮겨서 실행해주는 기능이다.

실행시 -ha 옵션을 줘서 실행하는데, 하나의 클러스터 내에서도 hagroup을 지정하여 그룹핑이 가능하다. 예를 들어 클러스터에 10개의 노드가 있을때, 4개는 업무 A용 HA그룹으로 나누고, 6개는 업무 B용으로 나누는 식으로 업무별로 나눌 수 도 있고

Machine이 3개가 있을고, 2개의 업무를 수행하는데, 각 Machine마다 Vert.x instance를 2개씩 돌린다고 했을때

Machine 1: vertx.instance=hagroup\_A , vertx.instance2=hagroup\_B

Machine 2: vertx.instance=hagroup\_A , vertx.instance2=hagroup\_B

Machine 3: vertx.instance=hagroup\_A , vertx.instance2=hagroup\_B

이런식으로 구성을 하게 되면, Machine 단위로 HA가 넘어가도록 구성을 할 수 있다.

**11. Clustering**

HA 없이 클러스터링 구성만을 할 수 있는데, event\_bus로 메세지를 주고 받을려면, 해당 vert.x instance들이 같은 cluster내에 들어 있어야 한다.

※ 설정은 conf/cluster.xml 을 이용해서 하는데, 이 파일은 hazelcast의 클러스터링 설정 파일이다. Default로는 multicast를 이용해서 cluster의 멤버를 찾도록 하는데, 인프라 특성상 multicast가 안되는 경우에는 (아마존과 같은 클라우드는 멀티캐스트를 지원하지 않는 경우가 많음). 직접 클러스터 노드들의 TCP-IP 주소를 적어두도록 한다.

또한 서버의 NIC가 여러개인 경우에는 어떤 NIC를 사용할지 IP주소로 명시적으로 정의해줘야 한다. (유선,무선랜이 있는 경우 등). 직접 주소를 정해주지 않는 경우, HazelCast의 경우 NIC리스트에 있는 순서중 첫번째 있는 주소를 클러스터 주소로 사용한다.

**\* Hello Vert.x**

간단한 Vertx 테스트

**1. 설치 :** Vertx의 설치는 어렵지 않다. http://vertx.io/install.html 를 따라서 하면 되고, 현재 2.1M5 (2014년2월6일 현재) 버전이 최신 버전이다.

주의할점이 인터넷에 돌아다니는 문서 (특히 한글문서)중에서 1.3x로 설치 가이드 하고 있는 문서들이 있다. 특정 코드나 예제가 안돌아 갈 수 있으니, 반드시 최신 버전을 확인하고 설치하기 바란다.

**2. 코드 테스트**

아래와 같이 server.js 라는 코드를 만들고

var vertx = require('vertx');

vertx.createHttpServer().requestHandler(function(req) {

  req.response.end("Hello World!");

}).listen(8080, 'localhost');

다음으로 vertx run server.js를 수행한후

http://localhost:8080 으로 접속하여 Hello World! 가 제대로 나오는지 확인을 한다.

**3. 지원 모듈 확장**

추가 언어 지원 모듈이나, 웹서버 같은 모듈을 인스톨 할려면 vertx install {모듈명} 을 사용해서 인스톨 하면 되는데, 이 경우 maven이나 외부 repository를 통해서 모듈 파일을 읽어와서 설치한다.

이때 회사 방화벽이나 Proxy 설정이 되어 있는 경우, 모듈 파일을 못 가지고 오는 경우가 있다. 이 경우에는 Local Maven directory나 사내에 Nexus와 같은 Maven repository를 설치하고 거기에 모듈을 저장한 후에 설치하도록 한다.

모듈 repository 설정은 conf/repo.txt에 하도록 되어 있다. maven과 bintray repository를 지원한다.

자세한 모듈 설치 방법은 http://vertx.io/mods\_manual.html#installing-modules-manually-from-the-repository 를 참고하기 바란다.

Vertx 모듈 목록은 http://modulereg.vertx.io/ 를 참고하기 바란다.

**\* TCP Push 서버 구현 아키텍쳐**

  다음은 간단하게, Vert.x를 이용하여 TCP/Push 서비스를 구현하는 아키텍쳐를 소개하고자 한다.

구현하고자 하는 시나리오는 TCP 클라이언트들이 접속되어 있을때, Vert.x 서버로 HTTP request를 보내면, 해당 클라이언트로 푸쉬 메세지를 보내주는 방식이다.

**1. Option A. Vert.x 의 클러스터링 기능을 이용하는 방식**

첫번째 구현 방식은 Vert.x의 클러스터링 자체를 이용하는 것이다.

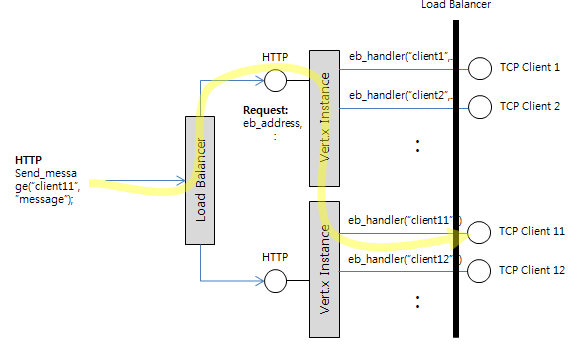
클라이언트가 접속이 되면, 각 클라이언트 socket 마다. event bus handler를 binding해놓는다. client의 id가 되어도 되고, mac address 또는 ip/port 등도 사용할 수 있다.

그리고, 각 Vert.x Instance 마다 Push request를 받는 HTTP/JSON 인터페이스를 개발했다고 하자.

여러개의 Vert.x 인스턴스를 클러스터로 구성하여 배포 하고, Vert.x <--> 클라이언트 간에는 L4나 HAProxy등의 로드밸런서를 둬서 부하를 분산한다.

Push request를 받는 HTTP 단 앞에서 로드밸런서를 둔다.

Request는 client id와 push에 보낼 메세지를 보내주면 HTTP Request는 L4에 의해서 아무 Vert.x 서버에나 라우팅이 될 것이다. HTTP Verticle에서는 client id를 event bus address로 사용하여, event를 보내면, Vert.x 클러스터링에 의해서 해당 클라이언트가 연결된 Vert.x 인스턴스로 이벤트가 발송 되고, 이 이벤트를 받은 Event Bus 핸들러에서 Client로 push 메세지를 보내게 된다.



이 아키텍쳐의 장점은 Vert.x의 기능을 그대로 사용하기 때문에 구현이 간단하다는 것이다.

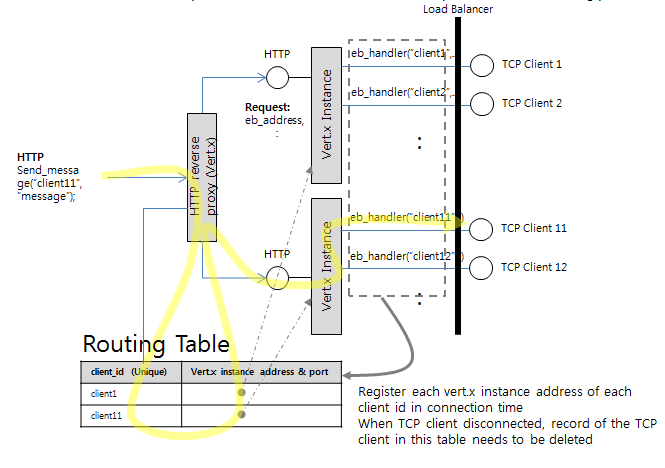
단 Event Bus에 치중한 아키텍쳐이기 때문에, Event Bus의 구현체인 Hazel Cast의 사용량이 많아지게 되고, GC를 유발하여, 순간적인 시스템 멈춤(full gc)를 유발할 수 있다.

시스템이 크지 않으면 큰 문제 없겠지만, 수백만 커넥센등을 동시 처리할 경우에는 HazelCast의 상용 버전에서 지원하는 Elastic Memory라는 기능을 고려해볼 필요하가 있다. 이 기능은 데이타를 Java Heap 영역에 쓰는 것이 아니라, Direct로 메모리에쓰고 관리하기 때문에, GC를 유발하지 않기 때문에, 이 아키텍쳐에서 오는 문제점을 예방할 수 있다.

**2. Option B. - Custom router를 사용하는 방법**

다른 옵션으로는 Vert.x의 클러스터링 기능을 사용하지 않는 방식이다.

클러스터를 사용하지 않는 방법이다. Event Bus 방식과는 달리, Push 요청은 반드시, Client와 TCP로 연결된 Vert.x 인스턴스들로 보내져야 한다. 이를 위해서는 앞단에서 Push Request를 받는 HTTP 서버 앞의 로드밸런서에서, Client가 어느 Vertx Instance에 접속되어 있는지를 알아야 한다. 이를 위해서 일종의 Custom Load Balancer를 Vert.x로 만들어서, TCP Client가 접속될때, 이 Customer Load balancer의 라우팅 테이블에 클라이언트가 어느 Instance로 접속되었는지를 저장해놓고, HTTP Push 요청이 들어 왔을 때, 이 라우팅 테이블을 기반으로 하여, 해당 Vert.x Instance로 요청을 보내는 방법이 있다.



**\* Vert.x에 대한 몇가지 잡담**

* 다들 gradle을 쓰더라.
* StackOverflow보다는 google groups를 이용하자.
* 옛날 Tuxedo와 구조가 유사함. HazelCast는 BBL. Verticle instance는 Tuxedo Service와 유사하다.
* Single Thread 기반의 ELP를 사용하기 때문에, 코딩 잘못하면 까딱하다가 시스템을 보내버릴 수 도. Multithread 모델에서는 문제가 된 Thread만 stuck되고 다른 Thread들은 작업을 하겠지만, Single Thread에서는 한번 Stuck되면, 뒤의 작업들이 줄줄이 막혀 버린다.
* 써보니 정말 편하더라. EventHandler만 구현하면 되기 때문에 코딩양도 적고, 컴파일 없이 바로 실행해볼 수 있기 때문에 개발 및 테스트가 편하다. (진입 장벽이 낮다.)
* 메뉴얼 만드는 사람들은 지원하는 모든 프로그래밍 언어로 만들어야 하기 때문에, 참 힘들겠더라.
* Tim Fox가 VMWare에서 다시 RedHat으로 옮겼다더라.

요 몇일간 Vert.x에 대해서 공부를 한것을 글로 정리하고 싶어서 포스팅 했는데, 생각보다 내용도 많아지고.. 어려워 졌다!!

그렇지만, Vert.x의 경우 상당히 유용하고, 비교적 Learning curve가 높지 않으니 고성능, 대용량 서버 개발이 필요하다면 반드시 살펴보기를 권장한다.