High-Speed Reactive Microservices

- Conference: JavaOne
- Session Type: Conference Session
- Session ID: CON5797
- Speakers: ***Jason Daniel*** and ***Rick Hightower***
- Room: Parc 55—Cyril Magnin I
- Date and Time: 09/19/16, 12:30:00 PM - 01:30:00 PM

High-Speed Reactive Microservices

Jason Daniel (TWITTER @JasonDaniel44)
Rick Hightower (TWITTER @RickHigh)

Introduction

Jason Daniel

Rick Hightower

Background project we worked

What we are working on now

Overview

- Based on Microservices and Reactive programming
- Attributes of high speed reactive microservices
- Example
- How we got here
- Single writer
- Data ownership and leasing
- Service Store
- Related tools

(HSRM - High-speed reactive microservices)

Attributes of HSRM

- HRSM = (H)igh-(S)peed (R)eactive (M)icroservices
- in-memory services
- non-blocking
- own their data (or lease it)
- Scale out often involves sharding services
- Reliability is achieved by replicating service stores
- Calls to other services and service stores are streamed and/or batched
- Implement bulkheads and circuit breakers
- Handle back pressure

Advantages HSRM

Cost - less servers or less cloud resources

Deliver more with same amount of developers

Embrace OOP, data and logic live together

- Cohesive code base
- Ability to react to service calls

Expect to write less code and for it to run faster

High speed services employ the following

- Timed/Size Batching
  - Reduce thread hand-off, sync.,
  - Optimize IO throughput
- Streams
- Callbacks / Promises / Async call coordination
- Call interception to enable data faulting from the service store
- Data faulting for elasticity (like memory paging)
  - like OS virtual memory (loads parts of file into RAM when it finds it is not loaded)
  - Call comes in, call queued, user not loaded, user loaded from service store in next batch, call continues

Call speed, non-blocking calls

- Streams, pipes, sockets
- Bi-directional async communication
- For speed, prefer RPC calls that are non-blocking and can be sent in batches (POST or WebSocket or Streams)
- Dumb fast pipes and batching calls or streams of calls
- Reactive programming

Reactive Manifesto and HSRM

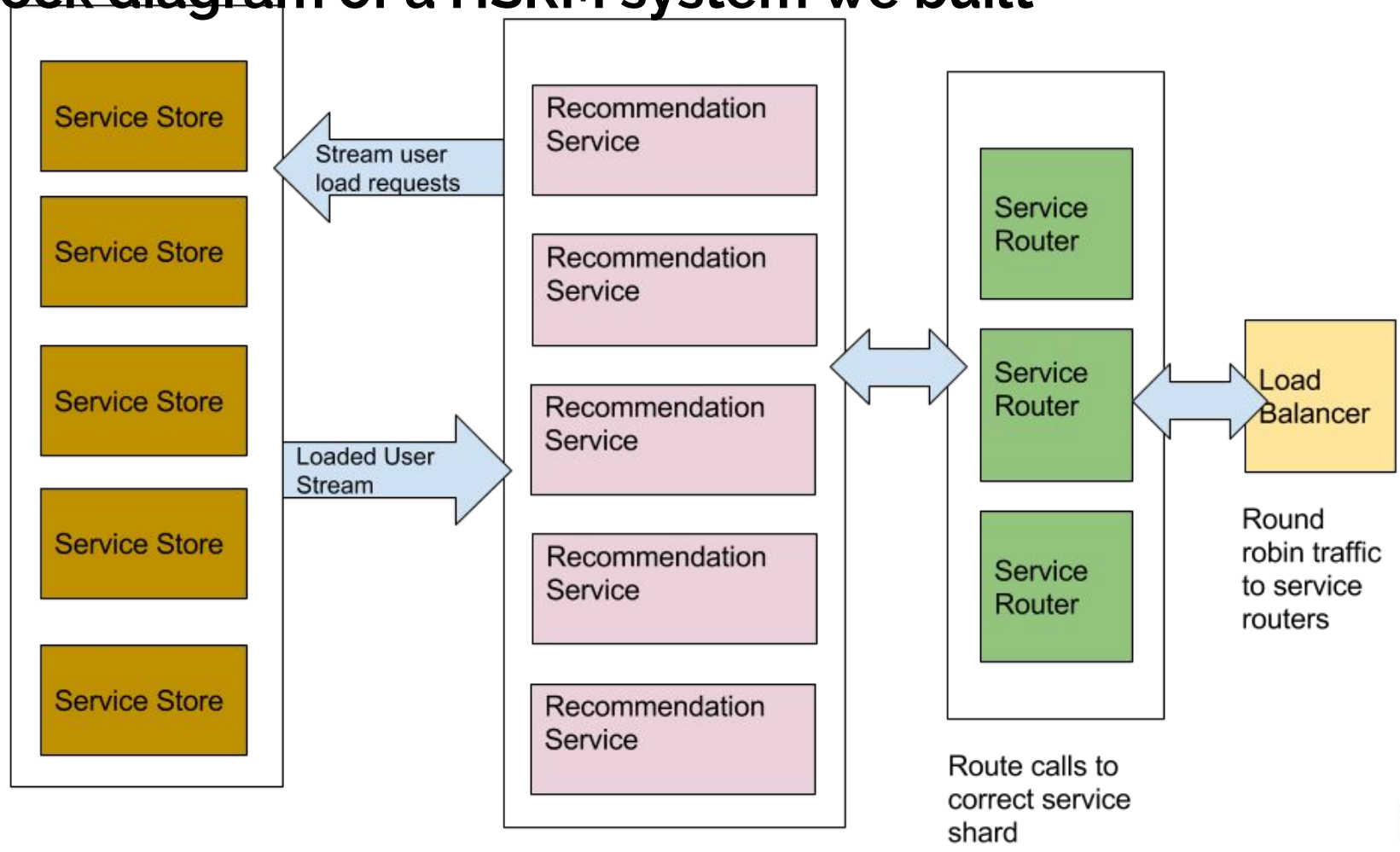
- Responsiveness
  - No hanging, no cascascading failures
- Resilience
  - Service Discovery, Health checks , Reconnect, Recover (Mesos), no SPOF
- Elasticity
  - Find new services, scale, shard, discovery, grow
- Message driven
  - Streams, back pressure, async

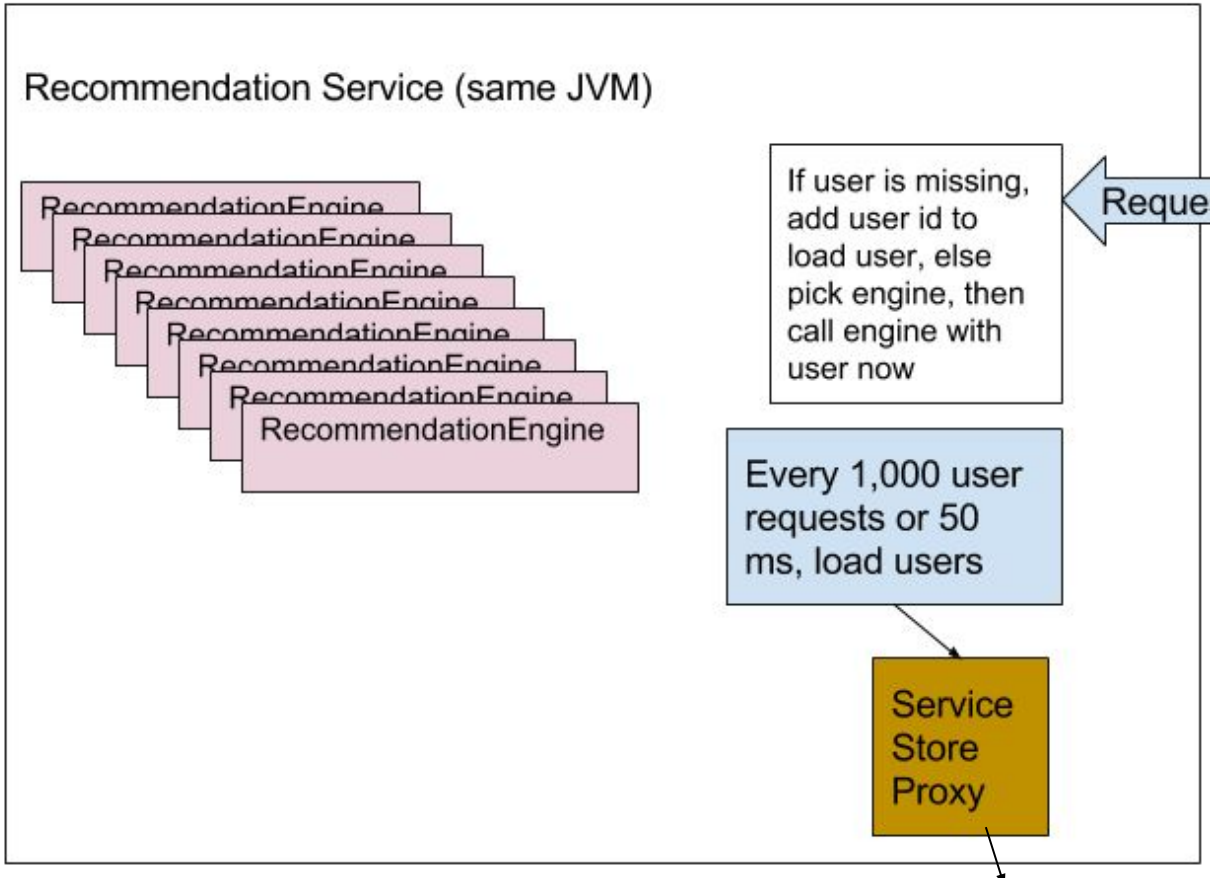
Example

Example Recommendation Service

- Recommendation service
- Watch what user does, then suggest recommended items
- 100 million users
- Recommendation engine can run about 30K recommendations per second per thread
- We run 8 to 16 recommendation engines per microservice (240K to 480K recommendations per second)
- Each recommendation service can handle 200K requests per second
    - Does event tracking which is not CPU bound
    - Does a large calculation which is a big N+1 * N+1 in memory comparison

Block diagram of a HSRM system we built





Call to service is queued and replayed after user is loaded

```
@Service
public class RecommendationService {

    ...
    private final UserStoreService userStoreService = ...;
    private final Reactor reactor = ...;

    @ServiceCall
    public Promise<List<Recommendation>> recommend(final String userId) {
        return invokablePromise(returnPromise ->
                getUser(userId)
                        .ifPresent(user ->
                                pickEngine(userId).recommend(user)
                                        .thenSafe(returnPromise::resolve)
                                        .catchError(returnPromise::reject)
                                        .invoke())
                        .ifAbsent(() -> {
                            loadUserFromStoreService(userId);
                            addOutstandingCall(userId, returnPromise);
                        })
        );
    }

}
```

Async call recommendation engine if user present

If not add user id to call batch, add to outstanding call for user.

Adding an outstanding call (promise)

```
@Service
public class RecommendationService {
...

    private Expected<User> getUser(final String userId) { return expectedNullable(users.get(userId); }
...
    private void addOutstandingCall(String userId, Promise<List<Recommendation>> returnPromise) {
        expectedNullable(outstandingCallMap.get(userId))
                .ifPresent(promises -> promises.add(returnPromise))
                .ifAbsent(() -> {
                    final List<Promise<List<Recommendation>>> list = new ArrayList<>();
                    list.add(returnPromise);
                    outstandingCallMap.put(userId, list);
                });
    }

}
```

```
@Service
public class RecommendationService {
...
    @Init
    private void init() {
        initUserStream();
        //
        reactor.addRepeatingTask(Duration.ofMillis(50), () -> {
            if (userIds.size() > 0) {
                userStoreService.loadUsers(Collections.unmodifiableList(userIdsToLoad));
                userIdsToLoad.clear();
            }
        });
    }

    private void loadUserFromStoreService(final String userId) {
        userIdsToLoad.add(userId);
        if (userIdsToLoad.size() > 100) {
            userStoreService.loadUsers(Collections.unmodifiableList(userIds));
            userIdsToLoad.clear();
        }
    }

    //Work with user stream from service store
    private void initUserStream() {
        userStoreService.userStream(userList -> {
            if (userList.complete()) {
                initUserStream();
            } else if (userList.failure()) {
                //Log & Recover
            } else if (userList.success()) {
                reactor.deferRun(() -> {
                    handleListOfUserFromStream(userListResult);
                });
            }
        });
    }
```

Every 50 ms check to see if the userIdsToLoad is greater than 0, If so request those users now.

When a user is not found loadUserFromStoreService is called. If there are 100, outstanding requests, then load those users now.

Listen to the userStoreService's userStream

```java
@Service
public class RecommendationService {
...

    private void handleListOfUserFromStream(StreamResult<List<User>> userListResult) {
        userListResult.get().stream().forEach(user -> {
            users.put(user.getId(), user);
            expectedNullable(outstandingCalls.get(user.getId()))
                    .ifPresent(recommendationPromises ->
                            recommendationPromises.forEach(recommendationPromise ->
                                    pickEngine(user.getId()).recommend(user)
                                            .thenSafe(recommendationPromise::resolve)
                                            .catchError(recommendationPromise::reject)
                                            .invoke()
                            )
                    )
                    .ifAbsent(() -> {
                        //Log not found when expected
                    });
        });
    }
```

Process the stream result.

Populate the user map (or use a Simple cache with an expiry and a max number of users allowed to be in system).

Since the user is now loaded, see if their are outstanding calls (promises) and resolve those calls.

How we got here. Why High Speed Microservices?

Our interests, background, experience

Been studying high-speed computing (mechanical sympathy)

Working with a media company that built services that can have 100 million users

Worked with Actor systems before

Interest in creating bounded services which could be more accurately bid for fixed bid projects and/or just better project management 

Remote, focused, disparate teams with different experiences and talents working on the same project.

Experience: We were key members on a team that built a 100 million users microservice

Need for speed

Large media company

Very spiky traffic

100 million users

Mobile (iOS, Android), game console, Tivo, AppleTV, etc.

Existing services handling less traffic using 1,000 of servers 

Many services hiding behind CDN (updates take a while)

Had mandate to deliver services that could stand up to load and not cost a fortune to run

Reactor Pattern

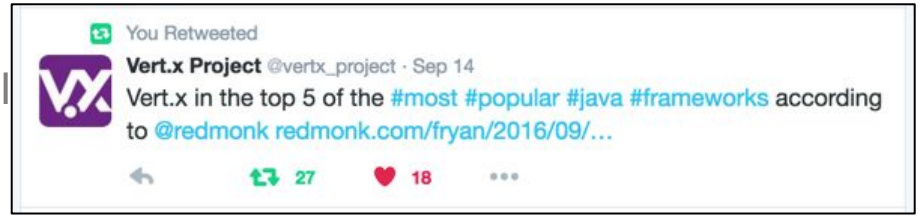
Reactor pattern:

- event-driven,
- handlers register for events,
- events can come from multiple sources,
- single-threaded system
- handles multiple event loops
- Can aggregate events from other IO threads

Implementations of Reactor Pattern

- Browser DOM model and client side JavaScript
- Node.js - JavaScript
- ***Vert.x - Java (Netty also Java)***
- ***Spring 5 - Reactor (not then)***
- ***Java EE - Reactive/Microservice focus***
- Twisted - Python
- Akka's I/O Layer Architecture - Scala
- Swing/GTK+/Windows C API

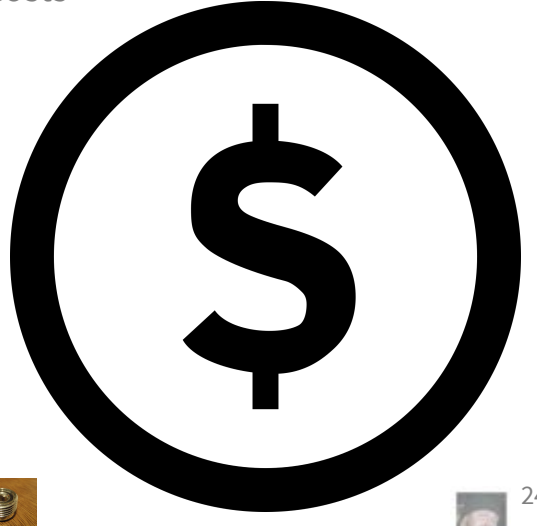
Reactor pattern frameworks do good in the I
the top of the [Techempower benchmarks]



Why High-speed reactive microservices?

Scale out model / Cloud model

- Performance issue, run more instances
  - Knee jerk reaction for some is to address every performance issue with adding more nodes, more cache, more resources
  - Every perf issue can be addressed with same hammer
  - But 2,000 boxes are harder to support than 20 or 10 not to mention costs
- Scale issue, run more instances
- This works but at what cost?
  - More cost
  - More reliability, but more complexity
- Move towards scale up and out (more with less)



Advantages of high-speed microservices

Observed

- Less developer resources
- Less hardware/cloud resources
- Handle more traffic
- Cheaper operation costs

100 million user service. Using a ***high-speed microservice architecture design***, we created a system that handled more load on 13 servers than another similar system at the same company that used 2,000 servers (and needed a costly team to monitor and tweak), and another company in the same industry used 150 servers to handle less traffic with a similar system. Our dev team was smaller. The project had a quicker turnaround time.

Reliability vs. operational predictability

Reliability

- Replication, fault tolerance, number of nodes
  - More nodes equates to more reliability but at a cost
  - How much? Are there diminishing returns?
- Reliability is also a function of simplicity and operational predictability

Cloud myth more nodes = more reliable

True but less nodes also equals more operational predictability and there is diminishing returns with more nodes = more reliability

3 servers with 99% reliability

- 1 in a million chance all go down (1 in 1,000,000)

5 servers with 99% reliability

- 1 in a 10 billion chance all go down

Amazon EC2 offers 99.95% SLA. Other clouds offer much higher SLA. Chances are near astronomical that all instances go down at the same time.

If you can handle all traffic on 1 or 2 nodes

If you can handle all traffic on a few nodes, then you need three to five nodes max for reliability.

If you app is mission critical, this might mean 3 to five nodes in two geographic regions or availability zones.

It never means you need 1,000.

Reactive, Reactor and Reactive streams

HSRM need reactive programming

Reactive frameworks

Reakt, Streams, Vert.x, Baratine, Akka, Netty, Node.js, Go Channels, Twisted, QBit Java Microservice Lib, Spring 5 reactive, Java 9 Flow, RxJava, Java EE (Java-RS, Servlet Async, WebSocket), JavaOne Keynote: JavaEE getting reactive, etc.

Tools that can really help

Kafka, Kinesis, JMS, Sockets, Streams

12 factor services, Monitoring and alerting, containers, Mesos/Kubernetes, CI, etc. but that is just normal microservice accompaniments.

Reactive programming

Reactive Manifesto - what is it and why it matters

Reactor Pattern

Microservices

- Async programming prefered for resiliency
- Avoiding cascading failures (“***synchronous calls considered harmful***”)

Async programming - key to all

Even ***method calls*** and results can be streamed

Streams vs Async Service Calls

Microservices / RESTful services / SOA services

REST / HTTP calls common denominator

Even messaging can be request/reply

Streams vs. Service Calls

- Level of abstraction differences,
- Calls can be streamed, Results can be streamed
- What level of abstraction fits the problem you are trying to solve
- Are streams a implementation details or a direct concept?

Rules of the Road for HSRM

Single writer rule - in-memory

- Data ownership (or lease)
  - Ownership update data
  - Services own data for a period of time (lease)
- Only one service can write data to a domain object (or set of domain objects)
  - Reduces Cache inconsistencies (root of evil and complexity)
  - Best way to keep data in sync is do not use cache for operational data
  - Caches are used but for caching data from other services (if needed)
- Uses a service store (which is a streaming store for service domain data)
- Data lease
  - User not edited after ½ hour evicted (configurable)
  - Every update renews lease
  - Each node has max number of users store LRU (for reshuffle)
  - Users are batch requests and then batch streamed into Service

Avoid the following for High-Speed

- Caching (use sparingly)
- Blocking
- Transactions
- Databases for operational data (in direct line of fire)
    - Operation data is user and data we are tracking for recommendations
    - Operation is data to run the service
    - Operation data is not data for reports, offline analytics

Embrace the following for High-Speed

- In-memory service data and data faulting
- Sharding
- Async callbacks
- Replication
- Batching / Streaming
- Remediations
- Messaging/Streaming data(Kafka, Kinesis, etc.)
- Service Stores for operational data
- Transaction alternatives
    - Async, Pre-ack, then execute (async)
    - Remediation queue
    - Bundle request in persistent queue and run transaction out of process

Service Sharding / Service Routing

- Elasticity is achieved through leasing and sharding
- A service server node owns service data for a period of time
- All calls for that user's data is made to that server
- In front of a series of service servers is a ***service router***
- A ***service router*** could use a sticky simple round robin, consistent hash affinity based on a header or a more complex approach with more knowledge about back end services
    - Important part is you can add more back end services if needed and rely on lease expiration or an event to notify the service which data set/shard it is handling
    - Sticky IP round robin for a simple case or userId in HTTP header (sticky)

Fault tolerance

- More important the data and the more replication and synchronization that needs to be done
- More important the data the more resources that are needed to ensure data safety
- If a service node goes down, a service router can select another service node to do that work from service discovery
- Service data can be data faulted loaded in an async/data faulting/batch to new service
- Updates to data can be sent in a persistent streaming/messaging (Kafka, Kinesis, JMS)

Data ownership

- More data you can have in-memory in the service the faster your services can run
- Data ownership means at a given point in time one instance of a service owns a particular domain model object (user, artist, partner, song) or data model object tree (user's preferences)
- Data faulting, and data leasing help services own data and use the single writer rule

Why lease?

- If data is small enough one service can own it all, but if that does not make sense
- Leasing data provides a level of elasticity
- Leasing allows you to spin up more nodes
- Data must be loaded from the service store quickly and usually in batches and streams
- The Faster you can move data into service, the shorter you can keep a lease
- Services can save data periodically to the service store or even stream updates if needed
- Expired leases means data has to be reloaded from service store
- Expired lease can be like a cache expiry (except lease relies on single writer rule)

Service Store

- Vend data quickly
- Vend data in streams
- Keep most data in-memory
- Accept data requests in streams
- Service Store is about storing and vending service data
- Updates can be in streams
- Only for Service operational data

Service Actors and streams of calls

Active Objects / Threading model

Active object pattern

- separates method execution from method invocation

Minimize complex synchronization code

- Each active object resides in their own thread of control
- Method calls are queued

Akka ***typed actor model*** (close) / QBit implements ***service actors***

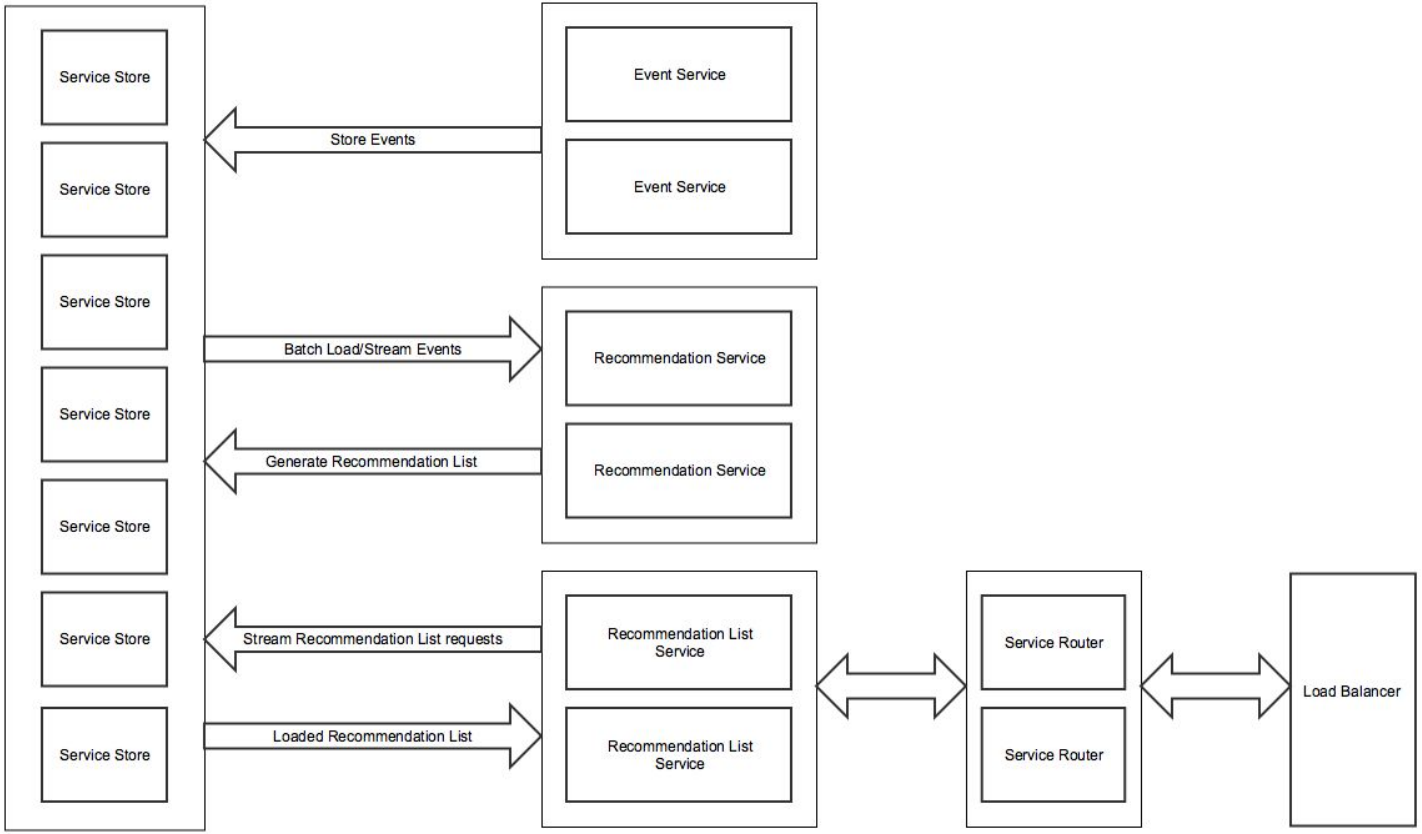
Queues/messaging are essential to handle back pressure and create reactive services

Active Objects consist of six element

- A client proxy to provide an interface for clients.
  - Client proxy can be local (local client proxy) or remote (remote client proxy)
- An interface which defines the method request on an active object
- A queue of pending method requests from clients
- A scheduler, which decides which request to execute next which could for example delay invocation until service data if faulted in or which could reorder method calls based on priority or which could work with several related services from one scheduler allowing said services to make local non-enqueued calls to each other
- Implementation of the active object methods. Contains your code.
- A service callback for the client to receive the result

What came from this experience

Proposed Version 2



What came out of it?

We both work with reactive programming on a daily basis

Went on to create other high-speed microservice

Implemented a ***OAuth*** rate limiter to limit service calls from partners that sits in front of thousands of backend services

QBit, a microservice service actor framework lib

Reakt, reactive Java promises and stream handling for async call coordination and stream handling.

Related projects

- QBit Java Microservice (built on top of Vert.x for IO)
  - Using Reakt reactor to manage callbacks,
  - REST and WebSocket services (WebSocket RPC) use Reakt Promises and Reakt Callbacks
- Lokate - service discovery lib for DNS-A, DNS-SRV, Consul, Mesos, Marathon
  - Uses Reakt invokeable promises (Vert.x for IO)
- Elekt - leadership lib that uses tools like Consul to do leadership election (uses promises)
- Reakt-Guava - Reakt Bridge to Guava listable futures
- Reakt-Vertx - Reakt Bridge for Vert.x AsyncCallbackHandler
- Reakt-DynamoDB - Reakt wrapper for async DynamoDB
- Reakt-Cassandra - Reakt wrapper for async Cassandra access
- Reakt-Kinesis - Reakt wrapper for async Kinesis (under development)

Related Talks

Reactive Java: Promises and Streams with Reakt

- Conference: JavaOne
- Session Type: Conference Session
- Session ID: CON5842
- Speakers: ***Geoff Chandler*** and ***Rick Hightower***
- Room: Hilton—Continental Ballroom 5
- Date and Time: 09/19/16, 04:00:00 PM - 05:00:00 PM

Reactive Java: Promises and Streams with Reakt in Practice

- Conference: JavaOne
- Session Type: HOL (Hands-on Lab) Session
- Session ID: HOL5852
- Speakers: ***Geoff Chandler***, ***Jason Daniel***, and ***Rick Hightower***
- Room: Hilton—Franciscan Room C/D
- Date and Time: 09/20/16, 04:00:00 PM - 06:00:00 PM

Conclusion

Streams and Batches to reduce thread hand-off and improve IO throughput

In-Memory, data-lease/ownership to increase throughput

- Single writer
- Lease with streams for failover and elasticity

Questions

?

Author Bio

Author Jason Daniel

Senior Director of Engineering NBC. Works with Spark, Kafka, Mesos, and reactive programming. Early contributor to QBit's rough starts. Expert in RxJava and Vert.x.

Author Bio Rick Hightower

Rick frequently writes about and develops high-speed microservices. He focuses on streaming, monitoring, alerting, and deploying microservices. He was the founding developer of QBit and Reakt as well as Boon.