bet365

Using Erlang, Riak and the ORSWOT CRDT at bet365 for Scalability and Performance

Michael Owen
Research and Development Engineer

Background

bet365 in stats

- **Founded in 2000**
- **Located in Stoke-on-Trent**
- **The largest online sports betting company**
- **Over 19 million customers**
- **One of the largest private companies in the UK**
- **Employs more than 2,000 people**
- **2013-2014: Over £26 billion was staked**
  - **Last year is likely to be around 25% up**
    - **Business growing very rapidly!**
- **Very technology focused company**

bet365 technology stats

- Over 500 employees within technology
- £60 million per year IT budget
- Fifteen datacentres in seven countries worldwide
- 100Gb capable private dark fibre network
- 9 upstream ISPs
- 150 Gigabits of aggregated edge bandwidth
  - 25 Gbps and 6M HTTP requests/sec at peak
- Around 1 to 1.5 million markets on site at any time
- 18 languages supported
- Push systems burst to 100,000 changes per second
  - Almost all this change generated via automated models
- Database systems running at > 500K TPS at peak
- Over 2.5 million concurrent users of our push systems
- We stream more live sport than anyone else in Europe

Production systems using Erlang and Riak

- **Cash-out**

  A system used by customers to close out bets early.

- **Stronger**

  An online transaction processing (OLTP) data layer.

Why Erlang and Riak?

Our historical technology stack

- Very pragmatic
- What would deliver a quality product to market in record time
- Mostly .NET with some Java middleware
- Lot and lots of SQL Server

But we needed to change

- Complexity of code and systems
- Needed to make better use of multi-core CPUs
- Needed to scale out
  - Could no longer scale our SQL infrastructure
    - Had scaled up and out as far as we could
  - Lack of scalability caused undue stress on the infrastructure
    - Lead to loss of availability

Erlang Adoption

Erlang – Key learnings

- You can get a lot done in a short space of time. A plus and a minus!
- Tooling is limited
- Hot code upgrades with state can be hard
- Dependency management could be better
- Get as much visibility into the system as possible e.g. stats, data etc
- Use OTP / reuse proven code
- Keep to standards (e.g. code layout etc)

Erlang – Key learnings

- Check your supervision tree
- Message passing is a double edged sword
- Keep state small (e.g. gen_server state)
- Binaries and GC: Heap vs reference-counted
- Explore whether you need the Transparent Huge Pages (THP) feature in the Linux kernel
- Don’t use error_logger as your main logger
- Validate all data coming into the Erlang system at the edge

Riak Adoption

Riak – Brief overview

- **Key value store**
- **Inspired by the Dynamo paper**
  - **http://www.read.seas.harvard.edu/~kohler/class/cs239-w08/decandia07dynamo.pdf**
- **Traditionally, an eventually consistent system (AP from CAP)**
  - **Riak 2.0+: Introduction of a strongly consistent option (CP from CAP)**
- **A Riak cluster is a 160-bit integer space – the Riak Ring**
- **Split into partitions – a virtual node (vnode) is responsible for each one**
- **A vnode lives on one of the Riak nodes**
- **Data is stored in a number of partitions (n_val setting – default 3)**
  - **Consistent hashing technique helps with identifying the partitions for putting and getting the data**

Riak – Why?

- Open source aspect
- Uses Erlang
- Based on solid ideas
- Horizontally scalable
- Highly available
- Masterless: No global locks – performance is predictable
- Designed to be operationally simple
- Support and community exists

Riak – Key learnings

- **Eventually consistent: Eventually the data will converge to the consistent value**
  - **Keep in mind:**
    - **A get/read may return an old value**
    - **A put/write may be accepted for a key at the same time as another concurrent put for the same key in the cluster (i.e. no global locking)**
  - **Bend your problem! E.g. Look at it from another side**
- **Data model for your use case i.e. normalisation isn't key: Trade off puts vs gets for your use case**
- **No bigger object sizes than 1MB**
- **Store data in a structure which helps with version upgrades**
- **Riak Enterprise**
  - **Multi-Datacenter Replication + Support**

Riak – Key learnings

- Consult Riak's System Performance Tuning documentation
- Different internode network vs inbound
- Monitor network and disk usage
- Use the "riak-admin diag" command
- Use Basho Bench to load test your cluster
- For bitcask backend: load test merging and tune for your use case
  - Setting log_needs_merge to true will help with this tuning
- Allow siblings (i.e. allow_mult = true)
  - With resolving siblings asap

Riak – What are siblings?

- A sibling happens when Riak does not know which value is the causally recent (E.g. because of concurrent puts/writes)
  - Uses version vectors to know this
    - Version vector A is Concurrent to version vector B (as opposed to Descends or Dominates)
    - Explained later
  - Referenced as vector clocks in the Riak documentation – should have been named version vectors
    - Talk: A Brief History of Time in Riak. Sean Cribbs (Basho). RICON 2014
    - https://haslab.wordpress.com/2011/07/08/version-vectors-are-not-vector-clocks/
    - Similar logic, however:
      - Vector clocks is about tracking events to a computation
      - Version vectors is about tracking updates to data replicas
  - Riak 2.0 introduced the option of dotted version vectors instead
    - Similar idea to the ORSWOT CRDT dot functionality (explained later)
    - Reduces potential number of siblings (i.e. causality tracking is more accurate) -> limits sibling explosion
- All sibling values are returned (i.e. more than one)
  - Big difference to the normal experience with SQL type data stores

Riak – allow_mult=false

- You can set allow_mult to false (i.e. no siblings to the client) with:
  - last_write_wins set to false
    - Uses version vectors. In conflict, the sibling with the highest timestamp wins.
  - last_write_wins set to true
    - Doesn't use version vectors – new value overwrites current value
- However, not recommended* because of potential data loss
  - Network problems
    - Reading: Fallacies of Distributed Computing Explained, Arnon Rotem-Gal-Oz
  - Complexity of time synchronisation across servers (speed of light, machines fail etc)
    - Reading: There is No Now - Problems with simultaneity in distributed systems, Justin Sheehy. 2015

** Perhaps for immutable data with separate unique keys*

Riak – Dealing with siblings

- Sibling values are returned on a get request
- Need to have a merge function to produce the correct value
- The merge function should be deterministic by having the following properties:
  - Associativity
    - Order of the merge function being applied to the data doesn't matter as long as the sequence of the data items is not changed
  - Commutativity
    - Order into the merge function does not matter
  - Idempotence
    - Merge function applied twice to the same value results in the same value
- Can be hard to get right
  - Can lead to possible data loss and incorrect data

CRDTs

CRDTs – What are they?

- Conflict-Free Replicated Data Types
- Can be:
  - Operation based: Commutative Replicated Data Types
  - State based: Convergent Replicated Data Types
- Reduces complexity by having no client side siblings
  - But still having no data loss
- Readings:
  - A comprehensive study of Convergent and Commutative Replicated Data Types. Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski. 2011.
  - Conflict-free replicated data types. Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski. 2011.
  - CRDTs: Consistency without concurrency control. Mihai Letia, Nuno Preguiça, Marc Shapiro. 2009.
  - http://christophermeiklejohn.com/crdt/2014/07/22/readings-in-crdts.html

CRDTs – Operation based

- **Commutative Replicated Data Types**
- **All replicas of the data are sent operational updates**
- **Relies more on a good network and reliably delivering updates**
- **Knowing the current true membership is more important**

CRDTs – State based

- Convergent Replicated Data Types
- Data is locally updated, sent to replicas and merged
- Update function must be monotonically increasing
- Generally easier to understand than operation based
- Easier to have an elastic membership of replicas
- However, more data is sent around the network

CRDTs – Types

- **Different data type implementations exist for:**
  - **Counters**
  - **Sets**
  - **Maps**
  - **…**
- **For our core use case the Sets data type made sense**
- **However, we were and are using Riak 1.4+ and a Sets CRDT isn't available**
  - **Introduced in Riak 2.0+**
  - **At the time, Riak 2.0+ wasn't even a release candidate**

CRDTs – riak_dt

- We decided to use the riak_dt dependency and integrate it ourselves into our system using Riak 1.4+
  - https://github.com/basho/riak_dt
    - Apache License Version 2.0 (http://www.apache.org/licenses/LICENSE-2.0)

- Different set based implementations exist (all state based CRDT's):
  - G-Set: Grow only set
    - i.e. no remove
  - OR-Set: Observe Remove Set
    - Able to add and remove
    - However, when an element is removed a tombstone still exists i.e. Size problem
  - ORSWOT: Observe Remove Set Without Tombstones
    - Able to add and remove, but doesn't have tombstones
    - In a concurrent add and remove of the same element –> add-wins
    - Reading: An optimized conflict-free replicated set. Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, Sérgio Duarte. 2012.

- For our core use case – we chose to use the ORSWOT

CRDTs – ORSWOT overview

- **Example:**

ORSWOT entry

ORSWOT element (i.e. data)

```
<<“KEY_NAME”>> = { [ {x,1}, {y,2} ] , [ {<<“Data1”>>, [ {x,1} ] }, {<<“Data2”>>, [ {y,1} ] }, {<<“Data3”>>, [ {y,2} ] } ] }
```

ORSWOT entries (a list for the purpose of example)

Value i.e. the ORSWOT

Version vector exists as part of the ORSWOT:

- List of tuples
- Each tuple being {UniqueActorName, Counter}
- ORSWOT operations happen through an unique actor
- Similar but a different instance of version vector to what will also exist for the overall key/value

Dots exist for each element in the ORSWOT set:

- Just a minimal version vector
- Each dot pair represents a tuple in the ORSWOT version vector at a particular point
- Usually a list of one
  - Can be a list of more than one when, for example, two ORSWOTs for two concurrent adds of the same element (i.e. using two different unique actors) are merged

CRDTs – ORSWOT overview

- **Adding an element:**
  - **Version vector as part of the ORSWOT incremented**
    - **Counter for unique actor being used is incremented (or set as 1 if not currently in the version vector)**
  - **The updated {UniqueActorName, Counter} pair is stored with the element as its dots**
    - **If an entry already exists in the ORSWOT for the element, this is replaced**

Example:

Adding <<“Data2”>> using unique actor y to the existing ORSWOT:

{ [{x,1}] , [{<<“Data1”>>, [{x,1}] }] }

Results in the new ORSWOT:

{ [{x,1}, {y,1}] , [{<<“Data1”>>, [{x,1}] }, {<<“Data2”>>, [{y,1}] }] }

and ORSWOT value (i.e. ignoring metadata / what a client would be interested in) of:

[<<"Data1">>, <<"Data2">>]

CRDTs – ORSWOT overview

- **Removing an element:**
  - **Version vector as part of the ORSWOT does not change**
    - **Of course when the put happens to store the ORSWOT's updated value, the version vector for the overall key/value is incremented**
  - **The elements entry is simply removed from the ORSWOT**
    - **i.e. No tombstones**

Example:

Removing <<"Data1">> from the existing ORSWOT:

{ [{x,1}, {y,1}] , [{<<"Data1">>, [{x,1}] }, {<<"Data2">>, [{y,1}] }] }

Results in the new ORSWOT:

{ [{x,1}, {y,1}] , [{<<"Data2">>, [{y,1}] }] }

**** Further options do exist, such as being able to delay removes***

- ***E.g. the ORSWOT object doing a remove might have not seen the original add for the element yet (e.g. because of being a replica not merged with the add yet)***
- ***Would also add further logic to the merge operation***

CRDTs – ORSWOT overview

- **Merging ORSWOT's: ORSWOT A and ORSWOT B**
  - **E.g. because of siblings detected by version vectors for the overall key/value**
  - **Version vectors for ORSWOT A and ORSWOT B merged**
    - **i.e. the least possible common descendant of both (*)**
  - **Elements merged:**
    - **Common elements only kept if there exists a non-empty dots for them from merging (*):**
      - **Common dot pairs for the element in ORSWOT A and ORSWOT B**
      - **Dot pairs for the element only in ORSWOT A where the dot pair count is greater than any count* for the same actor in ORSWOT B's version vector**
      - **Dot pairs for the element only in ORSWOT B where the dot pair count is greater than any count* for the same actor in ORSWOT A's version vector**
    - **Elements only in ORSWOT A only kept if there exists a non-empty dots for them after:**
      - **Keeping only dot pairs for the element where the dot pair count is greater than any count* for the same actor in ORSWOT B's version vector**
    - **Elements only in ORSWOT B only kept if there exists a non-empty dots for them after:**
      - **Keeping only dot pairs for the element where the dot pair count is greater than any count* for the same actor in ORSWOT A's version vector**

***** As you might think, if there doesn't exist an appropriate actor pair/count in the version vector then the dot pair is merged/kept***

CRDTs – ORSWOT overview

- **Merging example:**

ORSWOT A: { [{x,1}, {y,2}] , [{<<“Data1”>>, [{x,1}] }, {<<“Data2”>>, [{y,1}] }, {<<“Data3”>>, [{y,2}] }] }

Seen:

1. Adding element <<“Data1”>> via actor x
2. Adding element <<“Data2”>> via actor y
3. Adding element <<“Data3”>> via actor y

ORSWOT B: { [{x,1}, {y,1}, {z,2}] , [{<<“Data2”>>, [{y,1}] }, {<<“Data3”>>, [{z,1}] }, {<<“Data4”>>, [{z,2}] }] }

Seen:

1. Adding element <<“Data1”>> via actor x
2. Adding element <<“Data2”>> via actor y
3. Adding element <<“Data3”>> via actor z
4. Adding element <<“Data4”>> via actor z
5. Removing element <<“Data1”>> via actor z

Merged ORSWOT: { [{x,1}, {y,2}, {z,2}] , [{<<“Data2”>>, [{y,1}] }, {<<“Data3”>>, [{y,2}, {z,1}] }, {<<“Data4”>>, [{z,2}] }] }

CRDTs – ORSWOT overview

- **Merging example:**

ORSWOT A: { [{x,1}, {y,2}] , [{<<“Data1”>>, [{x,1}] }, {<<“Data2”>>, [{y,1}] }, {<<“Data3”>>, [{y,2}] }] }

ORSWOT B: { [{x,1}, {y,1}, {z,2}] , [{<<“Data2”>>, [{y,1}] }, {<<“Data3”>>, [{z,1}] }, {<<“Data4”>>, [{z,2}] }] }

Merged = { [{x,1}, {y,2}, {z,2}] , [{<<“Data2”>>, [{y,1}] }, {<<“Data3”>>, [{y,2}, {z,1}] }, {<<“Data4”>>, [{z,2}] }] }

- [{x,1}, {y,2}, {z,2}] is the least possible common descendant for the version vectors of ORSWOT A and ORSWOT B
- Common elements:
  - <<“Data2”>> : ORSWOT’s have common dot pair {y,1} – no other dot pairs to consider/merge with it => element in
  - <<“Data3”>> : ORSWOT A has {y,2} which has a greater count than {y,1} in ORSWOT B’s version vector [{x,1}, {y,1}, {z,2}]
    ORSWOT B has {z,1} which is included because ORSWOT A’s version vector [{x,1}, {y,2}] doesn’t have a count for z.
    Therefore {y,2} and {z,1} are merged to give [{y,2}, {z,1}] => element in
- Elements only in ORSWOT A:
  - <<“Data1”>> : From ORSWOT A, no dots exist from [{x,1}] which are greater than ORSWOT B’s version vector [{x,1}, {y,1}, {z,2}] => element not in
- Elements only in ORSWOT B:
  - <<“Data4”>> : From ORSWOT B, dot pair {z,2} for the element is kept because ORSWOT A’s version vector [{x,1}, {y,2}] doesn’t have a count for z => element in

CRDTs – riak_dt integration

- Erlang middle layer between clients and Riak
- Middle layer needed to have unique actors (gen_server's) as part of using the ORSWOT implementation
  - With getting a balance on the number of them e.g. due to impacting ORSWOT version vector size
- Unique actor names pre-defined as part of server setup configuration
  - With making them small e.g. due to impacting ORSWOT version vector size
- Clients don't have to deal with siblings
- Middle layer (i.e. client to Riak) does need to bring back siblings
  - Not as good as a Riak server side CRDT (i.e. Riak 2.0+)
- Still using version vectors on the overall key/value's
  - i.e. to know to do the ORSWOT merge operation

Getting It Live

Tooling

- From day 1 we ate our own dog food
- Monitoring
- Performance counters
- Error reporting (including correlation between systems)
- Built custom adhoc query tool
  - On replicated Riak cluster
- Custom reconciliation between new system and old system
- Created build and release scripts for automation

Released in phases

- Able to build on stable ground
- Able to get data to impact future decisions
- Built confidence
- Move functionality to using Erlang and Riak
- Not all phases were immediately business impacting
- However, overall able to get business impacting functionality out sooner

Replay testing

- Captured logs asynchronously in one phase
- Common interface between old and new systems
- Able to do reconciliation between the systems
- Big range of test data / realistic load profile
- Used for functional and performance testing
- Different logs for long weekend run vs quick run
- Complemented other testing such as specific unit/integration testing, fuzz testing and formal UAT

Performance testing

- Done early and repeated
- Built custom client using replay logs
- Able to increase load profile easily
- Used our custom tooling / monitoring
- Identified bottlenecks / tested horizontal scalability of system
- Adhoc changes and fed back into development
- Had a specific profile which could give a fair test between changes
- Basho Bench used to sanity test Riak cluster setup

Failure testing

- Did failure testing
- Built confidence and understanding
- Trying to make sure a failure seen in production isn’t the first time it’s experienced
- Tested common procedures e.g. taking nodes in and out of service
- Failures will happen – embrace them

Today with Stronger

- **The project was a success!**
- **We have a system which is:**
  - **Performant and able to deal with many times our peak load**
  - **Reliable and deals with failure using minimal human intervention**
- **We have introduced the business to a number of new technologies**
- **We have grown the capabilities of the business and our people**

Questions?