Name of system Chapter 15: Riak and Erlang/OTP
Reviewer: Daniel Kozimor
Date: November 1st, 2011.
Author of software: Basho Technologies Inc. Andy Gross is the Principle Architect.
Author of book chapter: Francesco Cesarini, Andy Gross, Justin Sheehy
Five star rating of book chapter: 2

Purpose of system:
Riak is a decentralized, distributed, NoSQL-based database system. It is a direct implementation of ideas detailed in a a white paper on Amazon's Dynamo - a distributed hash-table data store which powers much of Amazon's data-store. It is built up on the foundations and principles of Erlang and Open Telecom Platform (OTP) libraries. In fact, Riak could be thought of a set of Erlang applications working in conjunction. The key data structures which comprise Riak are implemented on top of provided OTP "Behaviours", such as Final State Machines (FSMs), Event Handlers and Worker-Supervisors Trees.

Riak is designed to be fully distributed with no central server and consequently no single point of failure. Data input is handled, along with replication, and conflict resolution, via gossip protocols. This lack of centralized server allows Riak to continue working under extreme duress, as the failure of individual nodes does not bring down the cluster. This also makes the database able to scale horizontally. Turning "on" more nodes on different machines, has the effect of allowing Riak to handle higher load. In addition, Riak can be integrated to applications via a conveniant REST api and contains a map/reduce engine to query and work over stored data.

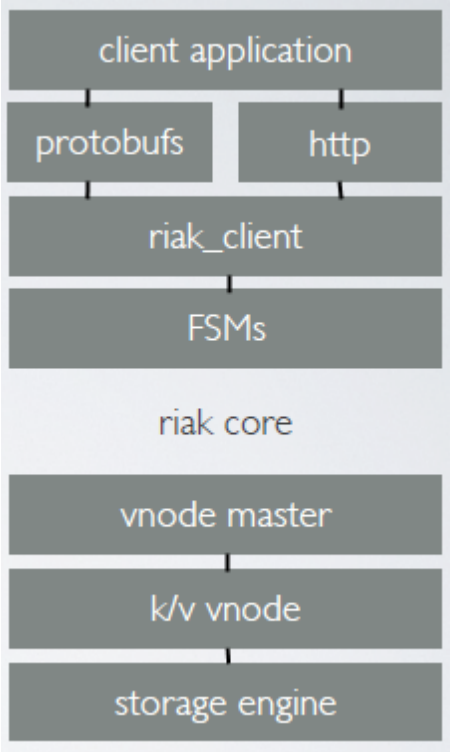
Basic metrics

KLOC: 50k (12k riak core, 17k riak kv, 10k webmachine).
Project start-up: 2008
Number of major releases: 10 point releases culminating in 1.0 release in Fall 2011.
Number of developers: Around 10 full-time developers.
Size of user community or number of installations: Unknown.
Major stakeholders: Mozilla, AOL and many other companies
Use of concurrency:
Because of it's distributed nature, Riak makes much use of concurrency. At the basic level, Erlang processes are used everywhere. They are light-weight software threads, also known as green threads, which run on local or remote machines (on other Erlang VMs in the latter case). Asynchronous message are passed between these processes to coordinate work. Gossip protocol (which are used to coordinate Riak nodes), and OTP distributed design patterns are built on top of them.
Implementation language: Erlang
Supporting software: Erlang and OTP libraries.

High level architecture

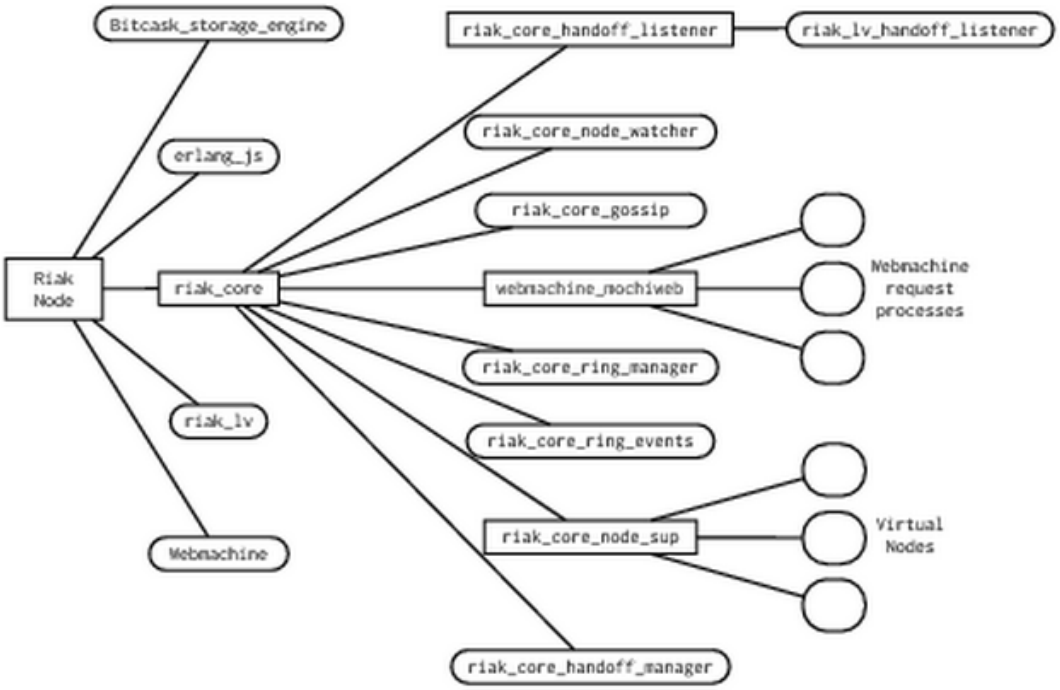
Diagram of software architecture

The following is a layered diagram for one Riak node. Under Riak's distributed architecture, each node is represented by this diagram. Nodes communicate with each other with a gossip protocol removing the need for centralized management.



- client application: Clients, written in many languages and frameworks which interact with the Riak database.
- http (Webmachine): A restful http api which provides an integration point for clients.
- protobufs: A binary RPC-type api which lets clients interact with the database.
- riak_client: native erlang interface that is wrapped by protobuys and http.
- FSMs: Requests are modeled as Final State Machines. "Quorum behaviour" and conflict resolution is handled here. Each FSM runs in it's own Erlang process.
- riak core: "generic" distributed system code that Riak nodes are built upon.
- vnode master: coordinates incoming messages to all local vnodes. One vnode master per node.
- k/v node: individual key-value vnodes (virtual nodes) are disposable, light-weight, abstractions of physical key-value storage.
- storage engine: disk storage engine.

The following diagram shows Riak's behaviour under Erlang's Worker-Supervisor design pattern. Again, this models a single node. Boxes represent supervisor, while ellipses represent workers. Supervisors monitor the state of their children (workers and supervisors) and in case of failure can restart them, or stop themselves and pass the baton to their parent (depending on the setting). This kind of restart cycle can propagate up to the roots of the tree. The nice side-effect is that a failure in one component is isolated from the rest of the system and can be restarted with little affect on it's neighbours.



High level scenarios

Data Replication

Data Replication is a core feature of Riak data store. At a high-level, any data written, will be replicated to adjacent nodes. The number of replications depends on the n_val property that is set on a particular bucket. Choosing that value involves a trade-off between performance (lower n_val) and data availability in face of failures (higher n_val). The default is 3.

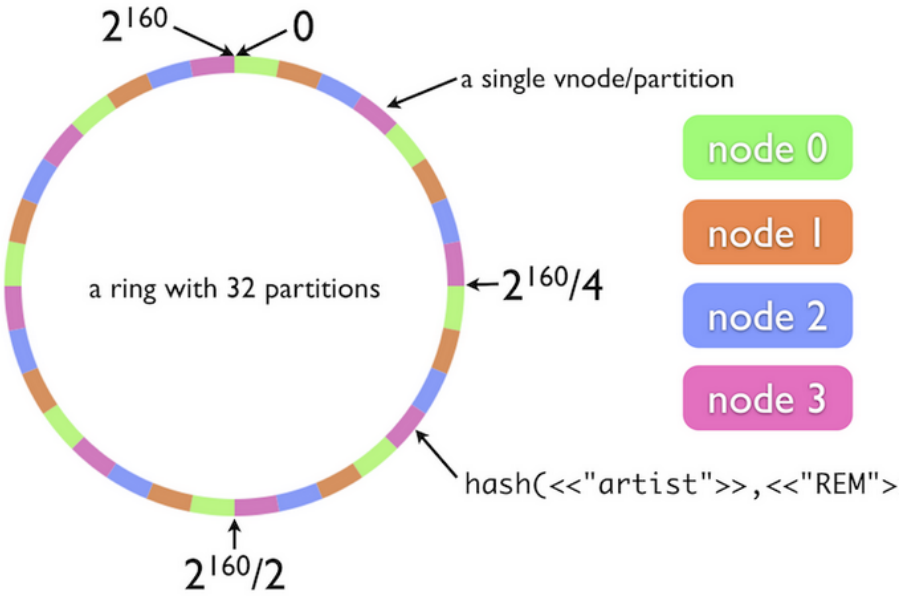
A typical scenario would go as follows:

- Client issues a PUT request via http api against one node of the cluster for with bucker and key id. The request is handled by WebMachine http api, which calls the erlang-bridge in riak_client.
- The request will spawn an Erlang process which runs the corresponding FSM

Behaviour. In this case it is riak_kv_put_fsm, which is built upon riak_core. This FSM process will determine all relevant actions needed to store data in the cluster.

- The bucket/key combination is hashed to a 160 bit integer.
- riak_core_manager uses the hash to look up a set of “preferred” partitions and chooses the first n_val partitions. The rest of the list is used as fallback in case of failure with preferred partitions. There is no guarantee that “preferred” partitions are actually on separate physical nodes, as multiple nodes can be one physical machine and the hash function merely maps, equally, the 160bit integer to the number of nodes in a Riak cluster (See diagram below).
- riak_core_manager finds out the locations within a cluster of parent nodes preferred partitions.
- Each parent node of the preferred partition is notified with the object. vnode_master does the resolution.
- If request to any parent node fails, then a fallback node is used.

The following diagram shows a partition space (hashed by a 160 bit integer), as described in scenario, and the way it may be divided amongst four nodes (each color on the ring corresponds to partitions space which belongs to that colored node). Each block would be managed by one vnode on a node, with a node having many vnodes/partition blocks.



Data structures or algorithms

Key data structures that Riak is built upon all come from Erlang OTP. These include:

- Finite State Machines
  Responsible for handling requests, managing client process communications and associated states.
- Event Handlers
  Distributed handlers which handle internal and external node events.
- Worker-Supervisor Trees
  Application business logic is divided in worker-supervisor trees in which supervisor processes spin and monitor children processes and handle failures. Worker processes are created by supervisors and usually perform some kind of work.

Architectural style: Because it's built with Erlang, a functional language with intrisic support for light-weight threads and message passing, every facet of Riak's Architecture is parallelized across many threads, with loosely coupled components. Architectural choices are also heavily influenced by OTP Behaviours.

Major evolutionary changes

As Riak is a fairly new project so the initial architectural choices, such as using Erlang OTP Behaviours, have carried over. The layered architecture of a Riak node is loosely coupled, which allows for swapping in new components for old ones. For example a new Map-Reduce engine, Riak Pipe, was created and included as an option in the core distribution. In addition, over the

course of the lifetime, support for at least three different kinds of storage engines was added.

Performance bottlenecks

Latency is an issue in a distributed system such as Riak. Queries that can only be fulfilled with nodes on different physical machines, will be constrained by network speeds.

Real time

Riak strives for real-time performance, during query, read, map-reduce and delete operations.

Notation for architecture

Riak is very well documented on Basho corporate and community sites. In most situations, Riak nodes are represented as layered brick diagrams, as the components with are loosly coupled and interchangeable. The worker-supervisor OTP pattern is represented as a tree with nodes as ellipses (workers) and boxes (supervisors). When talking about mapping of hash-space to nodes, a ring metaphor (along with a corresponding diagram) is used.

Methodology: Not clear, though given official releases are open source, and nightly builds are available, it seems to imply agile, iterative development.

Appendix:

Kruchten's eight context attributes applied to Brown/Wilson systems

1. **Size:** Med
2. **Criticality:** Med - company critical as it is a data storage engine.
3. **Age of system:** Med (2008)
4. **Rate of change:** Med - constantly developed. Though uses the same framework (Erlang OTP) still refactored.
5. **Business model:** Open Source with Commercial add-ons
6. **Stable architecture:** Lo. Though constantly developed, the architecture since beginning has been constant.
7. **Team distribution:** Med. Much of the team is located in-house, but there are a number of committers from different countries.
8. **Governance:** Hi. Basho Technologies Inc. funds much of development of Riak with contributions from community. However Basho developers control pull request.