Riak PG: Distributed Process Groups on Dynamo-style Distributed Storage

Christopher Meiklejohn
Basho Technologies, Inc.
cmeiklejohn@basho.com

Abstract

We present Riak PG, a new Erlang process group registry for highly-available applications. The Riak PG system is a Dynamo-based, distributed, fault-tolerant, named process group registry for use as an alternative to the built-in Erlang process group facility, pg2, and the globally distributed extended process registry, gproc.

Riak PG aims to provide a highly-available, fault-tolerant, distributed registry by sacrificing strong consistency for eventual consistency in applications where availability of the registry is paramount to application function and performance.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Algorithms, Design, Reliability

Keywords Dynamo; Erlang; Riak; process registry

1. Introduction

One common design goal found in applications consisting of many threads of execution is the ability to logically group computational units together. In Erlang, this is especially desirable and important because processes are extremely lightweight to spawn and are used to isolate computation. For example, Wiger describes in the original paper on the extended process registry [18] that at any point in time there may be 200,000 to 400,000 processes running in a given virtual machine.

Erlang provides two primary facilities for grouping processes into logical groups: the built-in named process group registry bundled with OTP, pg2 [9], and the extended Erlang process registry, gproc [17]. However, each of these existing solutions is less than optimal when designing a high-availability fault-tolerant distributed process group registry.

pg2, as documented [12], holds up reasonably well during node failures and network partitions. Each node is responsible for tracking state through an ets [7] table, and monitors are used to add and remove entries from that ets table as nodes are connected and disconnected. This works fairly well, as each node knows only about local processes, and can remove them from process groups when they die. However, it is possible for deleted groups to re-appear after a partition has healed.

In addition, pg2 attempts to coordinate a write against all connected nodes synchronously when performing any of the non-idempotent process group commands such as creating or joining a group. When network partitions occur or nodes fail, pg2 will block for the net tick time [8] interval and disconnect the nodes before returning the result of the write.

gproc, as documented [17] [18], with global distribution provided by gen_leader [15], is unreliable when experiencing node failures or network partitions. In addition to deadlocks during election [1], causing data unavailability, gproc has no mechanisms for properly resolving conflicting values when partitions heal, making data loss a possibility. Dynamic cluster membership is also problematic, as all candidate nodes need to be fully connected prior to initializing the gproc application.

Both of these existing systems attempt to provide guarantees of strong consistency, specifically stating that all updates will be reflected across all nodes in the same order. In the event of partitions, they attempt to maintain a strongly consistent view of data between connected nodes, by removing or filtering out processes by node. When partitions heal, they both attempt to merge this state back together.

Based on the observed behavior of both pg2 and gproc, we posit that there are three main challenges in providing a distributed named process group registry, specifically:

- Dynamic addition and removal of nodes in a running cluster without interrupting service.
- Coordination of state mutation across nodes in the cluster.
- Resolution of conflicting values when recovering from failures, such as network partitions and unreachable nodes.

This paper describes the Riak PG system, which provides a highly-available, fault-tolerant, distributed, named process group registry. The Riak PG system provides the following properties:

- Dynamic cluster membership through the use of virtual nodes and a consistent hashing algorithm.
- Replicated state, with quorum-based reads and writes.
- Guaranteed conflict-free resolution of data replicas through the use of convergent data structures.

In providing the above properties, Riak PG sacrifices strong consistency for eventual consistency in its modeling of process groups. Reads and writes are performed to a quorum of nodes to provide high-availability and fault-tolerance. However, in the event of partitions the system may not always return a complete listing of process group memberships. To minimize the adverse effect caused by the partitions, we can be guaranteed that the system

Erlang '13, September 28, 2013, Boston, MA, USA.

ACM 978-1-4503-2385-7/13/09...$15.00.
http://dx.doi.org/10.1145/2505305.2505309

will eventually converge to the correct value once the partition has healed, by leveraging data types that are designed to merge without conflicts.

Tradeoffs such as these can be found in Brewer and Fox's original paper [10] on the concepts of harvest and yield, where harvest represents the completeness of data returned in a response and yield represents the probability that a response successfully returns. Riak PG specifically provides a process group registry where availability is paramount, leaning towards higher yield, and applications are designed to handle an eventually consistent view of process groups.

2. Requirements

We identified the following requirements for the design of a distributed named process group registry.

2.1 Structured names

Given that the most common use case of a dynamic named process group registry is to group processes together to support the service location lookup pattern, it is likely that as the number of processes grows, the number of groups will grow.

Generating a unique atom for each group could lead to an overload of the atom table, which would hinder scalability of the system. We therefore want to provide a system where we can use structured Erlang terms as group identifiers.

2.2 Multiple non-unique names per process

We also assume that when using a named process group registry for the service location lookup pattern, groups will contain more than one member, and a process will be able to obtain memberships in multiple groups at the same time.

2.3 Dynamic cluster membership

In order to facilitate scalability of our system, specifically related to a growing number of processes and process groups, we need a solution that supports dynamic cluster membership of nodes. The system should be resilient to nodes being added and removed without the loss of state, specifically the process group listing.

2.4 Partition tolerance and conflict resolution

Given that nodes fail and network outages occur, the system needs to be able to withstand partitions and unreachable nodes. Achieving this goal while guaranteeing strong consistency is not possible when attempting to provide a high-availability system, so we require a consistency model similar to an eventual consistency data store.

To ensure that we can automatically resolve data discrepancies when partitions heal, we require that the data be stored in a data structure that monotonically advances and provides merge operations that are associative, commutative, and idempotent.

3. Uses

As Riak PG aims to provide a highly-available registry by trading a strongly consistent view of process groups for an eventually consistent view, we will use the following section to outline some example applications that benefit from this type of process registry.

Two types of applications that benefit from a globally distributed named process group registry are the publish and subscribe pattern and the service lookup pattern. Each of these applications has different semantics, but benefits from the same properties provided by the Riak PG system.

With the publish and subscribe pattern, process groups represent a logical grouping of processes that should receive messages when particular events occur. With the service lookup pattern, process groups represent a logical grouping of processes responsible for the handling of a particular task. The main difference between these two uses is that in the former, application code most likely will require the entire process listing, whereas in the latter, application code will most likely request one process out of the process listing. Riak PG provides several benefits here if your application is built to handle eventual consistency of the process group.

Group memberships are written to a quorum of virtual nodes. This prevents writes from blocking until partitioned or unavailable nodes are disconnected and all target nodes provide an acknowledgement. Writes to each of these replicas, regardless of ordering, are then guaranteed to converge using our conflict-free resolvable data type. Updates that occur during a partition, are also guaranteed to properly converge to the correct value. Riak PG also provides calls for filtering out the process listing to just local processes or processes running on connected nodes.

Because we no longer require synchronous writes to all nodes in the cluster, we are able to observe changes to the process group earlier, acknowledging that until the write has been propagated to all replicas not all consumers of the process listing may be viewing a total order of addition and deletion events to a particular process group. For some classes of applications, this may be perfectly acceptable: for example, situations where observing some partial ordering of events earlier is preferred over delayed, or failed, membership to an event stream that is total.

4. Background

The Riak PG system is built on top of Riak Core [2], a framework providing an implementation of many of the core concepts from the Dynamo paper [6]. The following sections and subsections review some of the concepts from Dynamo necessary to understand Riak PG's implementation.

4.1 Dynamo

Consistent hashing, hash-space partitioning and a configurable data replication factor are the concepts critical for understanding Riak PG's implementation on top of Dynamo.

4.1.1 Consistent Hashing

The Amazon Dynamo paper describes a key-value based storage system made up of a cluster of nodes, where every node in the cluster stores some subset of the total data. To distribute this data, a consistent hashing algorithm applied to the data's key is then used to determine a token in the hash-space for where this data should be distributed.

4.1.2 Hash-Space Partitioning

The entirety of the hash space is then is evenly divided between the nodes. Each even portion of the hash space is called a partition, and each partition is managed by a virtual node. Each physical node in the cluster hosts a number of virtual nodes, one for each partition assigned to that physical node. The hash resulting from running a key through the consistent hashing algorithm determines which partition is responsible for storing the data associated with that key.

4.1.3 Replication Factor

Dynamo replicates data on consecutive partitions. The replication factor N determines the number of replicas. When a key is mapped to a particular partition in the hash-space, the $(N-1)$ consecutive partitions are used to store replicas of the data. This collection of partitions is called the preference list or primaries.

4.1.4 Dynamic Cluster Membership

As the cluster grows and shrinks, partitions are redistributed to nodes, minimizing the amount of partitions that have to move between nodes to cut down on data transfer between nodes. This is a property of the consistent hashing algorithm described in section 4.1.1.

5. Implementation

Riak PG is structured similarly to all Riak Core applications: a set of coordinating virtual nodes, each modeled as an independent process implementing the OTP gen_server behavior and each claiming responsibility for one partition in hash-space. Riak PG uses a set of virtual nodes for managing a series of local Erlang dictionaries that contain replicated information about the current state of process groups in the cluster. The following sections detail the design, structure, and implementation of the Riak PG application.

5.1 Interface

Riak PG provides a similar API to the built-in distributed process group registry, pg2, as provided by the Erlang/OTP system. Functions are provided in the riak_pg module for creating, deleting, joining, leaving and returning the members, as well as local or connected members, of named process groups.

5.2 Structure

The memberships virtual node is primarily responsible for storing process group memberships, each of which maps a process group name to a vector of process ids in that group. The process group registry is stored as an Erlang dict, with the name of the process group acting as the key for the dict, and the value as a riak_dt_vvorset [3], an optimized Erlang implementation of an observed-removed set, as originally described by Shapiro et al. in [14]. We will discuss detailed use of the observed-removed set in Section 5.4.

5.3 Write coordination

Riak PG is implemented on top of Riak Core in a similar manner to the Riak KV [4] data store, an open source implementation of the Dynamo data store [6].

The Dynamo data storage system provides a simple distributed key/value storage engine. Each key is hashed to a particular location in the hash-space as discussed in Section 4. Starting at this location, the hash-space is walked $(N-1)$ partitions to determine the preference list for the provided key. The partitions that make up the preference list are responsible for storage and retrieval of that key's values. A finite state machine is then used to coordinate a synchronous write to a quorum of virtual nodes, while asynchronously writing to all partitions in the preference list, or fallback virtual nodes in the event of network partitions.

Within the implementation of process groups, each group is identified by an Erlang term, the structured names discussed in Section 4. These group names are then hashed via our consistent hashing algorithm to determine the preference list for the key in the cluster. These virtual nodes represent the nodes responsible for storing information regarding that process group.

Finite state machines are used to coordinate requests to all of the virtual nodes in the preference list. Each write, or in our system a process joining or leaving a group, is synchronously written to a quorum of virtual nodes, which acknowledge the write by modifying the dictionary for the given process group with an updated value of the observed-removed set, as seen in Figure 1.

```
1  %% @doc Respond to a join request.
2  handle_command({join, {ReqId, _}, Group, Pid},
3                 _Sender,
4                 #state{groups=Groups0,
5                        partition=Partition}=State) ->
6
7      %% Find existing list of Pids, and add.
8      Pids0 = pids(Groups0, Group,
9          riak_dt_vvorset:new()),
10
11     Pids = riak_dt_vvorset:update(
12         {add, Pid}, Partition, Pids0),
13
14     %% Store back into the dict.
15     Groups = dict:store(Group, Pids, Groups0),
16
17     %% Return updated groups.
18     {reply, {ok, ReqId}, State#state{groups=Groups}};
```

Figure 1. Virtual node handler for join requests

5.4 Observed-removed sets

Observed-removed sets provide a commutative set structure that model each change by monotonically advancing the data structure. At its core, it's composed of two sets: one that tracks addition of values to the set and one that tracks the removal of values from the set. This is referred to as a 2P set, which allows for the addition and removal of a particular value from the set once. To support multiple additions and removals of the same value multiple times, the set is simply extended by tagging each value with a unique identifier representing the insert.

The observed-removed set is one data type in a series of data types that are referred to as convergent replicated data types, or CvRDT, taken from the more generalized set of CRDTs, or convergent and commutative replicated data types.[14]. The CvRDT type has a unique property that guarantees, through a state-based merge operation which is associative, commutative, and idempotent, that no matter the order of operations the derived value of the data structure will always converge to the correct value, making them well suited for use in eventually consistent systems.

5.5 Read coordination

When attempting to retrieve the list of processes associated with a particular group, we treat it similarly to a read request in the Dynamo storage system. We perform a read against a quorum of nodes, but before returning the value to the user, we apply the merge operation supplied by the observed-removed set implementation against the values retrieved from the quorum of replicas. This ensures that writes visible on one particular replica, but not seen on another due perhaps to node failure or network partition, converge and we return the most up-to-date value seen between the two replicas.

Once the list of processes are returned to the user, the finite state machine coordinating the read request continues to execute until all of the values have been returned from all remaining replicas. At this point, the merge operation is applied to the remaining values, and we write back the updated values to the data store, but only if they are divergent from the stored value. This process, called read repair, ensures that stale or missing data on replicas are updated during the read process. This is shown in Figure 2.

Riak PG also provides a mechanism for returning only those processes local to the node requesting the process list. This is achieved by filtering the process group based on the node identifier of the requesting process.

```
%% @doc Perform read repair.
finalize(timeout, #state{replies=Replies}=State) ->
    ok = repair(Replies, State#state{pids=merge(Replies)}),
    {stop, normal, State}.

%% @doc Perform merge of replicas.
merge(Replies) ->
    lists:foldl(fun({_, Pids}, Acc) ->
                riak_dt_vvorset:merge(Pids, Acc) end,
            riak_dt_vvorset:new(), Replies).

%% @doc Trigger repair if necessary.
repair([{IndexNode, Pids}|Replies],
        #state{group=Group, pids=MPids}=State) ->
    case riak_dt_vvorset:equal(Pids, MPids) of
        false ->
            riak_pg_memberships_vnode:repair(
                IndexNode, Group, MPids);
        true ->
            ok
    end,
    repair(Replies, State);
repair([], _State) -> ok.
```

Figure 2. Read repair mechanism

5.6 Ownership handoff

Ownership handoff is the process by which partitions are redistributed to joining or leaving nodes as described in Section 4.1.4. It is handled similarly to a read request. As a node leaves the cluster and begins handing off partitions to other nodes, we ship the riak_dt_vvorset structures from the in memory dictionary over the wire. As entries arrive at the node that has assumed responsibility for a key that's being moved, we again perform the merge operation to ensure that the sets converge to the correct value.

5.7 Process monitors

Removing processes from the process group as they terminate is important but challenging. pg2, for example, has each node track group memberships of local processes. Monitors on each node are then used to observe termination of processes and remove the process from groups in which it has memberships.

However, when multiple nodes take responsibility for storage of the process group, we can no longer have each node only responsible for monitoring local processes; there is no way to track a process exit when the group is stored locally and the node running the process is on the opposite side of a network partition.

The strategy we employ is shown in Figure 3. We extend the original read repair process as seen in Figure 2 to prune processes that we know have been terminated based on the currently connected node list, by asking the node if the process is still alive.

Under the eventual consistency model, however, there is still the possibility that the process group may not immediately reflect a removal of a process from the group. Given this behavior, we defer handling of this to the application using Riak PG. When applications retrieve this process listing, it is up to the calling application to determine how to handle processes that are either terminated or unreachable. This is not different from pg2, as a race condition exists between the call to retrieve the current process listing and sending messages to those processes.

6. Evaluation

To evaluate Riak PG, we performed a series of tests based on objectives outlined in Section 1, specifically that the addition and removal of nodes dynamically is supported, and that the process

```
%% @doc Perform read repair.
finalize(timeout, #state{replies=Replies}=State) ->
    Merged = merge(Replies),
    Pruned = prune(Merged),
    ok = repair(Replies, State#state{pids=Pruned}),
    {stop, normal, State}.

%% @doc Based on connected nodes, prune out processes
%% that no longer exist.
prune(Set) ->
    Pids0 = riak_dt_vvorset:value(Set),
    lists:foldl(fun(Pid, Pids) ->
                case prune_pid(Pid) of
                    true ->
                        riak_dt_vvorset:update(
                          {remove, none, Pid},
                          Pids);
                    false ->
                        Pids
                end
        end, Set, Pids0).

%% @doc If the node is connected, and the process
%% is not alive, prune it.
prune_pid(Pid) when is_pid(Pid) ->
    lists:member(node(Pid), nodes()) andalso
      (is_process_alive(Pid) =:= false).
repair([], _State) -> ok.
```

Figure 3. Updated read repair with dead process pruning

registry converges to the correct values once network partitions heal.

6.1 Addition and removal of nodes

We have been able to successfully demonstrate that addition and removal of nodes from the cluster causes state to propagate successfully with the read repair process.

To test this, we configured two Erlang nodes running the Riak PG application and created a process group containing one process. When joining a third node to the cluster, and performing a read of memberships for that group, the group memberships are written back to the third node.

There are two scenarios that can occur when performing this read successfully from a quorum of nodes:

- Read from two replicas that contain the process group membership of one process.
- Read from one replica that contains no data, and one that contains the process group membership of one process.

In both scenarios, the finite state machine coordinating the request merges the values it reads and returns the merged value to the user. Since these merge operations commute, a valid process list is retrieved regardless if one of the process listings is empty. However, depending on the state of network partitions, this process list, while valid, may not be total; it might not include processes added to the group during a network partition.

6.2 Network partitions

Similar to the addition and removal of nodes, we have demonstrated that the operations of joins and leaves for processes to a process group all converge correctly when network partitions heal. Again, it is important to realize that until all partitions have healed and all data has been repaired through the read repair process, the process listing for all groups are not guaranteed to be total. We discuss

solutions for placing an upper bound on how long data may be stale after a partition heals in Section 8.2.

7. Related Work

The following section discusses related work in the field, specifically Howl [11] and CloudI Process Groups [16].

7.1 Howl

The Howl [11] project by Heinz N. Gies provides many of the same properties of the Riak PG system. Howl provides an application for channel publication and subscription using websockets built on top of Riak Core. Its implementation is very similar to Riak PG, in that quorum-based requests are used to coordinate state changes, and the Mochi Media state monad statebox [13] is used to resolve conflicting values at read and write time.

Howl is geared towards a much more specific use case than Riak PG, and given its currently implementation it would be possible to replace the inner distribution and state facilities with Riak PG.

7.2 CloudI Process Groups

CloudI Process Groups, or cpg [16] by Michael Truog is another project that aims to provide an interface compatible with pg2 but with better availability and partition tolerance. Rather than using global ets transactions, as pg2 does, cpg stores local state in-memory and as nodes are connected and disconnected, state is transferred and conflicts are manually resolved. As far we can tell, the exact failure semantics of cpg have not been tested.

8. Future Work

While we were able to demonstrate that Riak PG provides availability in the face of network partitions and deterministic conflict resolution when partitions heal, we identified a series of areas where improvements need to be made before Riak PG is recommended for production use.

It was observed that garbage collection in the observed-removed set is problematic in that it is still an active area of research [5]. In addition to that, without an anti-entropy mechanism to repair stale replicas, failed nodes could result in data loss. The following section will look at each of these issues and identify future research.

8.1 Garbage collection of the CRDT

The conflict-free resolution properties of the observed-removed set come from its storage of historical information regarding every addition and removal. In the implementation described by Shapiro [14], growth is unbounded, with an object and unique identifier per addition and removal.

The Erlang implementation used in Riak PG uses an optimized version of this set structure which is more space-efficient because additions and removals are tracked per actor, each actor represented by a particular virtual node running on a given node. While this structure limits the growth of the object, from an object per operation to an object per virtual node/node pair, if the cluster experiences the removal of nodes, the unnecessary data will not be garbage collected. It is anticipated that in scenarios such as this, consensus may be required to purge this information from all replicas in the cluster.

8.2 Active anti-entropy mechanism

Similar to the active anti-entropy mechanism described in the Dynamo paper, providing a way to preemptively repair the Erlang dictionaries running on each node is desirable in the event of nodes being replaced.

While values will be repaired at read and write time, if multiple replicas happen to be replaced during a period of time where no accesses to the membership list are performed, the merge and read repair processes will never be invoked, potentially resulting in data loss.

A. Code Availability

The implementation discussed in Section 5 is available on GitHub under the Apache 2.0 License at `http://github.com/cmeiklejohn/riak_pg`.

Acknowledgments

Thanks to Andy Gross, Kenji Rikitake, Steve Vinoski, and Ulf Wiger for providing valuable feedback during the research and development of this work. Thanks to Russell Brown and Sean Cribbs for putting together the first Erlang CRDT implementation.

References

[1] T. Arts, K. Claessen, and H. Svensson. Semi-formal development of a fault-tolerant leader election protocol in erlang. In *Proceedings of the 4th international conference on Formal Approaches to Software Testing*, FATES'04, pages 140–154, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-25109-X, 978-3-540-25109-5. . URL `http://dx.doi.org/10.1007/978-3-540-31848-4_10`.

[2] Basho Technologies Inc. Riak core source code repository. `http://github.com/basho/riak_core`, .

[3] Basho Technologies Inc. Riak dt source code repository. `http://github.com/basho/riak_dt`, .

[4] Basho Technologies Inc. Riak kv source code repository. `http://github.com/basho/riak_kv`, .

[5] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An optimized conflict-free replicated set. Rapport de recherche RR-8083, INRIA, Oct. 2012. URL `http://hal.inria.fr/hal-00738680`.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. . URL `http://doi.acm.org/10.1145/1294261.1294281`.

[7] Ericsson AB. ets - built-in term storage. `http://www.erlang.org/doc/man/ets.html`, .

[8] Ericsson AB. net_kernel - erlang networking kernel. `http://erlang.org/doc/man/net_kernel.html`, .

[9] Ericsson AB. pg2 - distributed named process groups. `http://erlang.org/doc/man/pg2.html`, .

[10] A. Fox and E. A. Brewer. Harvest, yield, and scalable tolerant systems. In *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, HOTOS '99, pages 174–, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0237-7. URL `http://dl.acm.org/citation.cfm?id=822076.822436`.

[11] H. N. Gies. Howl source code repository. `https://github.com/project-fifo/howl`.

[12] C. S. Meiklejohn. Erlang pg2 failure semantics. `http://christophermeiklejohn.com/erlang/2013/06/03/erlang-pg2-failure-semantics.html`.

[13] Mochi Media Inc. Statebox source code repository. `https://github.com/mochi/statebox`.

[14] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, Jan. 2011. URL `http://hal.inria.fr/inria-00555588`.

[15] H. Svensson and T. Arts. A new leader election implementation. In *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, ERLANG '05, pages 35–39, New York, NY, USA, 2005. ACM. ISBN

1-59593-066-3. . URL http://doi.acm.org/10.1145/1088361.1088368.

[16] M. Truog. Cloudl process groups source code repository. https://github.com/okeuday/cpg.

[17] U. T. Wiger. gproc source code repository. https://github.com/uwiger/gproc.

[18] U. T. Wiger. Extended process registry for erlang. In *Proceedings of the 2007 SIGPLAN workshop on ERLANG Workshop*, ERLANG '07, pages 1–10, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-675-2. . URL http://doi.acm.org/10.1145/1292520.1292522.