# PartitionPy : Verifying Correctness of Distributed Data Systems during Network Partitions

Jagadish Venkatraman (jagadish), Arijit Banerjee (arijitb),
*Stanford University*

https://github.com/vjagadish/partitionpy

## Abstract

Distributed data systems can be exceptionally hard to get right. In this paper we present PartitionPy, a framework that can verify the correctness of distributed data systems in the presence of network partitions. PartitionPy can be used to test the correctness of writes in key value stores, pub-sub systems, distributed message queues and distributed databases.

PartitionPy uses a state machine to automatically detect lost writes, simulate the presence of failures like network partitions, nodes being isolated from each other, asymmetric link failures, crashed nodes, delayed packets, dropped packets etc. PartitionPy does this by a systematic exploration of all failure scenarios and verifying that writes are consistent after all failures heal. In addition, PartitionPy can also be configured to simulate a particular sequence of failures to detect errors in replication protocols. We envision PartitionPy as a tool that can be used by the systems community in writing tests that verify writes in the presence of partitions.

We show that PartitionPy actually works by using it to detect bugs in two popular open source projects that are widely used in the industry - Redis, a distributed Key Value store and Apache Kafka, a state-of-the art open source messaging system.

## 1. Introduction

Distributed systems are composed of a lot of interacting components. When operating at the huge scales that that modern internet companies do today, even events that have a small probability of occurrence are almost guaranteed to happen. Engineering such distributed systems requires taking several failure scenarios into account – Networks are unreliable, network links can drop packets. Even worse, partitions can be asymmetric. Nodes could arbitrarily crash and recover. Testing data systems that run on several nodes for correctness is even more challenging because one needs to simulate all such failure scenarios.

We believe PartitionPy is a worthwhile software engineering contribution. A significant portion of development time in popular open source projects is spent around writing integration tests that simulate various failure scenarios. PartitionPy automates testing of all such scenarios by an exhaustive exploration of the failure mode space. PartitionPy includes several out-of-the-box tools to simulate such failures. Clients can just implement a simple API. In fact, our implementation that detected bugs in the replication protocol used by Apache Kafka was less than 60 lines of Python code. PartitionPy provides application developers with a clean framework that they can implement their own custom failure modes on. Clients can also schedule specific failures to happen at points in time and PartitionPy simulates such failures. In addition to automatically finding bugs in implementations, PartitionPy also provides a powerful framework for engineers on which to build tests exercising network failures.

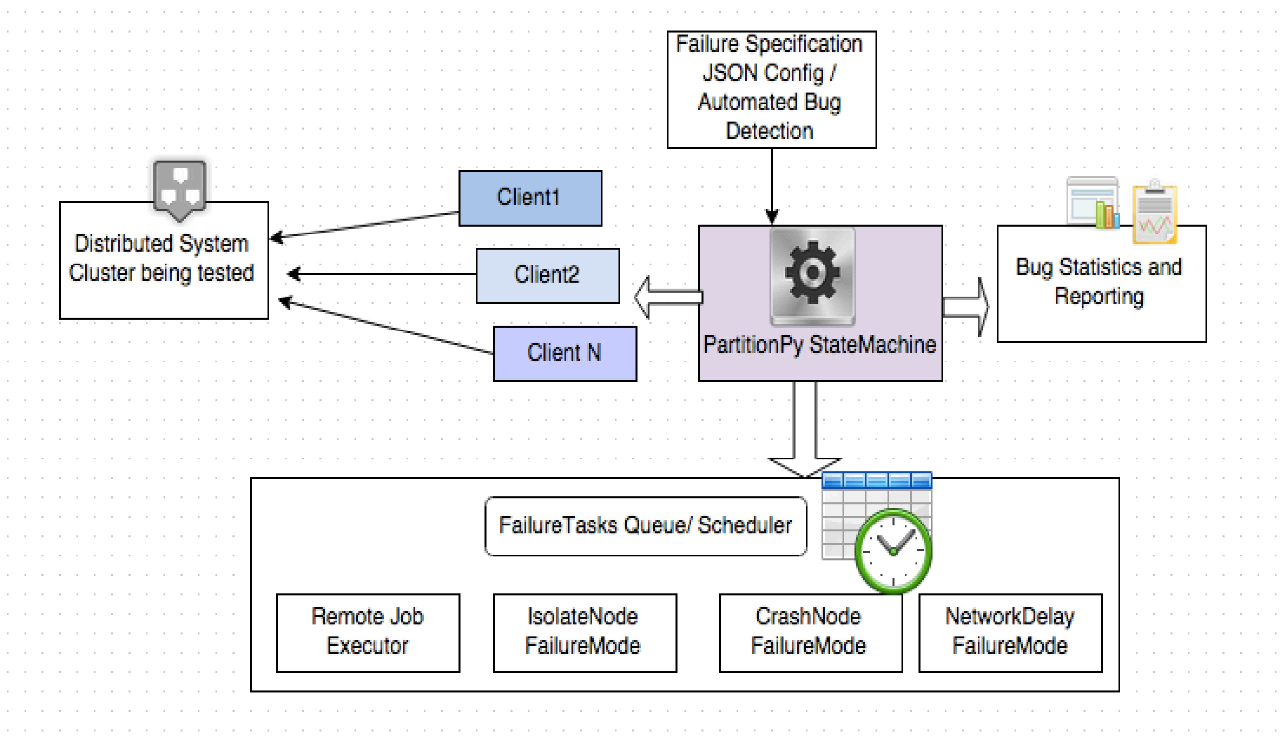

Failure Specification JSON Config / Automated Bug Detection
Distributed System Cluster being tested
Client1
Client2
Client N
PartitionPy StateMachine
Bug Statistics and Reporting
FailureTasks Queue/ Scheduler
Remote Job Executor
IsolateNode FailureMode
CrashNode FailureMode
NetworkDelay FailureMode

**Figure 1: Components of PartitionPy**

## 2. PartitionPy Architecture

At a high level, PartitionPy checks whether writes that the distributed system acknowledged as being successful persist even after failures. A state machine spawns several clients each of which inserts a range of integers into the database cluster. Each client keeps track of which writes succeeded and how long it took. The state machine also spawns a Failure scheduler module that executes failures like partitions, delays and crashes. After a while, the state machine gathers results from all the clients and verifies that all those writes that were acknowledged are indeed present in the state of the cluster. This architecture is partially inspired from Jepsen [1].

The PartitionPy state machine has two modes of operation:

**Automated Bug Detection:**

In this mode, the PartitionPy state machine exhaustively explores all failure modes on all nodes. Common failure modes include those that simulate lost packets, simulate network delay, simulate node crashes, asymmetric links etc. This is useful when the user wants to discover replication / protocol bugs in distributed systems in an automated fashion.

**Failure Specification Config:**

Users define the sequence of failures they want to define in the cluster by providing a config. This is useful when users want to test the behavior of the system under a specific sequence of failure scenarios. This can also detect bugs in client libraries and the way they gracefully handle failover scenarios and connection timeouts occurring due to crash of specific nodes.

A typical configuration looks as follows:

```
[
        {"action":"ISOLATE", "node": "n1"},
        {
        "action": "CRASH", "node" : "n1",
        "killCmd": "ps -effw | grep kafka | grep -v grep | awk '{print $2}' | xargs sudo kill -9",
        "restartCmd": "sudo /start-kafka.sh &"
        }
]
```

We see that workflows involving isolating a node, killing a process on several hosts and restarting it on some other hosts can be easily specified in a JSON like config language.

**State Machine:**

PartitionPy generates the set of failure states that can possibly result from a given initial state and systematically explores each of them recursively. The state machine submits the list of failure tasks to a FailureScheduler that executes all failures at specified times. The state machine then spawns client workers each of which update the state of the cluster by inserting some values. It then measures statistics on acknowledged and lost writes.

**Failure TaskQueue and Distributed Scheduler:**

PartitionPy has a taskqueue that clients can submit FailureMode tasks to. Each task can also be associated with a time to execute the task, a task priority and the time before which the failure is healed. PartitionPy also has a remote job executor that can run arbitrary tasks on remote hosts. All PartitionPy requires is the configuration of the network with nodes and their IPs. PartitionPy contains several built-in implementations of commonly used FailureModes, some of which we list below:

*AddDelayFailureMode:* Introduces a configurable delay in the network by configuring traffic control in the linux kernel at the node. One can also specify a delay that is normally distributed in the parameters. This is written as a wrapper around the linux tc command [2].

We also provide a similar *PacketLossFailureMode* where users can specify the packet loss rate.

*IsolateNodeFailureMode*: Isolates a node completely by causing all other nodes to reject packets that arrive from this node. We use iptables to implement this.

*KillRestartFailureMode*: Kills a process running at a node / a bunch of nodes. Restart it at a specified time.

*AsymmetricCutOffFailureMode*: Blocks a one way link from the two target nodes.

PartitionPy is highly flexible and users can implement their own custom failure modes and schedule them. Default parameters like the nodes in the cluster, the cluster config are passed as arguments to the method by PartitionPy.

Every failure mode implements the following API:

```
def doFailure(self, args):
def doRecovery(self, args):
```

**Client API:**

Any distributed system to be checked for lost writes using PartitionPy should implement the following simple API:

**Put(self, data, args): status**

Writes a key or key-value pair ‘data’ to the distributed cluster. Returns “True” iff the write succeeded. This method is quite general, to allow us to test systems like Queues and KV stores both of which have different data models.

**Get(self, key, args): data**

Returns the data corresponding to the key in the cluster. (This might be a value if it’s a KV store). Returns null if the key is not found.

**GetAll(self, args) : list of available data objects**

Returns a list of available data in the cluster. This will be invoked for reconciling the writes that succeeded and the writes are actually present in the cluster.

**Reporting**

The reporting and statistics module parses the output of each of the client’s logs and computes the writes that were succeeded, writes that failed, writes that were acknowledged successful but not present in the cluster, writes that were acknowledged failed but present in the cluster, and the time for each write. This helps us to understand how changes like failures of nodes and isolating nodes impact the overall latency of our operations.

PartitionPy also has scripts for visualizing data by generating latency graphs.

## 3. Evaluation

We evaluate the effectiveness of PartitionPy as a framework to detect protocol errors in this section. We tested two distributed data systems for the purpose of this analysis – Apache Kafka and Redis. PartitionPy discovered bugs in Kafka's replication protocol and also detected lost writes in Redis.

### 3.1.1 Apache Kafka

Apache Kafka is a popular open source pub-sub messaging system. It is currently the state of the art in messaging with a throughput of 2 Million transactions per second on a 3 node commodity hardware cluster. It is primarily used as a distributed queue. It powers a lot of companies like Twitter, LinkedIn, Netflix etc [3]. A *topic* is a Kafka abstraction to which messages can be published and from which messages can be consumed. Messages sent to a topic are divided into a number of *partitions*. *Producers* publish messages to topics. *Consumers* register and consume messages from topics. A Kafka *message* is any data that is sent / stored in the Kafka cluster by an application. Kafka‘s data is managed by servers called *brokers*. A Kafka *cluster* contains many such brokers each of which are responsible for a log. For each topic a Kafka cluster maintains a partitioned, replicated log.

**Kafka Replication Protocol Design:**
Each Kafka partition has a broker called the *leader*. Leader election happens via zookeeper. The leader is responsible for synchronously replicating messages that are sent to a topic. The *ISR (in sync replica set)* for a topic partition is the set of brokers that contain the replicated messages for the partition. When a leader is unable to contact a particular node, it removes it from the ISR set. The remaining writes can proceed being acknowledged by the new revised ISR set. Kafka claims [4] that it can tolerate upto f-1 failures, in an ISR set of size f. It chooses availability by allowing writes over failing writes when there is less than a majority quorum of nodes in the ISR. However, this protocol has a subtle problem. When the ISR shrinks to less than f/2 nodes, and later, all of them lose their zookeeper lease, a new leader that is elected from the remaining nodes will have no idea about some writes.

**Evaluating Apache Kafka's protocol using PartitionPy**

We set up 4 commodity EC2 nodes each with Intel CPUs – 2.5 GHz, with 1GB of RAM running Amazon's internal build of Linux.

Our Kafka cluster comprises of 3 broker nodes (n1, n2 and n3) with n1 as the leader. We spawned 3 client instances each of which published 1000 distinct non-overlapping keys as messages to the same Kafka topic. Node n4 ran the PartitionPy checker. Node n4 also ran a zookeeper instance that Kafka uses for leader election.

PartitionPy generated this sequence of events in its exhaustive search:

**Event 1:** When the ISR shrinks to less than f/2 nodes. This is simulated by the PartitionPy event *IsolateNodeFailureMode(n1)* called on the leader of the partition.

**Event 2:** The leader loses its zookeeper connection. This is simulated by the PartitionPy event *KillRestartFailureMode(n1)* called on the leader of the partition.

After **Event 1**, the ISR shrinks to one node. Now, all writes are acknowledged only by n1, because PartitionPy isolated the leader by preventing other Kafka brokers from accepting traffic from the leader. Now, for a while writes are persisted only in the leader. At this point, other nodes are completely disconnected from the leader. However, all of them are connected to zookeeper.

Now **Event 2,** causes the leader to crash and hence lose its zookeeper lease. This triggers leader election and a new leader is elected that has absolutely no knowledge of the writes that happened between Event 1 and Event 2. Those writes get lost from the cluster (even if only one node, the leader in the previous term was down). Thus, the cluster loses writes when even just one node is down.

PartitionPy's statistics module then gathered writes from each of the clients and verified whether there were lost writes (writes that were reported successful but not present in the cluster). **19%** of the total committed writes that were acknowledged are lost during network partitions.

| | |
|---|---|
| Writes acked Success | 2691 |
| Writes acked Failure | 309 |
| Writes acked Success **but absent (lost writes)** | 513 |
| Writes acked Failure but present **(spurious writes)** | 0 |

**Visualizing cluster latencies with PartitionPy's graph tool**

PartitionPy's visualization tool generates graphs from latencies recorded in each of the writes. Figure 2 shows the latency of the requests in the cluster. At about iteration 5, PartitionPy performs Event 1 (isolating n1), this causes some requests to fail because of connections to n1 timing out, as n1 attempts to retry and contact n2 and n3 in its ISR set. This explains some failures (red dots) at iteration 5, with extremely high latency of 1.2 seconds. The ISR shrinks to n1 and the latencies get back to normal.

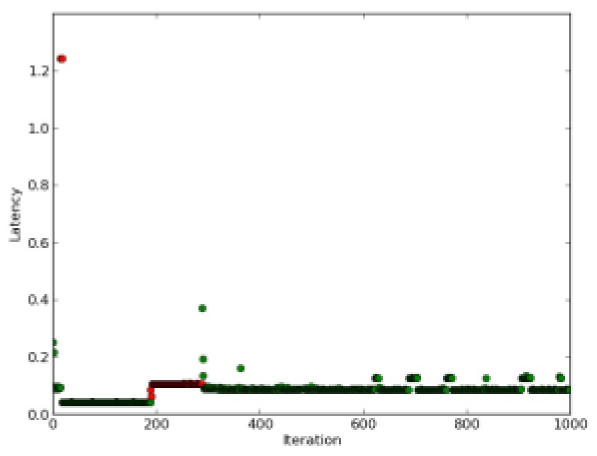


**Figure 2 Kafka Cluster Latency**

The requests till request# 200 succeed. All those writes from Iteration 5 to Iteration 200, were just written to n1. The latency is about 0.1s per request. There was a FailureMode that introduced network latency that was running in the background. There is a latency and failure spike at #200 because of Event 2 (leader crash). The failures continue iteration #300 until zookeeper detects the crash and the nodes n2 and n3 trigger leader election. After #300, leader election happens, requests start succeeding and latencies continue are back to normal. However, the writes from #5 to about #300 that were only in n1 are lost when n1 crashes at #300.

### 3.1. 2 Redis

In this section, we analyze the behaviour of another system, Redis [7], and describe conditions under which PartitionPy discovered writes being dropped.

Redis is a widely used open source key value store. Keys can be strings, sets, lists and other data structures. Redis provides fast performance by working with an in memory data set, though it also provides persistence by dumping data to disk periodically.

**Redis Architecture**

A distributed implementation of Redis usually consists of several master and slave nodes. Each master is responsible for a particular portion of the keyspace. When a write reaches the master, it replies to the client after performing the operation locally. Asynchronous replication is used between the master and the slave nodes to keep this data in sync. If a master fails or is unreachable for a long time, a slave is promoted to master.

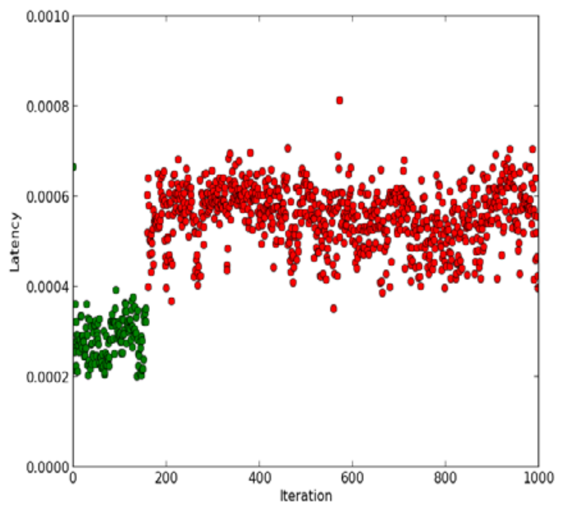
Redis uses Sentinel nodes, which are typically colocated with the Redis nodes to detect node failure (using a gossip based protocol) and perform failover.

Our Redis setup consists of one master node (say n1) and two slave nodes (n2 and n3). Intially, the key value store is empty. The failure mode config simply isolated the master node. After a while, the network partition was healed. All the while, a Redis client was sending write requests for numbers in increasing order in the range 1 – 1000 to the node located at n1.

In the graph, green circles represent successful writes while the red circles are unsuccessful. From this graph, we were able to reason about the following sequence of events leading to data loss in Redis:

**1.** Writes to the master (n1) are successful. **2.** n1 is partitioned away from n2 and n3, for a brief period, writes to it are successful. **3.** The sentinel located at n1 realizes that it cannot communicate with other nodes, changes n1 into a read only slave node. Writes to n1 start failing (red circles begin in the graph). **4.** In the meantime, node n2 (say) is promoted to a master in partition (n2, n3) **5.** When the partition heals, n1 becomes a slave of n2 and syncs data. This sync causes writes to n1 during step 2 to be dropped, since they were never replicated **6.** All later writes to n1 fail since it is still a slave and Redis only accepts writes to the master. (this could have been fixed by querying sentinels for the master before querying)

Although this behaviour may not be unexpected, PartitionPy helped us find one condition for



dropped writes in Redis, without much additional effort (by implementing the Put(), and GetAll() functions).

| | |
|---|---|
| notAcked-Present | 0 |
| Acked success but Lost | 139 |
| notAcked and Absent | 842 |
| AckedPresent | 19 |

**Related Work:**
Bug finding and error checking is a well researched area in systems. Explode[5] is a checker for finding storage systems errors in popular storage systems like databases, file systems, version control systems etc. LineUp [6] is a tool from Microsoft Research used to verify linearizability in systems like the .NET framework.

Our work is more concerned with errors that occur due to lost writes in the presence of network partitions. Most related to our work is Jepsen [1]. Our work differs from Jepsen in the following ways:

Most notably, PartitionPy specifies a single API that all data store implementations need to implement in order to be tested using various failure configs. This allows us to follow the 'write once, test many' paradigm. The idea is that each data store implements this API for easy testing using PartitionPY (we've implemented the wrappers for

Redis and Kafka). Jepsen creates different models and tests for each system.

Secondly, we provide neat abstractions of various conditions to be tested as 'failure modes', and a simple way to describe different failure modes using JSON type configs.

Thirdly, our implementation is in Python, a language more commonly used than Clojure, which is used by Jepsen.

**Conclusion and Future Work:**
In this work we presented PartitionPy, a novel framework that can detect errors in distributed systems. To the best of our knowledge PartitionPy is the only framework that supports these features. PartitionPy contains implementations of several common failure modes and can be used to test distributed storage systems. We demonstrated PartitionPy works and effortlessly found lost writes in Kafka and Redis due to protocol errors. We want to extend PartitionPy to check linearizability in addition to lost writes. We may test the behavior of systems like Cassandra, MongoDb, Postgres during partitions. We plan to open-source PartitionPy and hope that it is used by the distributed systems community as the automated testing infrastructure for several projects involving distributed storage.

**References:**
[1]https://github.com/aphyr/jepsen
[2]http://lartc.org/manpages/tc.txt
[3]Kafka: A Distributed Messaging System for Log Processing, Jay Kreps, Neha Narkhede, Jun Rao from LinkedIn, at NetDB workshop 2011
[4]http://aphyr.com/posts/293-call-me-maybe-kafka, the Jepsen blog post on Kafka
[5]Explode: a lightweight, general system for finding serious storage system errors, Junfeng Yang,Can Sar,Dawson Engler, OSDI 2006
[6]Line-Up: A Complete and Automatic Linearizability Checker, Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan
[7]http://redis.io/, Redis, an open source key value store