RepPy: Replicated Python

Varun Ramesh, Rishi Bedi, Bogac Kerem Goksel

1 Motivation

We present RepPy, a library for porting legacy code to run in a distributed, fault-tolerant environment by monkey patching standard library functions. RepPy allows replicated computation by getting different machines running the same code to converge on a single return value for non-deterministic function calls using a consistent distributed key-value store.

Various applications of this platform are possible, especially in cases where legacy code which was not designed with replication in mind needs greater fault-tolerance properties. This could be in mission-critical systems like flight computers, or across different cloud service providers to provide resilience in the face of datacenter outages.

2 Approach and API

Our implementation can be found at `github.com/rbedi/distributed-python`.

2.1 Consensus Protocol

For consensus, we use `etcd` with the python client library `python-etcd`. `etcd` implements a key-value store on top of the replicated state machine provided by the Raft consensus protocol [2]. Raft provides safety regardless of network timing and availability as long as the network RTT and mean time between machine failures are bounded. These bounds also affect the optimal Raft configuration. `etcd` defaults for the Raft heartbeat frequency and election timeout are 100ms and 1000ms respectively, which optimize for an RTT of 100ms between nodes. `etcd` allows for the customization of these values, suggesting an RTT of 135ms for continental US, 350-400ms for US-Japan and a safe upper bound of 5s for globally distributed nodes. In our experiments, we used the `etcd` default values for timeouts which are safe for machines located in the same data-center. Users would need to tune the Raft parameters for machines that are further apart.

2.2 Replicating Computation Results

2.2.1 Normal Operation Case

The `etcd` key-value store supports linearizable read and write operations as well as atomic multi-operation transactions that allow procedures like compare-and-swap. To replicate return values of non-deterministic function calls, we have each machine do the computation locally when the relevant function call is made, and then attempt an atomic compare-and-write operation with a deterministically generated key and the local computation result as a value. The value is only written to the log if no previous value exists. If a previous value exists, the machine reads this value and returns it as the result of its computation. The atomic compare-and-write guarantees that in this basic case, only one value will be written to the key-value store for each computation, and thus guarantees consistency across all clients.

2.2.2 Handling and Recovering From Failures

Our system guarantees consistency in case of network partitions or member failures, since replicated function calls do not return until a value can be read from the log. As a result, in case of network partitions or member failures, members that are not connected to a majority of the nodes block at the most recent non-deterministic computation locally while allowing the quorum to keep progressing. In case a non-majority of nodes fail and restart, they can easily catch up with the most

recent state of the system as they will be able to re-read the values for earlier non-deterministic calls from the key-value store.

2.2.3 Membership Changes

`etcd` allows dynamic cluster reconfiguration as the Raft protocol supports this through committing the reconfiguration into the replicated log at a future index. This allows users of RepPy to add new `etcd` nodes to their cluster and run RepPy in new machines mid-computation if they so wish. Since all values are stored in the key-value store as long as the computation is running, new nodes will catch up with the rest of the nodes just like a restarted node, and will become full participants after catching up.

2.3 Extensibility

While we provide pre-wrapped versions of some common non-deterministic standard library functions, our API allows clients to replicate any non-deterministic function call as long as its return value is serializable and one of the following is true:

- It has no side effects
- Its side effects are deterministic
- Its side effects only affect a resource shared by all the machines (such as a shared file system or a shared web server)

For the first two cases, the basic case operation is sufficient, while for the third case, we provide a non-idempotent wrapper that guarantees the function is only run once across all the machines.

Some functions may also require additional validation of agreed-upon values (such as ensuring the monotonicity of time values); we discuss some of these issues in the next section.

```
import distributed as reppy
replication_safe_func = reppy.generate_distributed(function_to_wrap)
function_to_wrap = replication_safe_func
```

For user-written functions, our module's wrapper functionality can also be used as a decorator:

```
@generate_distributed
def my_random():
    # generate a random number somehow
    return my_definitely_random_number
```

3 Examples Uses

To validate the utility of our extensible wrapper library and test replicated execution of real programs, we selected several library functions and wrapped them for use in a replicated setting.

3.1 Time

`time.time()` is an obvious example of a non-deterministic function call whose evaluation should be synchronized across replicates.

Since the replicas may not have synchronized clocks, however, it is necessary to ensure that sensible properties of time which programs would expect, and may depend on, are respected. In particular, time should always be non-decreasing. That is, if replicas agree on value v_i as the result of the ith invocation of `time.time()`, the agreed-upon value v_k of `time.time()` upon any invocation $k > i$ should satisfy $v_k \geq v_i$. To enforce this constraint, we simply store the maximum previously returned value, and only accept new values if they are greater than that maximum.

In some cases, simply persisting one node's value to the cluster is not enough. Suppose that, due to a BIOS failure or misconfiguration, one node's Unix time is vastly wrong compared to the others. In this case, we have a separate implementation, where every node tries to write their value to the key using etcd's in-order keys functionality. Using this functionality, etcd treats the key as

a directory, and creates keys of name `key_name/{seq_no}` where `seq_no` is strictly ordered. We can then query for all keys that are under that directory.

The nodes then wait until a configurable quorum n of values have been written. We then get the first n values and perform an aggregation function (in the case of time this is the median), thus returning that aggregation.

3.2 Randomness

Making a pseudorandom number generator like the `random` module behave consistently requires at minimum synchronizing on a random seed at the beginning of replicated execution. This is insufficient in all cases, however, to ensure consistent results across arbitrary replicas, given varying architecture-dependent treatment of floating-point operations. This has been well-documented, for example, in synchronization of game state which depends on floating point-heavy physics simulations. [1]

Thus, we have two options on how to handle replicated use of functions in the `random` module. In cases where all replicas are operating on the same architecture and can guarantee consistent handling of floating point operations, the client program need only agree on a random seed at the beginning of execution. If, however, explicit consensus is needed for each random call, each call to a function in the `random` module can be wrapped by our module. This is clearly more expensive.

3.3 Input

The builtin Python function `input`, used to request the user for a line of input, is blocking. To make this work in a distributed fashion, we converted `input` to be non-blocking. We then wait for either stdin or a value for a key in the etcd cluster. If a line is read from stdin first, we put the line into etcd and read the result. If the key's value is changed first (meaning a line was typed into another instance), we simply return that value, printing it to stdout so that it looks as if it was typed in. Lines are buffered by the shell, so we only perform consensus at the point of a full line of input.

3.4 Non-Idempotent Operations

All of the operations described above involve all nodes running their local versions of a computation, then persisting one of those results (or an aggregate of those results) to the etcd log. This is unsuitable for some cases where only one node should actually perform the operation. An example of this is a file move operation on a distributed filesystem such as AFS or object store such as S3. In the above situations, all of the nodes will attempt to move the file, and only one will succeed while the others will fail (the client code will throw an exception). Unfortunately, the exception might be stored into the cluster before the success result, thus the exception will be replicated and re-thrown on every machine.

To fix this, we used distributed locks, provided by the python-etcd client library. Our strategy is as follows:

- We try to acquire a lock that is uniquely named by appending `_lock` to the name of the function key.
- Once we acquire the lock, we try to read the value at a function key.
- If the key does not exist, run the local version of the function and save the result or thrown exception to the cluster.
- We release the lock and return the result.

We implemented this function as a separate wrapper generator called `generate_distributed_locked`. We wrapped and demoed `os.rename`.

4 Evaluation

We manually evaluated the correctness of our implementation by composing test programs which exercised each of our wrapped functions. To evaluate how much our replication approach slowed

down program completion, we benchmarked our implementation using a cluster of Google Cloud servers. We use a cluster of up to 16 servers, with 8 in `us-west`, 4 in `us-central`, and 4 in `us-east`, each with two CPUs and 4GB of RAM. In our experiments with 8 or fewer nodes, they all belonged to `us-west`; our experiments with 12 nodes consisted of those in `us-west` and `us-central`, and our experiments with 16 nodes consisted of all 16 across the three zones.

We present results run on a test case which combined a varying number of replicated function calls (to `time.time()`) and some fixed non-replicated computation (SVD of a random 900x900 matrix).

The following plots demonstrate the behavior of the system as a function of the number of replicated calls in the program. The x-axis is simply the number of such calls. In theory, this should be normalized by the runtime of the program in the unreplicated case (so the real metric would be number of calls per second of unreplicated program execution time), but this is not important since the runtime of the program barely changes in the unreplicated case with additional calls (since the computation of the underlying function is extremely fast).

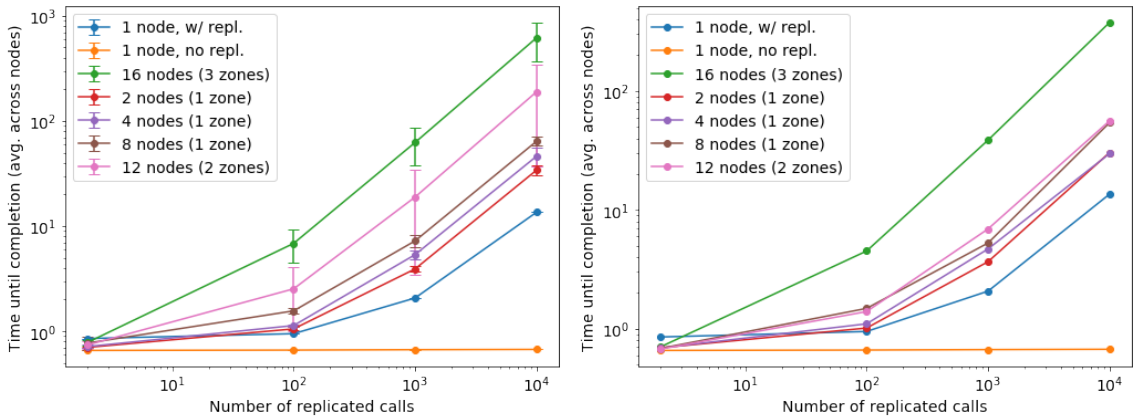

Figure 1: Completion time as a function of number of non-deterministic calls. Left: (a) Average completion time, averaged across all nodes participating in the replicated cluster. Mean ± stdev; Right: (b) Best completion time, across all such nodes.

As expected, completion time scales with number of replicated calls. Significant overhead is imposed by the `etcd` library itself for large numbers of function calls, showing a significant difference between the 1 node without replication (i.e., `etcd` nowhere in the picture) versus the 1 node with replication (i.e., `etcd` creating a single-member cluster and pushing all changes to its local key-value store). Also note the steady increase in computation time between 2, 4, and 8 nodes (i.e., doubling the number of nodes appears to cause a roughly linear increase in computation time). The jump from 8 to 12, however, is more significant - this is because nodes 9-12 belong to `us-central`, while the other eight nodes all sat in the same datacenter (perhaps even on the same rack). Note, however, the similarity between 8 and 12 nodes in Figure 1b: the best-case completion time (i.e., time at which first node in the cluster has finished execution) is the same for both arrangements. This is because the 12-member cluster still has an 8-member group which constitutes a majority that can come to consensus without the other four distant nodes (i.e., $8 \geq 12/2 + 1$). Thus, members of this eight-member group finish as quickly as before, and the four distant members take significantly longer to catch up. A client depending on externalized values from program execution thus wouldn't have to wait any longer for the 12-member case than for the 8-member case.

The 16-member case does not share this property, because now, consensus cannot be found in the co-resident 8 nodes in `us-west`. In order to reach a majority, at least one distant node must be contacted. This drives up the completion time, of course, but also reduces its variance (note the size of the standard deviations for 10,000 function calls for 12 vs 16 nodes).

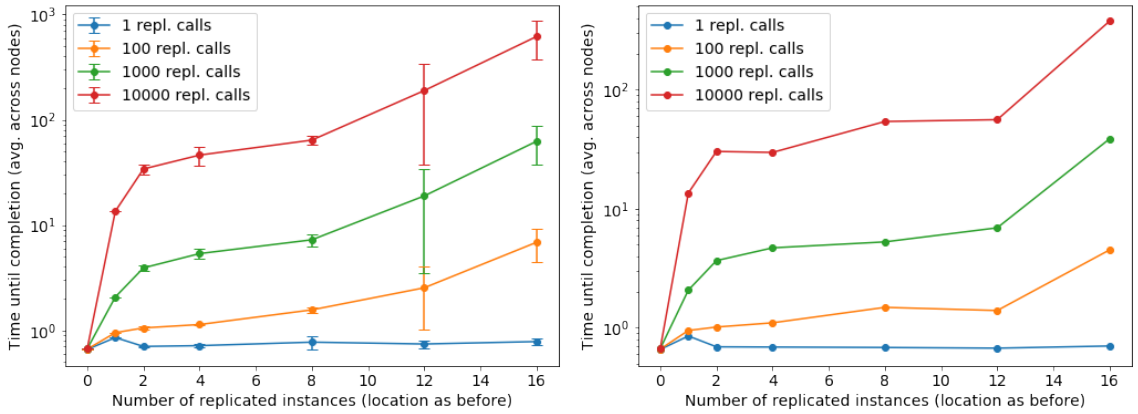

Figure 2: Completion time as a function of number of number of replicated nodes. Left: (a) Average completion time, averaged across all nodes participating in the replicated cluster; Right: (b) Best completion time, across all such nodes.

We can also present these results as a function of the size of the cluster. This demonstrates more clearly the approximately flat-progression in performance for 2 to 12 nodes (where majority can be obtained in one geographic zone), and then the jump at 16 when zones in multiple nodes are needed for progress.

This evaluation characterizes the system sufficiently as a prototype - the next area of evaluation would be real client programs instead of toy examples. Replicating clearly imposes a significant overhead, mitigated by having a majority of nodes be co-located.

5 Future Work

5.1 Locking During Long Operations

For our current locked operation implementation, we use a TTL of 2 seconds to ensure that, if a program crashes while holding the lock, other instances will eventually be able to acquire the lock and continue evaluating. This does not work in cases where operations are long - instead we should attempt to extend the lock while the operation is running.

This introduces two more concerns - if a lock cannot be extended, then we should cancel the operation that is in progress - one way to do this is to run the operation on another thread and kill the thread if the lock expires. However, this can be dangerous and result in inconsistent states. Furthermore, suppose that the locked operation is not atomic and partially completed before a failure. The node that next acquires the lock needs to pick up where the previous node left off, but there is no general way to do this by simply wrapping a function.

5.2 Actually Running Useful Programs

While we demonstrated a proof-of-concept using toy programs for a variety of useful library functions, we did not actually replicate any useful legacy code which was written without replication in mind. This would be a clear objective of future work. The first step would be sampling a variety of Python programs and profiling the source of their non-deterministic behavior requiring synchronization. It would be interesting to automate this by hooking into every line of execution on two replicas, and detecting when their state (on a memory-level) diverge. At each point of divergence, the two replicas' state would be forcibly synchronized so the next point of divergence can be accurately detected.

References

[1] Fiedler, G. Floating point determinism.

[2] Ongaro, D., and Ousterhout, J. K. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference* (2014), pp. 305–319.