Arpeggio: Distributed Redis with Consistent Hashing

Dmitriy Brezhnev, Yifei Huang and Vikas Yendluri
{brezhnev, yifei, vikasuy}@stanford.edu

***Abstract*— Redis[1] is a fast key-value, in-memory database that has gained popularity within the last few years. However, it suffers from some constraints, mainly a lack of fault-tolerance and the fact that all data must fit in memory. In Arpeggio, we create a distributed Redis system with consistent hashing to address these two shortcomings of Redis. The system is completely decentralized, and is based on Chord[2]. Consistent hashing distributes the load. Replication allows the system to tolerate fail-stop failures. Join protocols based on leases and heartbeats guarantee consistency. Even thought Arpeggio sacrifices some of the optimized performance metrics of Redis, optimizations such as client-side caching can be implemented to improve throughput.**

I. INTRODUCTION

Redis is a popular key-value database and was named the most widely-used key-value store database according to DB-Engines.com [3]. Its support of more complex value types, including hashes, lists, and sets coupled with its fast in-memory requirement make it widely used among major tech companies including Twitter, Github, Pinterest, and Flickr.

Nonetheless, Redis is still a new database system and many desirable features are not yet implemented. Redis is designed as an in-memory database, so it works best when all contents of the database can fit into memory. The current version in production runs a single database instance attached to a single machine, largely limiting the amount of information that the system can hold as a whole (because all data must fit on that single machine's memory). Furthermore, Redis does not meet failure-recovery and availability requirements. If the single Redis machine goes down, we can no longer service clients that are requesting information from that machine.

In Arpeggio, we aim to scale Redis both in memory capacity and fault-tolerance by implementing a distributed system of machines running Redis. Each of the N server in the system will hold a portion of the keys, allowing us to store up to N-times the number of keys as one machine. Furthermore, the system automatically adjusts for failures. A key is stored at multiple Redis servers. If one machine goes down, the Arpeggio system will automatically reconfigure and use one of the replicas to respond to the client.

To simplify the key distribution and maximize the flexibility of the system, we decided on a fully distributed system based on a consistent hashing protocol similar to Chord [2]. In fact, according to wikipedia, an arpeggio is a "type of broken chord" [4]. To increase the size of the system, a node simply asks any node in the cluster to join. When a new node joins, it is given an ID and a set of keys that it is responsible for. All nodes keep track of all other nodes for efficient one-hop requests. Because of this, we also use a lease-based join protocol for node joins. Each node runs their own instance of Redis to store its own keys. The client-side API exposes two functions: *get(key)* and *set(key, value)*.

Our design of Arpeggio is advantageous in the following ways:

- **Distributed Load.** Arpeggio aims to distribute the keys among all the Redis servers since the keys for a node are determined by a hash of its port number. Even distribution of keys is essential for even load distribution in most applications.
- **Decentralization.** The system is fully distributed, and no node is more important than another.
- **Availability.** We implement automatic join and failure recovery protocols. The system will automatically adjust keys when a node joins or fails, so we can always find the node that is responsible for a certain key.

In Section II, we describe the detailed design of our system, including API, routing, and node-join protocol, and failure recovery. Section III goes through the implementation details, including communication protocols and spinning up Redis servers. Section IV compares our system to various baselines also built off of Redis. Correctness and future work is mentioned in Section V.

II. DESIGN

In this section we make the following distinctions: *clients* are users of Arpeggio; *client-library* is the library users use to trigger calls to the Arpeggio nodes;

nodes are Arpeggio servers that form the distributed hashing layer between Redis and the clients.

A. API

There are two APIs that are relevant. One is the client-side API, exposed by the client-side library. The second is the Node api, which the client-side API uses to communicate with nodes. The client-side API hides all of the internals of Arpeggio communication to make Arpeggio seem like a key-value store to a programmer. In addition, the client-side API converts the calls made by the client into calls to the node API, which uses redirects to find the correct Arpeggio node to respond to a request.

The client-side API exposes three functions: *connect([Arpeggio node address])*, *get(key)*, and *set(key, value)*.

- **connect([Arpeggio node address]).** This function connects the client to a given server node specified by its address: (ip, port). This argument is optional: If an address is not specified, the client will go through a list of typical node addresses and try to connect to each one. If the address specified is not a server node, an error is returned. This client must issue the connect call before making a call to get or set
- **get(key).** This function returns the associated value of the key. It does not matter that the key may be stored on an Arpeggio node different from the one the client connected to, since redirects are handled invisibly by the Arpeggio-Client API.
- **set(key, value).** This function sets a key to a given value in the Redis databases. Invisibly to the user, the client-side library takes care of the logistics of sending the key to the right Arpeggio node.

B. Routing client requests

When a client-side API call is made, the library translates this call into its node API counterpart. If the library receives a get or set requests, it makes an HTTP request to the node it has been connected to. The node responds with either a ”success” and the value or with ”redirect” and the key and address of the correct node. The method by which the correct node is found is covered in the next section. If ”success” is received, the library returns the received value to the user. If ”redirect” is received, then the library retries with the correct address. The requests are equipped with a TTL field that work is fail-safe for any discrepancies in routing. The client library maintains the routing information received from the Arpeggio node.

C. The Distributed System

Our server system is modeled after the Chord system [2]. The basic layout of the servers is the same: Each server is given an id number between 0 and $2^N - 1$, inclusive. This id number is determined by a hash of its port number. We can envision the nodes as located at their corresponding key on an identifier circle, modulo 2^N. Each node is responsible for storing the keys that are equal to its id number, and those that come between its id number and that of its predecessor. For example, say that we have N=3 and servers with ids 1, 5 and 6. Server 1 is responsible for keys 7, 0 and 1. Server 5 is responsible for keys 2, 3, 4, and 5. Server 6 is responsible for key 6.

Arpeggio uses one-hop to find the correct server to contact for a specific key. It does this by locally storing a list of all the other nodes (*node_list*) so it knows the node to contact for each key (to respond to clients). In order to keep a consistent list among all nodes, we implemented a quorum-based join protocol. Two nodes may have conflicting node lists if one was involved in the join quorum and another wasn't. To resolve this conflict, we used *version* numbers to determine which list is correct (lists with higher *version* numbers are considered most up-to-date).

D. The Join Protocol

When a server wants to join, it contacts any node in the system, which returns the server node's *node_list* and *version* number. The new node will find its successor and contact it, asking it to forward the keys it should hold. During this phase, the successor keeps a *forwarding* flag, which forwards all keys that belong to this new node to the new node, as well as storing it at itself.

To establish quorum, we use leases [8]. After the new node finishes acquiring its keys, it writes down a *lease_timestamp*, indicating the start of its lease. The leases are held for a fixed amount of time (5 seconds in the initial implementation). It immediately broadcasts to all the other nodes, requesting leases and sending them its requested *version* number. Each Arpeggio node will check that the requested version number is greater than what it currently holds, and check that it isn't already holding a lease that hasn't timed out. If those two requirements are met, it updates its *lease_timestamp* and *lease_node*, which indicates which node started the lease, and returns *True* to the new node. Otherwise, it returns *False*.

The new node will wait to see if quorum is reached. If quorum cannot be reached after the lease times out, it returns an error and stops execution. If quorum is reached, it sends its new *node_list* (with itself included) and *version* number to all nodes that have acquired the lease. The quorum must also include the node's successor, for reasons mentioned in the Discussion section.

E. Heartbeats and Failure Detection

If a node is not involved in the join quorum, it needs to learn about the new configuration. To do so, each node repeatedly sends a heartbeat to its successor. The successor will reply with its current *version* and *node_list*. If the successor has a larger *version* number, the node updates its *version* number and *node_list*.

Node failure can be detected with these heartbeats as well. If a successor fails to respond, a node can conclude that it has failed. It will request leases from all other nodes to update the *version* number and *node_list*. No new node can join the Arpeggio cluster during this lease term. Unlike joining, if a quorum cannot be acquired, it will try again because failure recovery is a mandatory operation, unlike joins.

We assume a fail-stop model. When the node comes back up, it can request to join the Arpeggio system just like a new node would. Whatever keys it has missed, it can acquire from its successor.

F. Replication

Each node replicates its keys at its R successors. This is ideal because if a node fails, its keys already exists at the new owner of those keys (the failed node's successor). Each node always maintains a list of its replicas based on its *node_list*. Before it write its own keys, it forwards them to all its replicas. If any of its replicas fail, the write fails. The write must be replicated on all replicas before a node returns to the client.

When a node fails, its predecessor P call for a new configuration change. After the configuration change, each node will get its new replicas. If their replication list or key range changed, they will create background threads to send all its keys to any of the replicas that need it (everybody if key range changed, only new successors otherwise). This way, we can still maintain an R-way replication.

III. IMPLEMENTATION

Our implementation of Arpeggio is written in Python. Each Arpeggio node creates its own instance of redis, where it stores keys and values. Each node also exposes a webserver for communication with clients. The implementation started off of an incomplete implementation of chord in python called PyRope, but our codebase has replaced most of the PyRope implementation and at this point, only uses its RPC framework [**?**].

A. Communication

Communication between Arpeggio nodes is implemented through a Python JSON-RPC library[6]. As our experience has shown, RPC scales poorly to handle client Redis requests, so each Arpeggio node runs a Http server. We used Flask [7] as the web framework. Hence, all communication between the client-side library and the Arpeggio nodes happens via HTTP, whereas communication between Arpeggio nodes (heartbeats, pings, quorum) happens via JSON-RPC.

A client connects to Arpeggio by importing a library that exposes the API, discussed in II.A. The redirects are handled at the client instead of the servers, and are invisible to the user of the API.

IV. EVALUATION

We evaluated the performance of Arpeggio on an Arpeggio cluster of 3 machines. We compared its performance to 2 baselines.

1) "Redis Single": this baseline involves running a single redis instance with no sharding. Thus, all sets and gets go directly to this one redis instance.
2) "Redis Sharded" this baseline involves running 3 redis instances, and deterministically sending gets/sets to one of the 3 instances by taking the md5 hash of the key of get/set and modding it by 3, and sending the get/set to that instance

Our tests ran on a single Amazon EC2 t2.micro instance [5]. By running nodes and clients on the same machine, we eliminated network latency, and as such, our results measure the overhead Arpeggio incurs over using vanilla redis.

Evaluation shows that unfortunately, Arpeggio does not currently satisfy the performance requirements expected of a production system mainly due to communication overhead via gets. Table 1 profiles the time taken to evaluate gets on N keys, where N varies from 10 to 10000. During the test, 10 clients concurrently retrieve N keys. The time this takes grows linearly with N across Arpeggio, Redis Single, Redis Sharded, but Arpeggio is much slower than the baseline systems. In

the table, the Get column Refers to how long Gets take in Arpeggio. R. Single and R. Shard indicate how long the gets take on "Redis Single" and "Redis Sharded."

To infer why gets are slow, we also measured how long it takes to Ping the Arpeggio Cluster. This is the column "Ping" in table 1. The Arpeggio Nodes do no work on a ping, and they do not need to communicate with other nodes; they merely respond; so this test measures how long communication takes to the Arpeggio cluster. We can see that the Ping time is quite close to the Get time if we ping the same number of times as keys we made get requests for. If we subtract Ping time from Get Time (the "Get - Ping" column), the overhead is not too much worse than Redis Single or Redis Sharded. Since our Gets and Sets are serviced over http and use json string serialization, we predict we are adding too much overhead in communication, which is why Pings are as slow as Gets. The Get itself seems to not incur too much overhead, but the communication over the network via HTTP does. Note that inter-node communication happens over RPC which should have less overhead, but the fact that clients connect to the Arpeggio cluster using HTTP means that they pay too much overhead cost.

Table 2 runs a similar test as the test for Table 1, but instead the test has 10 concurrent clients set N keys, where N varies from 10 to 10000. Again, the time taken grows linearly with N across all 3 systems, but Arpeggio is slowest. We subtracted out the ping time (Column "Set - Ping") again to show that set time is not too much worse than Redis, but is slow because of communication overhead.

Table 3 profiles the number of gets and sets that can handled per second by the three systems (Arepeggio Cluster, Redis Single, and Redis Sharded). Specifically, we measured how many gets and sets each system could handle when only 1 client was performing requests to the system, and when 10 clients were performing requests to the system. Redis Single and Redis Sharded have similar performance characteristics, but Arpeggio is not able to sustain the rates that those 2 systems exhibit. Additionally, the get/s and sets/s rate doesn't slow down with the number of clients, but this is perhaps because we are currently bound on the latency of http communication and not on dealing with concurrent client requests in Arpeggio. Redis Single and Redis Sharded both perform much faster when only 1 client is making requests, vs. 10.

We can see from these experimental results that in adding overhead to redis to make it behave in a cluster, Arpeggio currently adds significant overhead to get/set requests, mainly because of using HTTP to service gets to the client. Inter-node communication is not hampered by this problem. Curtailing this overhead (by using straight TCP or UDP, and a faster message passing system instead of JSON-ified strings) is in scope for future work.

TABLE I
AVE. TIME FOR GETS OF N KEYS, 10 CONCURRENT CLIENTS (IN SECONDS)

Keys	Get	Ping	Get-Ping	R. Single	R. Shard
10	0.130	0.123	0.006	0.001	0.001
100	1.92	1.76	0.162	0.094	0.102
1000	20.09	18.04	2.054	1.61	1.66
10000	215.16	180.81	34.353	17.1	18.4

TABLE II
AVE. TIME FOR SETS OF N KEYS, 10 CONCURRENT CLIENTS (IN SECONDS)

Keys	Set	Ping	Set-Ping	R. Single	R. Shard
10	0.185	0.123	0.061	0.001	0.001
100	2.64	1.76	0.876532	0.092	0.105
1000	26.95	18.04	8.91	1.477	1.715
10000	271.94	180.81	91.13	16.84	17.78122

TABLE III
GET/SECOND AND SETS/SECOND

System	Num Clients	Gets/Second	Sets/Second
Arpeggio	1	496	372
	10	464	367
Redis Single	1	18518	18348
	10	5820	5938
Redis Sharded	1	17021	16155
	10	5417	5624

V. DISCUSSION

A. Correctness

We argue that our implementation is consistent. First of all, Arpeggio currently only supports atomic *set* and *get* operations. Naturally, each transaction is serializable.

The system is also linearizable with respect to a single client. This means that if a single client performs

write W_1 before W_2, W_2 will be returned when a following *get* operation is called by that same client. To see this, note that each node holds a list of all other nodes in the Apreggio system. When a node receives two sequential requests from the same client, it will forward them in sequential order to the node with the correct key. The node with the correct key will then process them in the same order. Since Redis is single-threaded, they will be written to Redis sequentially as well. Therefore, Arpeggio is linearizable in the normal case.

We also need to make sure that linearizability holds during joins and failures. When data is being forwarded from an Arpeggio node to a new node, it maintains a *forwarding* flag. Keys in the new node's range are first written at the Arpeggio node from which it receives data, and then written to the new node. The values being forwarded will be processed after the dump, so we guarantee linearizability during joins.

We also guarantee consistency: When an Arpeggio node returns a success for a *set* operation, all following requests for that key should return the updated value. Under normal operation, this is obviously because all requests for that key are routed to a single server. When a node's *version* is not up to date, and it doesn't have the rightful owner of the key, it will route the client to one of the successors of the correct node. Since the join protocol guarantees a node always holds up-to-date information about its predecessor, we will eventually be routed to the correct node, where we can retrieve or set the correct key.

Consistency is maintained in failure recovery as well. When a node fails, the keys will be held at the successor. Each node cannot return to the client until all its replicas have written the transaction. Therefore, every write returned by the failed node also appears in its successor.

B. Future Work

In the future, we would like to implement transactions within Arpeggio. Arpeggio is already designed with leases, so we will aim to use the same leases to implement transactions, similar to Paxos [9]. Leases can be associated with locking certain keys, and transactions can be owned and initiated by the server that receives the request from the client.

We also want to support more Redis operations in addition to *set* and *get*. Redis has functions that append to lists and sets that exist in values associated with a key. These operations are common among Redis users, so it would be good to support them.

Lastly, we want to optimize performance. We believe that better communication protocols (i.e. not JSON-RPC) will allow for faster communication. Also, can also improve performance by storing the *node_list* tables at the clients. Therefore, each Arpeggio node will never need to reroute (unless the client doesn't know about a join or a failure). This could decrease the communication costs significantly. We could also improve performance by assigning node ids better so that keys are distributed more evenly among the nodes.

VI. CONCLUSIONS

Over the past few years, Redis has become a very popular choice as a fast in memory key value store. However, it comes with sacrifices to achieve this purpose. For example, it relies on being able to fit all of its content into memory and it has no replication. Arpeggio tackles those problems, providing a consistent and fault-tolerant system while scaling Redis. In exchange, we sacrifice performance. However, performance could improve with optimizations such as caching at the client.

REFERENCES

[1] Redis. http://redis.io/

[2] Stoica, Ion, et al. "Chord: a scalable peer-to-peer lookup protocol for internet applications." Networking, IEEE/ACM Transactions on 11.1 (2003): 17-32.

[3] DB-Engines Rankings. http://db-engines.com/en/ranking/key-value+store.

[4] Wikipedia: Arpeggio, aka "broken chord". http://en.wikipedia.org/wiki/Arpeggio

[5] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2

[6] Json RPC Python Library. http://json-rpc.org/

[7] Flask. http://flask.pocoo.org/

[8] Gray, Cary, and David Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. Vol. 23. No. 5. ACM, 1989.

[9] Lamport. Paxos made simple. http://www.scs.stanford.edu/14au-cs244b/sched/readings/paxos_made_simple.pdf

[10] PyRope. https://code.google.com/p/pyrope/