# Distributing Knowledge: A Distributed Database System for Universities

Kruttika Jain
Computer Science and Engineering Department
University of California, Santa Cruz
Santa Cruz, California, USA
kjain4@ucsc.edu

Juan Lozano
Computer Science and Engineering Department
University of California, Santa Cruz
Santa Cruz, California, USA
juclozan@ucsc.edu

## ABSTRACT

Libraries have been using technology for automatization since the 1980s. Integrated Library Systems have been an important tool that changed processes and improved the management of resources [2]. Distributed Systems can be seen not only as operations running in different computers interconnected but more broadly consider that the software on top of that network was also designed for such configuration [3].

This project explores the domain of Integrated Library Systems and Distributed systems, particularly proposes distributing the database of registered books.

The goal of this project is for students to be able to request and return books on-line or through the library faculty. The redis key-value store and Python have been used for distributing the data into several replicas and ensuring consistent transactions. The RAFT algorithm is used to manage consensus and for fault tolerance.

## CCS CONCEPTS

Information systems • Data management systems • Database design and models

## KEYWORDS

Distributed Database, Consensus algorithm, Byzantine Agreement, Integrated Library Systems

## 1. INTRODUCTION

A library management system is a resource planning and management software that helps libraries track the number of resources, who has loaned which resource and the return date. The system must allow the addition of new books to the database. There must be both 'reserved' and 'regular' to view. The system should be accessible by all students through either their personal computers or the library desk. The quantity of the books must be updated when it is loaned out as well as when it is returned. The database must provide a consistent view to all the students. The design specifications of The University Library Database, Distributing Knowledge, are the following:

1. The total books, journals, etc. with quantity.
2. The names and quantity of books available to be loaned out currently
3. The book is type reserve or not.

The major use cases of this system are:

1. View the books available on loan
2. View the type of book
3. Loan a book
4. Return a book

Using this specification and also considering the scope and development available time, we used the Key-Value database technology to store the data. Also, a convention is used to create keys that allow us to identify the subsequent values.

## 2. PROTOTYPE

### 2.1 The Database

The prototype consists of a Redis Database, which is a Key-Value database, in-memory, and designed for durability technology that implements the built-in possibility of a distributed architecture. Redis supports various data structures such as lists, strings, maps, bitmaps, sorted sets, etc. Redis is an open source software originally developed by Salvatore Sanfillipo[6] The documentation suggests a master-slave approach to reach consensus. For this implementation, the dataset is in the form of a nested dictionary.The Keys are created in a way that lets the user know if its a books or a student. The book keys are composed of letter 'B', followed by three numeric digits. The values are stored in a nested dictionary composed of six tuples, such that "book" refers to the book name, "author" and "year" refer to themselves, "quantity" is the number of books currently in the library, "nloaned" is the number of books loaned out and "type" indicates whether the book is regular or reserved. At this point, the transactions we have implemented do not need to consider this field. The student key starts with 'S' followed by three numeric digits. These values also consist of a nested dictionary in which "student' refers to the

student name, “bookloaned” refers to the name of the book that has been loaned by the student and the “returndate” refers to the date on which the book should be returned. At this point of time, we do not consider this field.

{Key: BXXX:

{"book": str "Val",

"author": str "val",

"year": int val,

"quantity": int val,

"nloaned": int val,

"Type": str "Regular/Reserve"}}

{Key: SXXX:
{"Student": str "A",
"bookloaned": str "val",
"returndate": int val}}.

## 2.2 Redis Client and Redis-Py

Once the dataset’s data structure has been defined, the Redis server’s basic functionality was tested using their client interface. ‘redis-cli’ is the command line interface that can be used to send commands and receive replies from the redis server. A redis client is created for each of the four nodes. Commands such as ‘KEYS *’ can be used to retrieve all the keys of the database on each of the nodes to ensure data has been replicated on all the nodes. The Python programming language was used to implement the project modularly using the redis module built for python named Redis-Py by Andy McCurdy [3]. This module requires a running redis server and thus and can be installed using pip. Redis-py has a ‘pipeline’ functionality that was used to initialize the database, to loan books and to return books. This functionality reduces the number of back-and-forth packets between the redis client and server by buffering multiple commands to the server in a single request and thus leading to high performance. Thus using Redis-Py python objects and pipes to Redis server were created to store and manipulate the data on these servers.

Firstly, Redis-py was used to create the pipe to the server at port 8100 and then to implement three operations: Database population, LoanBook, and ReturnBook. Data Population creates the data of the library management system. LoanBook modifies the field “Quantity” by subtracting one unit from the current and by adding one unit to the field “nload. This transaction also adds the book name to the “bookloaned” field of student item, so the library knows who has loaned which book. ReturnBook operation can be seen as the complement of LoanBook, given that it subtracts one from “nload” and adds one to “quantity” in the book item that is being returned and also deletes the book name from the “bookloaned” field of the student item. To make this database distributed and to support transactions with ACID properties, other redis servers on redis ports 8101, 8102 and 8103 are also initialized and the library() class is called through their clients. Thus this prototype consists of four nodes. The library() class is initialized on all the ports only to ensure that the node has the database populated in it before accepting any transactions. Post this initialization, all requests to loan a book or to return a book can be made on any of the four nodes and the update performed through this transaction would be replicated to all the other nodes. for example, if student s001 requests to loan a book b001 (on port 8100) which has quantity 3, then the quantity of the book is reduced to 2 and this is replicated to port 8101, 8102 and 8103. Thus if another student, s002 tries to loan book b001, from any of the four ports, they will see that the quantity of book b001 is 2.

# 3. FAULT TOLERANCE

The RAFT [4] algorithm is used for consensus and fault tolerance. RAFT reaches consensus by electing a leader. The leader manages log replication and remains the leader until the node fails or disconnects. The RAFT consensus has two phases:

Leader Election: A node that has not received the leader’s message over the election timeout period, starts by increasing the term counter. Then it votes for itself as the leader and sends a message to all the other servers requesting their vote. If it receives a majority of the votes, it becomes the leader.

Log Replication: Each of the client requests are sent to the followers as AppendEntries messages. In case the follower is not available, the leader retries until it is eventually stored in the follower. Once confirmation of the majority is received, the transaction is considered committed.

In case of leader crash, the new leader compares its log with the logs of the followers and discards all entries after the last common entry.

The PySyncObj [5] library(written in Python) is used for consensus and for building a fault-tolerant distributed system. PySyncObj uses RAFT for for leader election and log replication. First, PysyncObj is installed using pip and then the ‘SyncOb’j and ‘replicated’ functions are imported from it.

The ‘library()’ class inherits the SyncObj. SyncObj is initialized to the self-address, and the partner addresses using super(). The super() function is used to implicitly refer to the superclass and thus while referring the super class from the subclass, the name of the superclass does not need to be written explicitly. In our case, the superclass on a given node is the local address of that node and the subclasses are the local addresses of the other three nodes. For example, for the node on redis port 8100, the superclass is the address ‘localhost:4321’ and the sub classes are ‘localhost:4322’, ‘localhost:4323’ and ‘localhost:4324’. Next, the ‘@replicated’ decorator is used. A decorator in Python is a design pattern that allows a user to add new functionality to an object without changing it’s structure. In our system, this decorator is used before each of the functions- init_db, loanitem, and returnbook to ensure replication.

A file exists for each of the four nodes. While the class is initialized on every node, the class object and functions may be called on any node, and they will be replicated to the rest of the nodes.

The pdb module is used as breakpoint such that none of the python files execute and terminate without explicitly quitting.

All loanitem and returnbook transactions are done atomically on any of the servers and replicated consistently. Moreover, on

shutting down node 4( port 8103), the rest of the nodes perform consistently and without any interruption

## 4. BYZANTINE FAULT TOLERANCE

A Byzantine Fault tolerant system is one which overcomes the set of failures that come under the Byzantine Generals problem. This is the most difficult set of failures. A byzantine fault occurs when a node fails and there is inconsistent information on whether the node has failed or not. The node may act arbitrarily or maliciously, leading to the database reaching an inconsistent state. Thus a server in a system may appear failed to some nodes and working to others. This project uses the RAFT consensus which elects a leader to manage log replication. However, RAFT is not equipped to deal with malicious nodes that act arbitrarily instead of failing or shutting down. For example, in the case of this project, if the server at port 8100 is the leader and 8101, 8102 and 8103 are the followers, and if one of the nodes, say 8102 turns malicious, then Byzantine Fault Tolerance would be required. Oral Messages Algorithm could be used to tolerate this failure, however this allows forged messages and requires that the number of faulty nodes be lesser than one-third of the total number of nodes. Thus in this project, only one node could be faulty for Oral messages to be able to tolerate Byzantine faults. Another solution is that of the Signed Messages algorithm. For the given scenario, 8100 will first send a signed message to 8101,8103 and 8102. Since 8102 is malicious, it will send a wrong or different message to 8101 and 8103. But since 8100's signature cannot be forged and any alteration to signed messages can be detected, thus both 8101 and 8103 will know that 8102 is malicious and consensus will be reached between 8100, 8101 and 8103 only. Another method that could be applied is PBFT(Practical Byzantine Fault Tolerance)[7] that gives very high performance and state machine replication. In the case of this project, there would be one primary (say 8100) and the rest of the three nodes would be replicas. A client request would be sent to the primary, which would multicast it to the replicas. The replicas at 8101, 8102 and 8103 would execute the request and send a reply to the primary. On receiving two (considering there is one malicious node in the system) replies from different replicas, a commit would be performed. Some open-source libraries such as UpRight[8] are also available that tolerate regular as well as Byzantine failure but may be difficult to customize towards the needs of individual projects.

## 5. PERFORMANCE ANALYSIS

The performance analysis included a *Main* program that executes the population (and replication) of the keys and values in the redis server. To work in this environment, it was necessary to execute four python scripts at the same time. This was done using the terminal splitting tool for linux distributions named Terminator. Through this tool, once the four terminals are displayed, one can broadcast commands, and thus the four scripts are executed simultaneously. Testing of various use cases was performed, for instance, execution of 'loanitem' for the same book at the same time by different students. In this case if the quantity of books available was larger than the number of number of requesting the book, then the system executes correctly and each student gets one copy while the loaned book *quantity* is also updated correctly. Another use case is when the number of requests for the same book are greater than the available copies. In this case the systems internally orders the transactions and the first ones gets the copy while the last one will receive a message of unavailability. There can thus be seen as random assignment of who gets the copy.

For the Performance Analysis a logfile was implemented from Python's datetime library. This file kept track of every *loanitem* transaction, i.e as soon as a *loanitem* was invoked, the exact time was measured and the exact time at the end of the transaction was also logged. The table shows the execution time of 51 loan transactions, the maximum value is 23.95 ms, the minimum value is 1.47 ms and the Median is 10.23 ms. Considering the values in this table, it is clear that a certain wait time should be provided between transactions, and it should be approximately 40 ms. Testing for the same was performed in a linux envorinment with four different redis nodes on a local machine. Taking network latency into account, this number should be higher for a more realistic implementation scenario.

**Table 1. Execution times(ms) of 51 loanitem transactions**

| Transaction Time (ms) | |
|---|---|
| 17.677 | 2.619 |
| 10.377 | 13.371 |
| 4.862 | 14.078 |
| 14.796 | 10.486 |
| 14.699 | 4.808 |
| 9.77 | 15.021 |
| 5.435 | 12.995 |
| 10.025 | 22.463 |
| 15.867 | 13.848 |
| 20.461 | 6.535 |
| 19.619 | 3.011 |
| 7.772 | 1.471 |
| 3.539 | 13.751 |
| 1.971 | 10.371 |
| 14.797 | 4.842 |
| 10.238 | 12.452 |
| 5.096 | 13.118 |
| 12.734 | 9.29 |
| 7.669 | 4.771 |
| 3.614 | 12.799 |
| 4.237 | 12.275 |
| 12.066 | 23.955 |
| 6.197 | 2.895 |
| 10.501 | 8.095 |
| 5.352 | 6.352 |
| 5.104 | |

## 6. CONCLUSION

The system implemented is a distributed library management system for universities. The redis key-value store was used as it provides high performance through in-memory storage and access. Furthermore, Redis-Py was used to integrate redis in the Python environment to populate the database and define transactions using Python. Four nodes were created at port 8100, 8101,8102 and 8103. Database initialization and transactions were defined on all nodes. Any transaction performed on one node was replicated to all the nodes. The RAFT algorithm, which uses a leader-follower model, was used to implement consensus and to provide fault tolerance. Possible methods to implement Byzantine fault tolerance were discussed. Finally, the main goal of providing a distributed database system that guarantees ACID properties was achieved and tested. Testing was done by running simultaneous transactions, first by multiples clients accessing different data and then with two clients accessing the same data, considering both cases where the number of resources are greater or lesser than the number of client requests. Moreover, fault-tolerance was also tested by shutting down one node. In all cases, there were no concurrency errors.

## 7. REFERENCES


[1] Wang, Y., & Dawes, T. A. (2012). The Next Generation Integrated Library System: A Promise Fulfilled?. Information Technology and Libraries , 31 (3), 76-84. https://doi.org/10.6017/ital.v31i3.

[2] Czaja, L. (2018). Introduction to distributed computer systems: Principles and features . Springer. DOI https://doi.org/10.1007/978-3-319-72023-4.

[3] Andi McCurdy Github, Retrieved from https://github.com/andymccurdy/redis-py.

[4] Raft Github, Retrieved from https://raft.github.io/.

[5] PySyncObj GitHub, retrieved from https://github.com/bakwc/PySyncObj

[6] Redis retrieved from https://en.wikipedia.org/wiki/Redis

[7] Miguel Castro & Barbara Liskov(1999). Practical Byzantine Fault Tolerance. Proceedings of the Third Symposium on Operating Systems Design and Implementation

[8] UpRight Google code archive retrieved from https://code.google.com/archive/p/upright/