Secure Distributed Data Base for IoT Systems

Nicola Ruaro, Atefeh Mohseni

1 Introduction

We're collectively trying to capture fine-grained, ongoing data from a fast-growing universe of IoT devices because the more data, the better the analyses are possible from that data. On the other hand, many IoT applications rely on small, non-rechargeable batteries, so reducing the operations overhead on devices is very important. So there are two major challenges with IoT in the cloud: The rapid growth of data, and the lack of IoT data security. In this project we tried to addressed these two main issues by developing a secure and distributed database for IoT systems.

2 Security and Data Persistence Objectives

As mentioned above, security and availability are our two main objectives here. For security, we mainly focused on confidentiality meaning the data has to be secure on the motion and all the access attempt to data has to be authorized.

2.1 Access Control

Since, our database is publicly available through a REST API we need to assure that servers check client's credentials before executing their queries. "HTTP Basic Authentication" is a method for an HTTP user agent (our clients) to provide a user name and password when making a HTTP request. It's a simple and lightweight technique for enforcing access controls because it does not require cookies, session identifiers, or login pages. In order to protect client's password at servers we used "bcrypt". Bcrypt is an adaptive password-hashing function that iteration count can be increased over time to make it slower, so it remains resistant to brute-force search attacks even with increasing computation power. We used "libbcrypt" library [3] which is written in C++.

2.2 Data Encryption

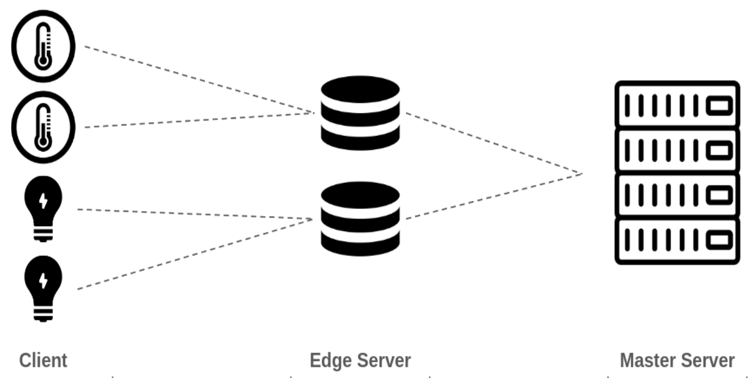
We need to guarantee the confidentiality of the data in transit when data actively moving from clients to servers across the internet. Therefore, we assign certificates to clients, edge servers, and master servers to use encrypt messages before it sends out using HTTPS. "cpp-httplib" [1] provides both HTTP and HTTPS connection in C++.

2.3 Data Persistence

In order to ensure availability of our service and reduce the overhead on clients, we designed our system in a multi layer architecture which you can find the details in the next section.

3 Implementation Details

In this section, we describe the system architecture and design choices we made.



We implemented the whole project in C++ [6], but for the evaluation we also implemented a Python version of the client. In the following we explain main components of our system.

3.1 Client

Client send read/write/delete queries to the edge server. In order to improve performance, client put `write` and `delete` requests in an ordered queue and sends query request in a batch. This eliminates the redundant write/delete requests on the same key into database. There are two main threads at the client, one for network operations and the main thread. Therefore, the main thread is not blocked during network communication.

3.2 Edge Server

We implemented our own key-value format database which edge-server inherits from. We define three main queries for read, write, and update. Edge server is meant to be close to the clients and we deployed a REST API for connection between clients and edge servers. The main functionality of edge server is to authorize clients and execute their queries on database and return the result. Since we used REST API, we used `Put` for write query, `Get` for read query and `Delete` for delete query.

3.3 Master Server

It's important that we guarantee the data persistence among edge servers. We need to make sure if edge servers crashed we don't lose access to it's database; therefore, we also implemented a master server. Master server receives backup files from the edge servers and store them on a secure, permanent disk storage. Since, there might be a similar key being used by different edge servers, master-server does not translate backup files into a database object and store them as a `json` file.

4 Evaluation

4.1 Profiling and Performance Considerations

Our method for profiling is offline mainly to check program's resource usage. We used Linux `time` utility with custom format string as shown below:

```
#!/bin/bash
/usr/bin/time -o "output-file" --append -f "%e real,\t%U user,
\t%S sys,\t%P CPU, \t%Mk max mem,\t%F major pagefaults,
\t%R minor pagefaults" pypy3 client.py);
```

The syscall used in `time` command is wait4. In linux, the parent process can query whether the child process terminates through the wait4() system call [4]. wait4() function can obtain not only the status information of the sub-process, but also the resource utilization information of the sub-process, which is obtained through the parameter usage.

4.1.1 Runtime Systems Benchmarks

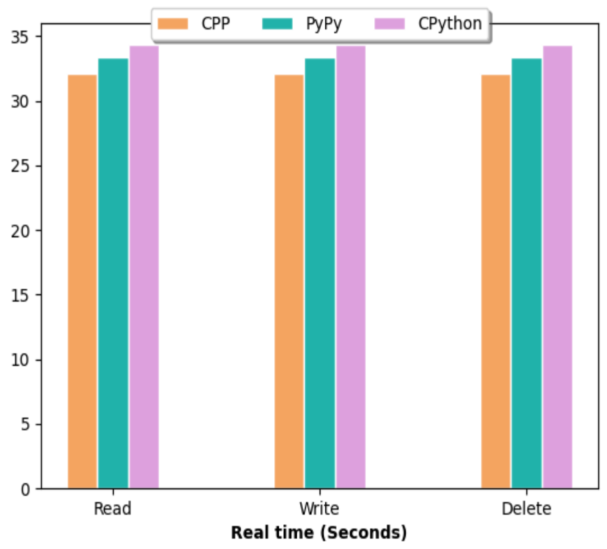
In this project we worked with `PyPy` and `CPython` runtime systems which briefly described below.

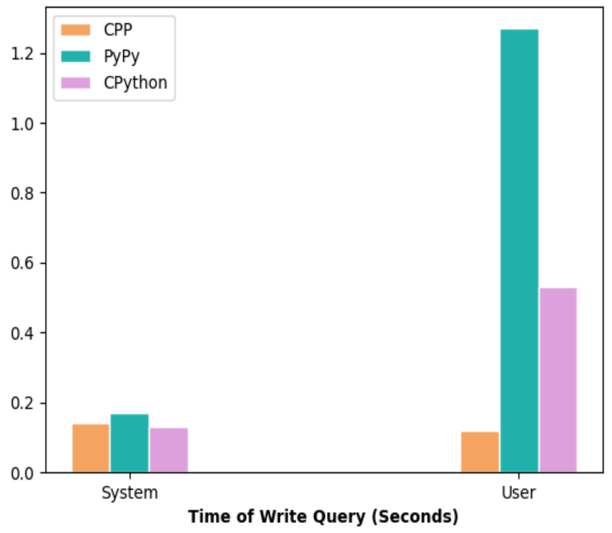
Table 1: Comparison between PyPy and CPython

	PyPy [5]	CPython [2]
Runtime System	just-in-time compiler	Mixed mode (interpreter with profile guided optimization)
Garbage Collection Algorithm	Incminimark (incremental with generational moving collector)	Reference Counting

4.1.2 Profiling - Execution Time

Here, we are comparing the execution time of the program within different runtime systems.

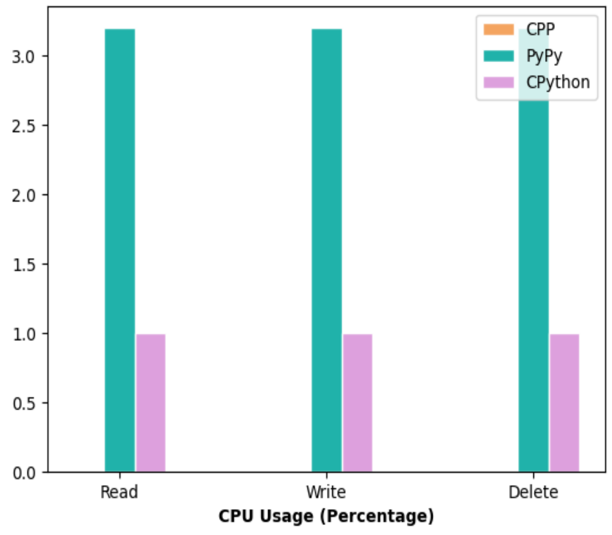




`real time` refers to actual elapsed time - time from start to finish of the call. `User` is the amount of CPU time spent in user-mode code within the process. This is only actual CPU time used in executing the process. Similarly, `Sys` is the amount of CPU time spent in the kernel within the process. As you can see the figures above, our program spends most of the time in user mode. `PyPy` is the slowest among the CPython and C++ executable mainly due to the time PyPy runtime spent for compilation of the codes that runs once in our program. The left chart, shows our program has a lot of network connection and while CPU is not involved, the process is waiting for network responses.

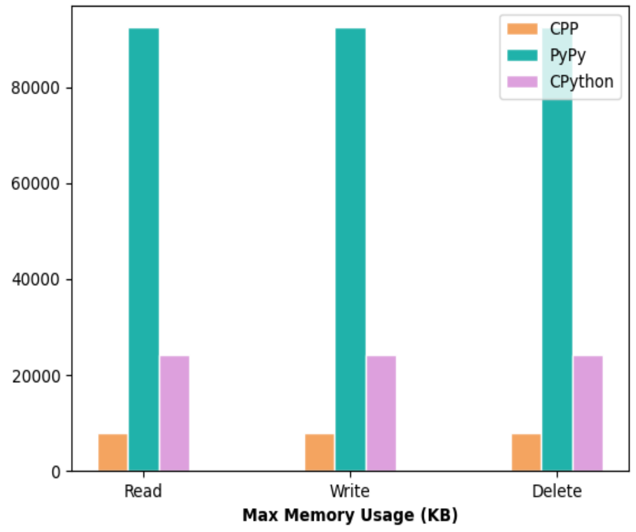
4.1.3 Profiling - CPU Usage

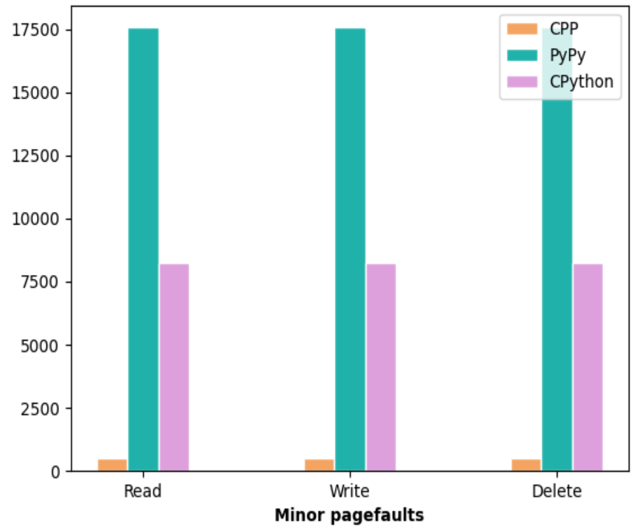
As mentioned above, our program does not have heavy computation and CPU is not too involved. You can see the CPU usage of the program in the chart below. C++ CPU usage is 0% but PyPy used CPU up to 3%.



4.1.4 Profiling - Memory Usage

Based on the memory usage result, PyPy used more memory than the others and it refers to the memory PyPy takes for compilation purposes. In the right chart, the number or minor page faults are shown. Since, CPython uses reference counting GC it reserves the locality of references in memory and causes less page faults.





5 Conclusion

In this project we learnt about multi-layer design implementation in IoT systems and main challenges of distributed databases. Since, the performance is important in IoT we implemented our project both in C++ and Python to closely evaluate the difference. We learnt that the choice of runtime system is dependent on the type of application. For instance, since there was no loop or repetition in our code, CPython- a mixed mode runtime system outperforms PyPy -a just-in-time compilation.

References

[1] C++ header-only http/https server and client library. https://github.com/yhirose/cpp-httplib.

[2] Cpython. https://github.com/python/cpython.

[3] libbcrypt library in c++. https://github.com/trusch/libbcrypt.

[4] Linux kernel learning notes: wait4. https://programmer.help/blogs/linux-kernel-learning-notes-4-wait-waitpid-wait3-and-wait4.html.

[5] Pypy's documentation. https://doc.pypy.org/en/latest/.

[6] Secure distributed database for iot. https://github.com/atefehmohseni/IoT_secure_distributed_database.