Distributed DB for Iot Systems

Nicola Ruaro, Atefeh Mohseni

1 Introduction

In a fast-growing universe of IoT devices, we are collectively trying to capture more fine-grained, continuous data, because such data enables more solid and useful analyses. However, many IoT applications rely on small, non-rechargeable batteries, and reducing the operations overhead on devices is very important. There are two significant challenges with IoT in the cloud: the rapid growth of data and the lack of IoT data security. In this project, we 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 in this project. For security, we mainly focus on confidentiality, meaning that the data must remain secure while being transmitted, and all the access attempts to such data have to be authorized.

2.1 Access Control

Our database is publicly available through a REST API. For this reason, we need to ensure that our servers validate the client's credentials before executing their queries. HTTP Basic Authentication is a method for an HTTP user agent (an IoT client) to provide a username and password when making an HTTP request. It is a simple and lightweight technique for enforcing access control, and it does not require cookies, session identifiers, or login pages. In order to protect the stored client's password (in the server), we use `bcrypt`. `bcrypt` is an adaptive password-hashing algorithm that allows increasing its iteration count to make the algorithm slower. For this reason, `bcrypt` is guaranteed to be resistant to brute-force attacks and rainbow tables even at the increase of the attacker's computational power. To implement our access control mechanism, we used the `bcrypt`[1] library, written in C++.

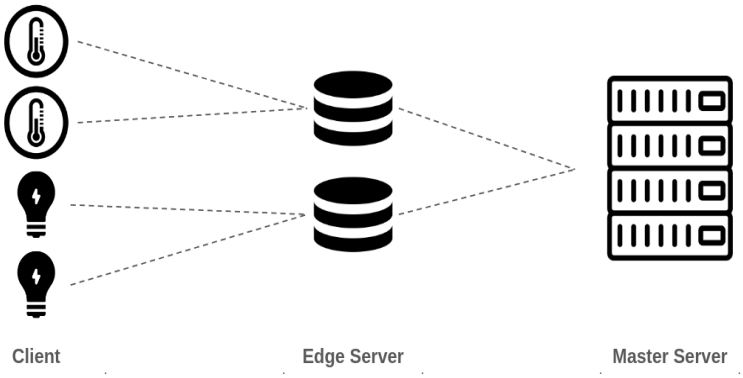
2.2 Data Encryption

In order to guarantee the confidentiality of the data while in transit–from clients to servers, as well as from the edge servers to the master servers–we assign certificates to clients, edge

servers, and master servers and encrypt messages before sending them out using HTTPS.

3 Implementation Details

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



We implement the whole project in C++ [5]. However, we implement the client service in both Python and C++ to compare the two implementations. In the following section, we explain the main components of our system.

3.1 Client

The clients send `read`, `write`, and `delete` requests to the edge servers. In order to improve performance, the clients put `write` and `delete` requests in an ordered queue. This allows sending the requests in batches and eliminating redundant requests (e.g., on the same key). Our client implementation runs on two worker threads: a network thread and the main thread.

The main thread collects the data from the environment and initiates the requests to communicate with the edge server. The network thread optimizes and dispatches the requests to the edge servers. For this reason, the main thread can continue to collect data and does not have to wait for the network communication–since the network latency can be high.

3.2 Edge Server

We implement our key-value store database, which is used heavily by the edge servers. We define three main database queries: `READ`, `WRITE`, and `DELETE`. The main functionality of the edge server is to authenticate clients, execute their queries on the database, and return the query result. In our implementation, we use a REST API with three different endpoints for client-server communication.

3.3 Master Server

In a distributed database, it is crucial to guarantee data persistence among the edge servers. In order to guarantee a reliable database with low data loss (e.g., in case of a server or network failure), we implement a third architectural component: the master server.

The master server receives backup files from the edge servers and stores them in a secure, permanent disk storage. The master server does not store the backup files in a database object but stores them to–and restores them from–`json` files.

4 Evaluation

4.1 Profiling and Performance Considerations

We profile the client execution on a large set of queries. Our profiling is mainly offline and allows us to assess the implementations' resource usage. The main profiling tool that we use is the Linux `time` utility with a custom format string (as shown below):

```
time -o /tmp/profiling --append -f '%e real,%U user,%S sys,\
%P CPU,%Mk max mem,%R minor pagefaults' pypy3 client.py
```

The Linux `time` utility uses the `wait4` [3] syscall to wait for the termination of the child process and retrieve both its status information and its resource utilization. The kernel returns the resource utilization metrics in a dedicated memory buffer after processing the syscall.

4.1.1 Runtime Systems Benchmarks

In this project, we worked with the `PyPy` and `CPython` runtime systems (as well as the C++ runtime), which we briefly describe in the following table:

Table 1: Comparison between PyPy and CPython

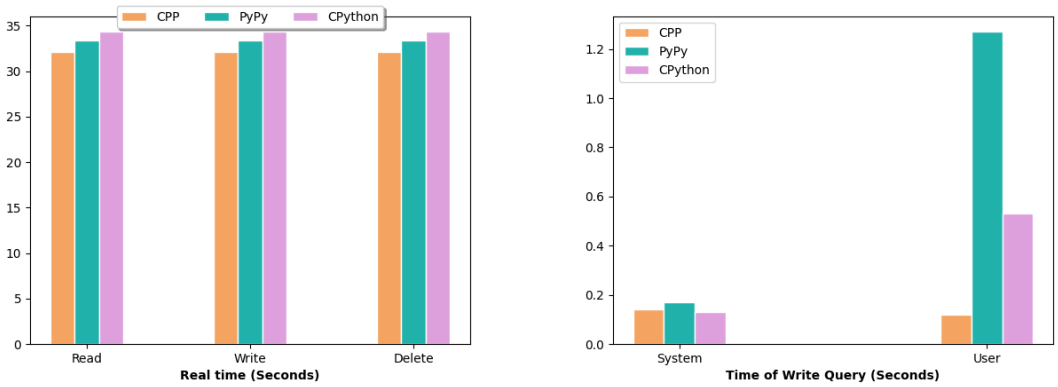
	PyPy [4]	**CPython** [2]
Runtime System	Just-In-Time (JIT) compiler	Interpreter
Garbage Collection Algorithm	`incminimark` (incremental, generational moving collector)	Reference Counting

4.1.2 Profiling - Execution Time

In this section, we compare the execution time of the client program within different runtime systems.

`real` time refers to overall elapsed time. `user` time indicates the amount of CPU time spent in user-mode while executing the process. Similarly, `sys` time indicates the amount

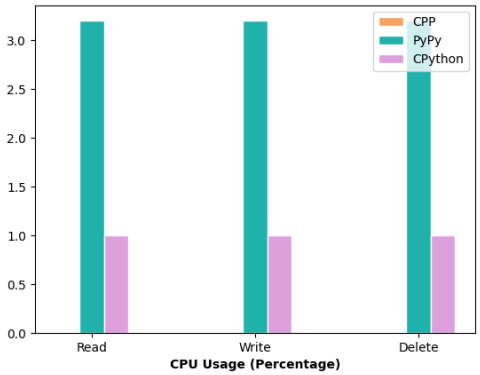
of CPU time spent in kernel-mode while executing the process. In our experiments, `PyPy` performs slower than CPython and C++. The difference in execution time is relatively small because the client program spends most of the time on network communication. However, we speculate that the overhead introduced by PyPy is due to the JIT compilation that does not pay off for this particular application. It might also be possible to re-structure the client code in such a way that makes the JIT compiler more effective.



As mentioned above, the left chart clearly shows that the client program has a significant network communication overhead and regularly waits for network interactions without using much CPU time.

4.1.3 Profiling - CPU Usage

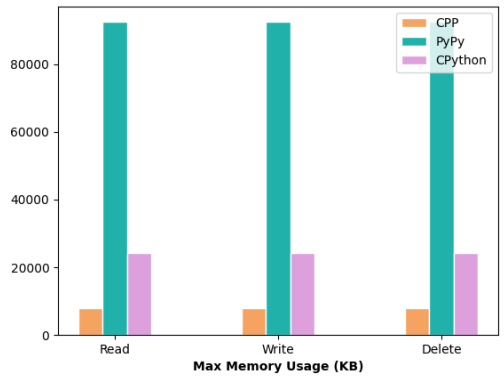
The client program does not have a heavy computational overhead, and the chart below presents its CPU usage. The client CPU usage averages to 0% for the compiled C++ program, while for PyPy, it is moderately higher, at 3%.

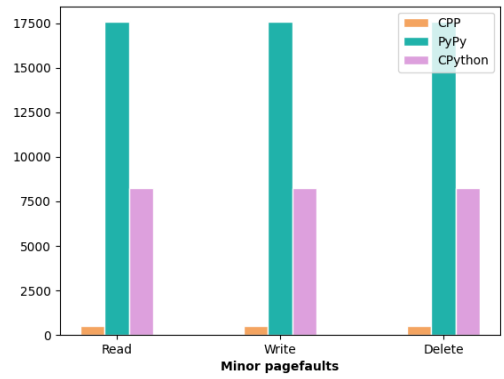


4.1.4 Profiling - Memory Usage

As shown in the memory usage charts below, PyPy uses more memory than CPython and C++. We speculate that this is due to two main reasons. First, CPython uses a garbage collector based on reference counting, and in C++ the objects are manually deallocated. In contrast, PyPy uses a hybrid garbage collection algorithm–it uses a nursery for the young objects and mark-and-sweep for the old objects.

The chart on the right presents the number of minor page faults. We speculate that the number of minor page faults in CPython is lower than PyPy because the reference counting GC algorithm can preserve the locality of the object references in memory.





5 Conclusion

In this project, we studied the main challenges of distributed databases in the context of IoT systems, and we designed and implemented a simple distributed database. Since performance is essential in IoT, we implemented our project in both C++ and Python and compared their resulting performance. Our experiments show that the optimal choice of a runtime system heavily depends on the type of application. For example, for our client implementation, CPython–an interpreter for the Python language–outperforms PyPy–a just-in-time compiler.

References

[1] C++ libbcrypt library. https://github.com/trusch/libbcrypt.

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

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

[4] Pypy documentation. https://doc.pypy.org/en/latest/.

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