JCRINN Journal of Computing Research and Innovation e-ISSN : 2600-8793

Available online at
https://jcrinn.com/
https://crinn.conferencehunter.com/

Journal of Computing Research and Innovation 11(1) 2026

Journal of Computing Research and Innovation

Constructing Realtime Simulations Engine: Breaking Shared Memory Limits in High-Level Scripting Languages

Tengku Ezharuddin Tengku Bidin[1*], Ahmad Azlan Mohd Yasir[2], Nur Ain Shabiha Noor Hisyam[3], Nor Farhana Zulkifli[4], Arina Sauki[5]

[1,2,3,4]Faazmiar Technology Sdn Bhd, 55100 Kuala Lumpur, Malaysia.
[5]Department of Oil and Gas Engineering, Faculty of Chemical Engineering, Universiti Teknologi MARA, 40450 Shah Alam, Selangor, Malaysia.

ARTICLE INFO

Article history:
Received 11 September 2025
Revised 6 November 2025
Accepted 10 December 2025
Online first
Published 1 March 2026

Keywords:
Distributed Systems
Engineering Simulation
Inter-process Communication
Multiprocessing
Python
Shared Memory

DOI:
10.24191/jcrinn.v11i1.617

ABSTRACT

In critical applications, particularly in areas of Realtime Simulations, usages of Unix-based Shared Memory (Shared Memory) concepts help greatly in managing the system's resources, to be called into play at any moment by the other subsystems. The fact that it is memory-based makes inter-process transactions very fast, as befitting the requirements of critical systems. Coupled with some semi-low-level compiled languages like C/C++ that incorporates the Unix Shared Memory easily, Shared Memories formed the very basis of most Realtime Simulations Engines. In modern times however, both the computer Operating System (OS) as well as programming languages have evolved: the former to cater more for distributed processes and global internet communications, while the latter into more rapid proto-typing as well as very high-level hierarchy in the man-machine interface, which resulted in many, many new languages being used, the most popular of which would be Python. In the OS side, issues about integrity, safety and privacy drive OS manufacturers to do away with many would-be leeways in which security issues (or lack thereof) could creep in, and the usage of Shared Memories would be the primeval talking point. In short, while operating systems like Windows and macOS still support shared memory mechanisms (e.g., POSIX SHM, memory-mapped files), these are limited to processes on a single host. Creating true shared-memory spaces across multiple networked systems is no longer natively supported. In this paper, we investigate the constraints of Python's Shared Memory model in distributed contexts and, realizing of its futility in fully utilizing it, propose four practical alternatives to overcome the hurdles, eventually opting one of them after being convinced that it is the nearest in behaviour to the Shared Memory model. This work serves as a guide for developers and engineers facing similar constraints in using Python for distributed high-frequency data sharing.

[1*] Corresponding author. *E-mail address*: ezharuddin@faazmiar.com

1. INTRODUCTION

Real-time Drilling Simulations are vital tools in oil and gas operations, enabling operators to anticipate challenges, reduce non-productive time and train personnel under dynamic and realistic operational conditions. These simulations must handle continuously updating parameters such as Rate of Penetration (ROP), Torque, Weight on Bit (WOB) and Pump Flow Rate with low latency and high reliability. Achieving this often requires processes running in parallel across different machines to access and act on a common data set.

Shared Memory (SHM) is a key enabler of high-performance inter-process communication (IPC), as it allows processes to read and write to a common memory space without the overhead of data serialization or network transfer (Bershad et al., 1991). In languages like C, shared memory can be configured for either local or networked environments depending on the system-level APIs used. Python, however, implements shared memory through its 'multiprocessing' module, which restricts access to processes on the same physical machine only.

This architectural limitation makes Python less suitable for simulations in systems that span multiple machines (Moritz et al., 2018), like in distributed control systems or networked simulators. However, certain developments requirements necessitate using Python due to its ease of use, richness in supplementary libraries, abundance of global support and presence of widespread user community and most of all its ability to do rapid prototyping.

However, it becomes apparent that attempts to use SHMs across machines using Python will fail not only due to the current OS local memory boundary enforcement but also, albeit using Linux or other Unix-based OS's, due to the inherent GIL (Global Interpreter Lock) inherent in Python's architecture. As a result, developers seeking to use Python in distributed real-time environments must explore alternative methods for efficient data exchange. This paper explores four such alternatives:

(i) Moving SHM logic to a centralized server with UDP for distribution

(ii) Using memory-mapped files on a shared network drive

(iii) Developing a custom UDP-based telemetry system

(iv) Implementing a volatile, memory-only networked layer

These approaches are analysed for their practicality, ease of implementation, and performance characteristics in real-world simulation scenarios.

2. METHODOLOGY

2.1 System architecture overview

Python's built-in shared memory capabilities, introduced in version 3.8 via the multiprocessing. shared_memory() module, are restricted to inter-process communication on the same physical machine (Python Software Foundation, 2023). When initially investigating this, we unfortunately overlooked this prerequisite and spent approximately 2 months constructing SHMs for our project using Python on Windows. This of course would have been alright if we had intended that all systems reside in the same server and they each fetch from this singular central machine, all the data needed for their dynamic variables. Unfortunately, what we planned and designed were quite different from what the current capabilities of the tools are. We had wanted that subsystems exist on their own but different autonomous machines while all the while communicating with the central server regarding all their data needs: storing, passing, fetching and replaying. At the same time the rest of the sub-systems are developed, which are displayed in the diagram as shown in Fig.1.

https://doi.org/10.24191/jcrinn.v11i1.617

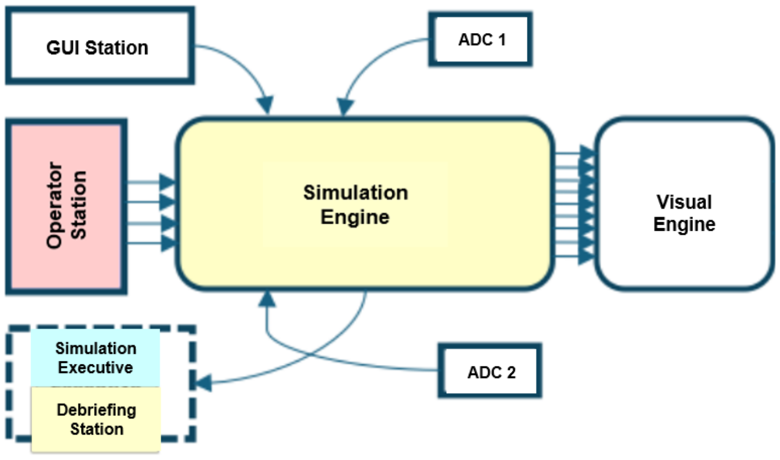

Fig. 1. Our simulator architecture: Bird's eye view

Fig. 1 illustrates a central server containing the engine (highlighted in yellow), with all other subsystems interacting with it to fulfil their data requirements. These subsystems include:

(i) The Operator Station (OSt). This is where all the initializations of the Simulation is carried on. The data and related well configuration data are pre-pared, defined and all the parameters needed for successful simulation are pooled together.

(ii) The Graphical User Interface (GUI) sub-system. There may be more than one subsystem in this case. This is the main conduit between the operator, to visualize current parameter values that exist in the initializations systems. The operator will not start any activities unless and until the GUI system conveys graphically that all parameters are such that it is safe to do so.

(iii) ADC1/ADC2 Sub-system. The ADCs are Analog-to-Digital Converters, controller subsystems that translate all analog signals to digital ones so that the user may perform all instructions he has via knobs and buttons that are given to him as options. They are run based on the RP2350 Pi Pico microcontrollers with 3.3 Volt DC voltages changed to 16 bit digital signals.

(iv) The Visual System. This is where all visualizations in 3D are done: run, calculated and displayed. All 3D algorithms used are in the Open System OpenGL, written in Python and managed by the Pyglet library. The system has components consisting of:

- An actual Graphics GPU-based Engine calculating millions of vertices, choosing any three of these vertices to form a triangle, forming a wireframe joining all these triangles, pasting texture files onto this wireframe with all within a sixtieth of a second. BEFORE the next sixtieth of a second comes, the engine will erase every single point, recalculate back the new positions of these vertices (because they have moved) and repeat the process back all over again, producing a smooth real time motion with refresh rate of 60 frames per second (fps).
- The operator is in controls on all happenings.

(v) The Simulations Engine. This is where the main actions take place, although not as visually manifested compared to the Visual Engine. This is where the entire simulations are weaved together from various subsystems. The SHM system, if ever at all, resides here and this is also where the User Datagram Protocol (UDP) server resides. In this paper we will concentrate on happenings in this system as far as the SHM goes.

(vi) BONUS/OPTIONS: The Simulation Executive. Though usually not part of any simulation architecture, we provide this system as a way of knowing that invariably the Simulations Engine is currently working. Because there is no way to determine whether data is currently exchanged properly, or whether they are created or even exist at all, we got to have some MEANS to "peek" into the engine to determine that yes: the engine is running and that they are churning the required numbers and data. Succinctly it also a means to check if any of these data is calculated correctly at all.

Currently, we named this newly developed system of ours as "The Peeker".

2.2 Problem context and constraints

The problem really comes in trying to fetch data from the central server in real time from the other different subsystems. In actuality, we had developed "The Peeker" very early on because we would like to see if we can read all the data that have been sent by our microcontrollers and all the other subsystems successfully. It is evident that there is no problem with reading data that arrives at the server at all (and thus that became the reason where we continued its development unsuspectingly) because as far as sending data to the central server is concerned, it was a one-way route! The data arrival was clearly observable. At this point, we had not yet had the necessity if the server were to share what is in its memory, to external subsystems. It was only when we installed the Peeker external to the server, that we began to face problems: it failed to function. This is where the problem manifests:

The multiprocessing.shared_memory() which is part of the built-in IPC module in Python is only designed for inter-process communication within the same machine only. This is unfortunate because it uses the same terminologies, as what we have experienced before in C/C++ running on UNIX: get shared memory (shmget()) and attach shared memory (shmat())(Steven & Rago, 2013). The fundamental difference is, despite using same concept terminologies, Python's multiprocessing.shared_memory() uses the operating system's local shared memory facilities, which are not networked (Eli, 2012).

We were therefore required to abandon creating and using shared memories via Python, if we are to use modern Operating Systems, and have to find alternative means with which data can be manipulated, still using Python, without sacrificing ease and speed.

2.3 Alternative approaches

Several alternatives were considered as follows:

*Option 1: Stop cross-machine communications and move SHM logic to server-side only (*Kerrisk, 2010*).*

In this approach, all shared memory handling is centralized. The server creates and writes to the shared memory, typically after reading data from microcontrollers or simulation sources. The server then sends the relevant values to remote readers via UDP. These readers no longer access shared memory directly, simplifying their role to display or process received data. In our case, the Peeker cannot exist outside of the server, which defeats the purpose. Also, whatever other subsystems we have, they will have to abandon sharing information with the server: they wait for the server to send data and transmit data to the server in return, without true data sharing.

This approach does not address our requirements. Basically, what it is simply saying is that "forget about creating systems that share memories with the server", which is contrary to our wish because remember that our requirement is to have an external reader to read the goings-on of the Engine's SHM. And by external we mean residing elsewhere outside the server. In fact, it could just be in the form of a simple iPad that one plugs in temporarily externally and takes out as soon as one is satisfied that everything is okay. Therefore, this option has got to be rejected.

*Option 2: Implement a custom UDP protocol (*Steven et al., 1998*)*

Now take note that in a system like ours, the server periodically sends key simulation values using UDP packets. And this means that the clients receive and display or store the data without interacting with shared memory. Which means one thing: either totally bypass the SHM or redefine your UDP packets so that it can minimize dependencies and reduces synchronization overhead, meaning SHM considerations. We then have a simple, fast, and well-suited for one-way telemetry scenarios.

In essence, this option is the most dangerous of all for us. Modifying UDP definitions to suit specific needs will make codes extremely non-portable, non-standardized and very risky: external support would be unavailable if issues arise. The only person to understand your modifications is you yourself, and when you get into problems, external helpers will not be able to give aids.

But in the real sense, one doesn't really modify the UDP itself but rather create our own data format/protocol on top of UDP. But still, it doesn't really solve the problem because customizing the UDP still does not give us cross-machine SHM access. It only gives reliable data transfer and a defined packet format flexibility. Therefore, this option too, is shunned.

*Option 3: Use memory-mapped files over a network file system (*McKusick et al., 2014*)*

In principle, although this is not what we are looking for, the concept is quite simple: A shared network drive (e.g., NFS) can host memory-mapped files accessible by multiple machines. Using Python's mmap() module, both the server and client processes map to the same file, effectively enabling indirect shared memory. While latency is higher than local memory, this method supports multiple readers without complex synchronization. In short, we are not building shared memories. We are building disk-based databases where all sub-systems can share and access, and we term this as a network file system. This is not what we really want.

*Option 4: Using volatile in-memory database (*Carlson, 2013; Islam et al., 2020; Leite & Tretyak, 2015; Rockenbach et al., 2017*)*

For the next option, we didn't really know of its real power and significance until we ran it up. When we did, it quickly dawned upon us: THIS is Shared Memory, although it never really says it is. And what is amazing is: irrespective of what Operating System one uses, it bypasses all the constraints of inter-machine memory sharing, as though it assumes that we are running just like in Unix, whereas we are not. The only caveat is being in-memory only system, it is volatile. If the machine is shut off, then all data will be lost forever.

There are many volatile memory-based databases available, but the one we evaluated was Redis, developed by Redis Ltd. In their website they advertise themselves as "The World's Fastest Data Platform" and testing confirmed their claims.

Redis is a high-performance in-memory key-value store that supports networked access. In this setup, the server writes sensor or simulation values to Redis, and remote clients fetch updates in near-real-time. Redis offers durability, high read speed, and scalability, making it suitable for systems requiring multiple subscribers.

3. RESULTS AND DISCUSSIONS

3.1 Suitability for real-time use

Among the four methods mentioned above, the Redis memory-based volatile memory database offers the best performance for real-time scenarios, and, with the usage of UDP, ideal for continuously streaming simulation values, especially when occasional packet loss is tolerable. It is also capable of near real-time performance, although introducing slight latency due to its network and data serialization over-head. Overall, this approach suits our requirements well.

Redis is optimized for memory operations, thus ensuring blazing speed (Liu & Yuan, 2019). It is also network-native: this means it is also built for distributed access. A notable advantage is its atomic operations: all the complexities about semaphores, locking/unlocking that one has to be aware of all the time when dealing with Unix SHMs, are all built-in, automatically done for you. Although volatile, this does not mean that Redis cannot survive restarts: it has that persistence capabilities. It should be noted here that Redis is written in C.

3.2 Python implementation

Python already has native libraries for socket programming and coupling this with Redis makes inter-system communications a breeze. Redis is well documented although its power is such that we utilized only a fraction of its capabilities. In essence, a Redis server can start off with the simplest:

```
# Server side
import redis
r = redis.Redis(host='server_ip', port=6379)
r.hset('sensor_data', mapping={
   'cell0': val1,
   'cell1': val2,
   'cell2': val3,
   'cell3': val4
})
```

This completes the implementation. All the complexities of shared memory multiprocessing are pooled in “import redis” and camouflaged therein. One does not have to manually deal with it. It just needs to invoke redis. Redis to kick off everything, and the only thing one needs to supply are the “map_ID’s” – (cell0’s and so on), together with the corresponding identifier names of the values of the data. Using these names, the data can be collected later and ready to be put into Redis’ database. On the Client/Reader side however, which exists on another machine, the invoking is much simpler:

```
# Reader side
r = redis.Redis(host='server_ip', port=6379)
values=r.hmget('sensor_data','cell0','cell1','cell2','cell3')
```

One needs to know only the IP address of the Server, and one must mention the same ID_MAPping and names. The implementation is complete.

3.3 Cross-platform and communications compatibility

Redis is highly portable across platforms, supported on Linux, macOS, and Windows. Network-based memory-mapped files are more dependent on filesystem compatibility (e.g., NFS on Linux or SMB on Windows), which may limit flexibility in mixed-OS environments. The server-side shared memory approach assumes clients do not need direct memory access, which improves compatibility as only standard networking is required.

UDP-based methods have the lowest communication overhead since they do not require connection setup, acknowledgments, or protocol handshaking. This makes them ideal for systems where bandwidth is not a limiting factor. Redis introduces some overhead due to key management and persistence features, but it remains efficient for high-read, moderate-write workloads.

3.4 Latency tests and findings

Latency tests were conducted to compare UDP message transmission and retrieval against Redis performance. The procedure involved sending segmented numerical data from a client to a server and awaiting its return along with receipt acknowledgments. The objective was to quantify the overhead introduced by Redis operations and to assess whether this overhead remained within acceptable limits for real-time applications. Fig. 2 presents a graphical comparison of the results.

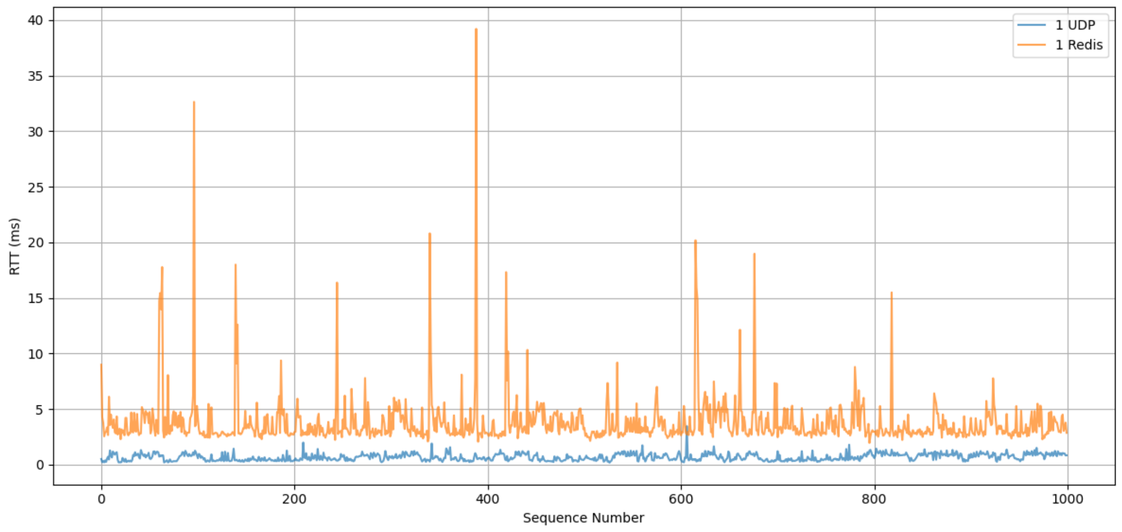

Fig. 2. UDP vs Redis latency comparison

The latency comparison data are presented in Table 2. The results align with expectations: pure UDP demonstrates negligible overhead in handling incoming and outgoing data, making direct comparisons challenging. The overall mean latency for UDP is approximately 0.67 ms, with a median of 0.61 ms, whereas the Redis Publisher records a mean of 3.1 ms and a median of 3.8 ms. Variability is minimal for UDP, with a spread of 0.32 ms, compared to 2.4 ms for Redis. Notably, both systems exhibited zero data loss. While this reliability is characteristic of Redis (for reasons elaborated later), it is unexpected in the case of UDP, where packet loss is typically common, particularly in wide-area network environments.

Table 1. Latency comparison table (per client)

Client	Protocol	Mean	Median	Min	Max	Std	JitterMean	JitterMax	PacketLoss (%)
1	UDP	0.67	0.612	0.171	3.46	0.317	0.212	2.874	0.00%
1	Redis	3.788	3.092	1.948	39.198	2.428	1.205	35.498	0.00%

The results indicate that Redis exhibits approximately five times higher network latency than UDP; however, in the millisecond range, this increase remains acceptable. The absence of packet loss in Redis provides insight into this behaviour where Redis relies on a TCP-based communication protocol, which ensures reliable data delivery. The design of TCP guarantees that all transmitted data are received intact, prioritizing delivery assurance even at the cost of additional delay. In contrast, UDP operates as a 'fire-and-forget' protocol, transmitting data without verifying its arrival. Thus, despite the use of UDP sockets for both client and server in the test configuration, Redis, in its default mode, employs TCP to guarantee safe and complete data transfer.

A latency of approximately 30–40 ms remains acceptable at millisecond fidelity and is likely imperceptible at sub-second scales, as is the case in the simulators under consideration. However, for applications requiring sub-millisecond precision, such delays become significant. It is noteworthy that Redis version 8 has been announced with support for sub-millisecond fidelity, addressing such requirements.

An additional observation concerns the jitter characteristics identified in the experiments. While UDP exhibited virtually no jitter, Redis demonstrated both frequent and higher-magnitude spikes. These spikes result in occasional delays in message arrival; however, all data are ultimately delivered without loss. It is important to distinguish between packet loss, which pertains to whether a message arrives, and jitter, which pertains to the timing of its arrival. For instance, in a sensor system updating data every 1–2 seconds, a jitter of approximately 35 ms represents only 2–3% of the cycle, rendering it technically negligible.

3.4.1 Alternatives to shared memory

There are various alternatives to Shared Memory other than Redis available in the market as outlined in Table 2. However, we did not have the opportunity to test all of them yet.

Table 2. Shared Memory Alternatives

Category	Examples	Strengths	Weakness	Authors/Year
Message Queues / Brokers	RabbitMQ, Kafka, ZeroMQ, NATS	Excellent for streaming, ordered delivery, pub/sub models	More overhead for small, frequent updates; often need consumers to ACK, not ideal for continuously updated shared variables	Heroux et al. (2020); Kumar (2013)
Network File–Backed Memory	NFS + mmap, Memcached	Simple, file-like interface; memcached is very fast for key-value caching	NFS adds filesystem latency; memcached is volatile and not designed for persistence; no pub/sub or atomic ops	Garfinke (2010); Turkstra (2013)
Shared Databases	PostgreSQL (LISTEN/NOTIFY), SQLite over network FS	Structured queries, durability	Heavier, higher latency, harder to achieve low-jitter real-time behavior	Heroux et al. (2020)
Direct Socket/IPC Solutions	TCP/UDP sockets, gRPC, custom serialization	Full control, minimal dependencies	Must implement own synchronization, consistency, fault-tolerance, etc.	(Garfinke, 2010; Couvée et al., 2023)
Other In-Memory Data Grids	Hazelcast, Aerospike, Redis Cluster (self-mention)	Distributed, scalable, HA	Usually Java-heavy (Hazelcast), harder setup	Heroux et al. (2020); Couvée et al. (2023)

Several approaches exist for achieving data sharing across distributed systems. Message-oriented middleware such as RabbitMQ or Apache Kafka provides robust event streaming and guaranteed delivery but incur higher latency and require consumers to process messages sequentially. Key-value caches such as Memcached offer low-latency lookups but lack built-in persistence and atomic update semantics. Traditional databases (e.g., PostgreSQL) provide transactional guarantees but are significantly slower for high-frequency updates and require complex schema management. Custom TCP/UDP socket implementations provide maximum control but demand careful handling of concurrency, fault recovery, and consistency protocols. In contrast, Redis offers a lightweight, in-memory key-value store with built-in support for atomic operations, publish/subscribe messaging, optional durability, and rich data structures, making it particularly suitable for near–real-time distributed applications.

4. CONCLUSION AND RECOMMENDATIONS

Python's multiprocessing.shared_memory() module provides efficient local IPC, but its design limits its applicability in distributed systems. This limitation is critical in simulation environments where processes span across multiple machines. Through this study, we examined and compared four practical alternatives i.e. server-side UDP, networked memory-mapped files, Redis, and custom UDP protocols. Each method addresses the core challenge of inter-machine memory sharing in a unique way, offering trade-offs between simplicity, speed, and robustness.

Our experiments show that while Redis Pub/Sub can simplify system integration, it adds latency overhead and jitter compared to raw UDP. For real-time control or high-frequency telemetry, UDP remains superior. Nevertheless, Redis is better suited when message durability, flexibility, or multi-consumer patterns outweigh strict latency demands.

Here one sees the latency comparison between raw UDP and Redis. As expected, UDP is extremely fast and stable which almost like the Shared Memory of the old Unix days. Redis adds a few milliseconds of overhead and some jitter, but it remains well within workable bounds. In return, we gain safety, portability, and scalability in making Redis a practical modern stand-in for the glory of Unix Shared Memory, without its risks. And this matters because in Simulations, what we need is not just raw speed, but safe, predictable, and portable data exchange and Redis gives us exactly that, without sacrificing the responsiveness that real-time simulation demands. Simulation developers should carefully assess their system requirements before selecting an approach. Where simplicity and low latency are priorities, UDP-based methods are favourable. For durability and concurrent access, Redis offers better guarantees. It enables Python to remain a viable option even in distributed, real-time simulation applications.

5. ACKNOWLEDGMENTS/FUNDING

The authors gratefully acknowledge the financial and technical support provided by Faazmiar Technology Sdn. Bhd., which made this project possible.

6. CONFLICTS OF INTEREST STATEMENT

The authors agree that this research was conducted in the absence of any self-benefits, commercial or financial conflicts and declare the absence of conflicting interests with the funders.

7. AUTHORS' CONTRIBUTIONS

Tengku Ezharuddin Tengku Bidin: Conceptualisation, methodology; **Ahmad Azlan Mohd Yasir**: formal analysis, investigation, and writing; **Nur Ain Shabiha Noor Hisyam**: formal analysis and investigation; **Nor Farhana Zulkifli**: formal analysis and investigation, **Arina Sauki**: writing-review and editing, and validation.

8. REFERENCES

Bershad, B. N., Anderson, T. E., Lazowska, E. D., & Levy, H. M. (1991). User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, *9*(2), 175–198. https://doi.org/10.1145/103720.114701

Carlson, J. L. (2013). *Redis in action*. Manning Publications.

Couvée, P., Lemarinier, P., Cedeyn, A., Agosta, G., Cattaneo, D., Fornaciari, W., Zaccaria, V., Bartolini, A., Torquati, M., Aldinucci, M., Colonnelli, I., Martinelli, A. R., Guimarães, F., Mohr, B., Goglin, B., Clauss, C., Krempel, S., Moschny, T., Pickartz, S., … Vysocký, O. (2023). European pilot for exascale - Software specification (101033975).

Eli, B. (2012, January 24). *Distributed computing in Python with multiprocessing*. Eli Bendersky's Website. https://eli.thegreenplace.net/2012/01/24/distributed-computing-in-python-with-multiprocessing

Garfinke, T. (2010). *Paradigms for virtualization based host security* [Doctoral dissertation, Stanford University]. Stanford University Repository.

Heroux, M. A., McInnes, L. C., Thakur, R., Vetter, J. S., Li, S., Ahrens, J., Munson, T., & Mohror, K. (2020). *ECP software technology capability assessment report – public (ECP-RPT-ST)*. Exascale Computing Project. https://doi.org/10.2172/1760096

Islam, A. A. R., Narayanan, A., York, C., & Dai, D. (2020). A performance study of Optane persistent memory: From indexing data structures' perspective. In *36th International Conference on Massive Storage Systems and Technology (MSST 2020)*.

Kerrisk, M. (2010). *The Linux programming interface: A Linux and UNIX system programming handbook*. No Starch Press.

Kumar, A. S. (2013). *Virtualizing Intelligent River: A comparative study of alternative virtualization technologies* [Master's thesis, Clemson University]. Clemson University TigerPrints.

Leite, L., & Tretyak, A. (2015). *In-memory databases and Redis* (Technical Report).

Liu, Q., & Yuan, H. (2019). A high performance memory key-value database based on Redis. *Journal of Computers*, *14*(3), 170–183. https://doi.org/10.17706/jcp.14.3.170-183

McKusick, M. K., Neville-Neil, G. V., & Watson, R. N. M. (2014). Design and implementation of the FreeBSD operating system (2nd ed.). Addison-Wesley Professional.

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., & Stoica, I. (2018). Ray: A distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)* (pp. 561–577).

Python Software Foundation. (2023). *Multiprocessing — Process-based parallelism*. Python 3.12 Documentation. https://docs.python.org/3/library/multiprocessing.html

Rockenbach, D. A., Anderle, N., Griebler, D., & Souza, S. (2017). Estudo comparativo de banco de dados chave-valor com armazenamento em memória. In *13th Escola Regional de Banco de Dados (ERBD)* (pp. 67–76). http://www.upf.br/_uploads/Conteudo/erbd2017/anais_ERBD2017_final_pos.pdf

Steven, R., Ferner, B., & Rudoff, A. M. (1998). *UNIX network programming: The sockets networking API* (3rd ed., vol. 1). Addison-Wesley.

Steven, W. R., & Rago, S. A. (2013). *Advanced programming in the UNIX® environment* (3rd ed.). Addison-Wesley Professional.

Turkstra, J. A. (2013). *Metachory: An unprivileged OS kernel for general purpose distributed computing (Vol. 9)* [Doctoral dissertation, Purdue University]. Purdue e-Pubs.