GOLDEN RATIO OF DATA IN SUMMARY
VOLUME 6, ISSUE. 1 (2026)
Website: https://goldenratio.id/index.php/grdis

GOLDEN RATIO
ISSN [Online]: 2776-6411

DATA IN SUMMARY | ENGINEERING, MATHEMATICS AND STATISTICS

Implementation and Analysis of Distributed Cache Architecture Between Virtual Machines in VMware to Reduce Memory Access Latency

Jan Everhard Riwurohi[1], Muh. Syahrir[2], Muhammad Farid Muslich[3], Indra Nurman[4], A. Adriansyah[5]

[1,2,3,4,5] Department of Computer Science Study Program, Faculty of Information Technology, Universitas Budi Luhur, Jakarta, Indonesia. Email: yan.everhard@budiluhur.ac.id[1], muhsyahrir005@gmail.com[2], muhammadfarid2424@gmail.com[3], indra.nurman@gmail.com[4], riana4619@gmail.com[5]

ARTICLE HISTORY

Received: November 7, 2025
Revised: January 24, 2026
Accepted: January 27, 2026

DOI
https://doi.org/10.52970/grdis.v6i1.1838

ABSTRACT
Virtualization technology allows multiple virtual machines (VMs) to run on a single physical machine, improving efficiency and flexibility. However, virtualized systems often face performance problems such as high memory access latency and repeated data requests between VMs. To address this issue, this study implements a distributed caching system using Redis as an in-memory cache shared between virtual machines. The experiment was conducted on the VMware vSphere platform using two virtual machines: one VM acted as a Redis cache server, and the other as a client for testing. Both VMs were connected using a host-only network to ensure stable communication. Testing was performed in two scenarios: without cache and with Redis cache, each executed 10 times. The main metric measured was response time in seconds. The results show a clear performance improvement after using Redis. The average response time without cache was 0.0113 seconds, while with Redis cache it decreased to 0.00046 seconds. This indicates that Redis reduced memory access latency by approximately 97.6%. The system also remained stable during testing without any connection issues. In conclusion, implementing a distributed caching architecture using Redis effectively improves response time, reduces memory access latency, and enhances system performance in a VMware virtualized environment. This study can serve as a reference for developing more efficient and responsive virtualization systems in modern computing environments.

Keywords: Virtualization, Redis, Distributed Cache, VMware, In-Memory Cache, Latency.

I. Introduction

The development of virtualization technology has become one of the major milestones in the transformation of modern computing infrastructure. Through virtualization, hardware resources such as processors, memory, and storage can be used more efficiently by dividing one physical machine into multiple virtual machines (VMs). One widely used platform for such needs is VMware vCenter, which allows users to run multiple operating systems simultaneously in a single host environment. Although virtualization provides high flexibility and efficiency, there are still challenges in terms of system performance, especially in terms of memory access between virtual machines. When multiple VMs run in parallel, there is often a duplication of the data retrieval process from the main storage or memory. This condition leads to increased latency and decreased overall system performance, especially in scenarios that involve high or repetitive data access loads.

ISSN [Online]: 2776-6411

One approach that can be applied to overcome this problem is to build a distributed cache architecture between virtual machines. In this architecture, one VM acts as a server cache that stores frequently accessed data, while another VM acts as a client that retrieves data from the cache before accessing the primary source. With this cache layer in place, data access response times can be accelerated because data doesn't always have to be retrieved from slower secondary storage. In-memory caching technologies such as Redis are a relevant solution in this context. Redis can store data directly in memory (RAM) and supports distributed communication between systems. This makes Redis ideal for implementation in inter-VM caching simulations on the VMware vCenter platform. Thus, this study aims to analyze how the implementation of a distributed cache architecture between virtual machines can reduce memory access latency and improve system efficiency. Through simulations conducted with two virtual machines, one as a cache server and one as a client, this research will measure system performance with and without cache implementation. The results of this research are expected to contribute to the development of more efficient and responsive virtualization systems, especially in the context of computer architecture and distributed memory management.

II. Literature Review and Hypothesis Development

2.1. Virtualization Technology

Virtualization is a technology that enables the abstraction of physical computing resources into multiple virtual environments. Through virtualization, a single physical machine can host multiple virtual machines (VMs), each running its own operating system and applications independently. This approach improves hardware utilization, reduces operational costs, and enhances flexibility in managing computing resources. A hypervisor, also known as a virtual machine monitor (VMM), plays a crucial role in managing and allocating physical resources such as CPU, memory, storage, and network bandwidth to each VM.

2.2. VMware vCenter and Virtual Machine Management

VMware vCenter is a centralized management platform designed to control and monitor virtualized environments built on VMware infrastructure. It enables administrators to manage multiple hosts and virtual machines efficiently, providing features such as resource allocation, performance monitoring, and workload balancing. vCenter allows multiple VMs to operate concurrently on a single host, making it a widely adopted solution in enterprise virtualization environments. However, as the number of VMs increases, competition for shared resources—especially memory—can lead to performance degradation.

2.3. Memory Management in Virtualized Environments

Memory management is one of the most critical aspects of virtualization performance. In a virtualized system, multiple VMs share the same physical memory, which can result in memory contention and increased access latency. When different VMs request identical or similar data from storage or memory, redundant data retrieval processes may occur. This redundancy increases memory access time and places additional load on the storage subsystem, ultimately reducing overall system efficiency, particularly in workloads with frequent or repetitive data access.

2.4. Distributed Cache Architecture

A distributed cache architecture is a system design in which cached data is shared across multiple computing nodes to reduce repeated access to primary storage. In the context of virtualization, a distributed cache can be implemented between virtual machines, where one VM functions as a cache server and other VMs act as clients. Before accessing the main data source, client VMs query the cache server for frequently

ISSN [Online]: 2776-6411

used data. If the data is available in the cache, it can be retrieved more quickly, thereby reducing access latency and improving system responsiveness.

2.5. In-Memory Caching

In-memory caching is a technique that stores frequently accessed data directly in main memory (RAM) rather than on disk-based storage. Because memory access is significantly faster than disk access, in-memory caching can greatly reduce data retrieval time. This approach is especially beneficial in systems that require high performance and low latency, such as distributed and virtualized environments.

2.6. Redis as a Distributed Cache Solution

Redis (Remote Dictionary Server) is an open-source, in-memory data structure store commonly used as a cache, database, or message broker. Redis supports fast data access, key–value storage, and distributed communication between systems. Its ability to operate entirely in memory makes it suitable for reducing latency in data-intensive applications. In a virtualized environment, Redis can be deployed on a dedicated VM as a cache server, enabling other VMs to share cached data efficiently and minimize redundant memory or storage access.

2.7. Performance Metrics in Virtualized Systems

System performance in virtualized environments is commonly evaluated using metrics such as memory access latency, response time, throughput, and resource utilization. Latency refers to the time required to retrieve data, while efficiency reflects how well system resources are utilized. By comparing system performance with and without a distributed cache implementation, the effectiveness of caching mechanisms in reducing latency and improving overall efficiency can be quantitatively measured.

III. Research Method

This study adopts an applied research design with a quantitative experimental approach, aiming to evaluate the effectiveness of a distributed caching architecture in reducing memory access latency between virtual machines within a VMware-based virtualization environment. The experimental approach enables direct measurement of system performance differences under controlled conditions, thereby ensuring objective and reproducible results. The research was conducted at Budi Luhur University, South Jakarta, with all system simulations executed locally using VMware vSphere deployed on personal computer hardware. The experimental environment was designed to replicate a typical virtualized infrastructure where multiple virtual machines communicate over a shared virtualization layer. The population of this study encompasses all virtualization scenarios involving inter-VM communication and data access. The sample consists of two purpose-configured virtual machines tailored to the experimental requirements: DC1CACHE01, functioning as a distributed cache server running Redis, and DC1CLIENT01, acting as a client generating data requests. This configuration allows focused evaluation of cache performance in a simplified yet representative virtualization scenario.

Data collection was conducted using two primary methods: literature review and system simulation. The literature review provided theoretical foundations related to virtualization technologies, distributed caching mechanisms, and in-memory data stores such as Redis. The simulation method was employed to obtain empirical performance data by executing controlled experiments under two scenarios: a baseline system without caching and a system enhanced with Redis-based distributed caching. Data analysis was performed using a comparative performance evaluation approach. Key performance metrics included response time, CPU utilization, memory usage, and overall system efficiency. Measurements were obtained using standardized system tools such as command-line time utilities, HTTP request tools (e.g., curl), and custom Python scripts to ensure precise and repeatable data collection. The resulting data were analyzed quantitatively to assess performance improvements attributable to the implementation of the distributed cache. By combining experimental rigor with quantitative analysis, this methodology provides a reliable framework for evaluating the impact of Redis-based distributed caching on virtualized system performance and supports the generalizability of the findings to similar virtualization environments.

ISSN [Online]: 2776-6411

IV. Result and Discussion

This chapter presents the results of the implementation and testing of a distributed virtual machine caching architecture system in VMware using Redis as an in-memory cache system. The implementation is carried out in accordance with the design and methodology described in Chapter III to analyze the effect of cache implementation on memory access latency between virtual machines. The process of implementing the distributed cache system in this study is carried out in stages and systematically, starting from the installation of Redis Server on DC1CACHE01 VMs, the preparation of client VMs DC1CLIENT01, and system performance testing. The first step is to verify the network and operating system configurations on DC1CACHE01 VMs. Based on the result of the 'ip a' command, it can be seen that the ens160 network interface has obtained the IP address 10.100.27.171/24, which corresponds to the internal subnet 10.100.27.0/24. This ensures that the VM can communicate directly with the client VM without going through an external network. In addition, the result of the 'cat /etc/os-release' command indicates that the operating system used is Ubuntu 24.04 LTS (Noble Numbat), which was chosen for its stability and compatibility with the Redis package.

Once the initial verification is complete, a system update is performed using the 'sudo apt update' command to ensure that all packages and dependencies are on the latest version. This update is important to prevent errors during installation and maintain system stability. Redis Server is then installed using the command 'sudo apt install redis-server -y', and the Redis service is activated and its status verified via 'systemctl status redis-server'. The verification results show that Redis is running in an "active (running)" state and is ready to receive a connection through the default port 6379. Additional information, such as the primary PID, CPU usage, and memory allocation, is also displayed to ensure Redis runs stably as a background service. Redis version 7.0.15 is successfully installed, indicating that the system is ready to be used as the main cache node. The next step is to set up a DC1CLIENT01 client VM with IP 10.100.27.172, which is in the same subnet as the cache server. Network and operating system verification is done to ensure that connectivity between VMs can run smoothly. The operating system used is also Ubuntu 24.04 LTS, so compatibility and consistency of test results can be maintained. Once the system update is performed, redis-tools are installed to support testing the connection to the Redis Server. In order for Redis to be accessible from the client VM, the configuration of the Redis file is changed in the 'bind' section. By default, Redis only accepts local connections via '127.0.0.1' and '::1'. For this research, the configuration was changed to 'bind 0.0.0.0' or 'bind 10.100.27.171', so that Redis could accept connections from all network interfaces, including from client VMs within VMware's internal network. This configuration is critical because it allows Redis to function as a distributed cache that can be accessed by other virtual machines within a single host. With an isolated and stable network setup and a uniform operating system, communication between VMs can be carried out optimally. Redis Server, which has been configured and is running on DC1CACHE01, is now ready to receive data requests from DC1CLIENT01, so system performance testing can continue. This implementation is an important foundation for building an efficient cache architecture, which aims to reduce memory access latency between virtual machines and improve resource efficiency in VMware's virtualization environment.

4.1. Protected Mode Settings

The next part of the configuration relates to the security of Redis access, which is Protected Mode. By default, Redis will reject connections from outside the machine (remote host) if this parameter is active.

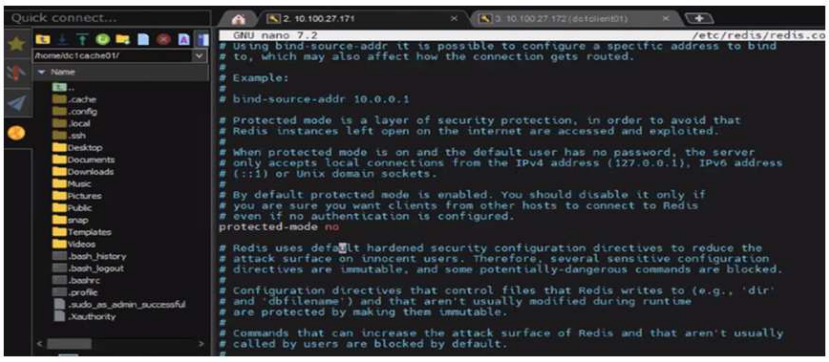


Figure 1. Protected Mode Settings

ISSN [Online]: 2776-6411

Configuration snippet shown in the image. If protected-mode is yes, then Redis can only be accessed from 127.0.0.1. If protected-mode is no, then Redis allows connections from other hosts on the same network.

In this study, because communication is only done in VMware's isolated network, it is safe to disable this feature. Thus, the client (DC1CLIENT01) can make a direct connection to the Redis Server (DC1CACHE01) without being rejected by Redis's internal security mechanisms.

4.2. Port and TCP Backlog Settings

The next configuration section sets up the communication ports and parallel connection capacity (TCP backlog) that Redis can handle.

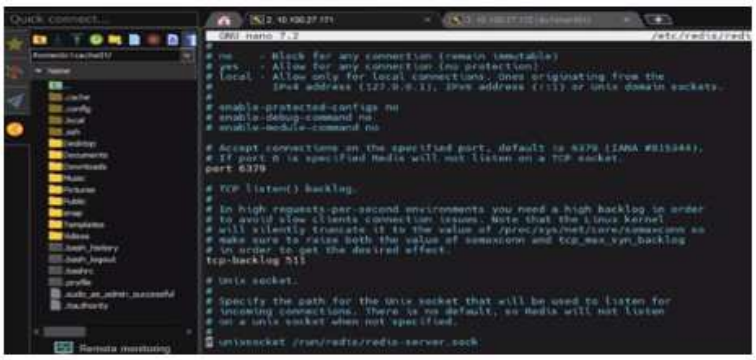


Figure 2.Port and TCP Backlog Settings

The relevant configuration content is:
-"port 6379
-tcp-backlog 511"
Explanation:

a. Port 6379 is the standard port used by Redis Server to receive connections from clients.
b. TCP-backlog 511 sets the maximum number of connection queues waiting to be received by the server. This value is recommended for environments with high connection rates to prevent connection rejections due to full queues.

With this setting, Redis is optimized to handle more requests in parallel without degrading performance. This is important because in a distributed cache architecture, Redis will often receive simultaneous requests from clients for GET and SET data operations.

4.3. Daemon Mode Settings and Process Supervision

At the end of the configuration, Redis is set to run as a daemon service (background service) and is overseen by the systemd unit.

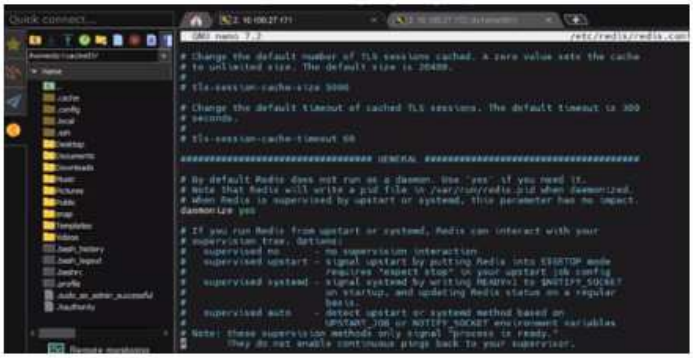


Figure 3. Daemon Mode Settings and Process Supervision

The contents of the configuration in the image above are:

ISSN [Online]: 2776-6411

-"Daemonize yes
-supervised systemd"
Explanation:

a. daemonize yes makes Redis run in the background, so there is no need to run it manually from the terminal every time the system boots.
b. supervised systemd allows Redis to integrate with the Ubuntu initialization system, so that the service can be monitored and controlled through commands:
-"sudo systemctl start redis-server
sudo systemctl enable redis-server
-sudo systemctl status redis-server"
c. This configuration ensures that Redis will automatically activate whenever the system is turned on, and administrators can easily monitor the status of the service.

Thus, Redis Server on DC1CACHE01 is not only functionally ready to use, but also fully integrated with the operating system and can run stably for distributed cache experiments.

4.4. Verify Redis Server Service Activation

Once the configuration process on the /etc/redis/redis.conf file is complete, the next step is to activate and verify the status of the Redis Server service on the system. This is done to ensure that Redis is running correctly using the new settings and is ready to receive connections from the client (DC1CLIENT01).

Figure 4.Verify the Activation of the Redis Server Service

The commands executed sequentially in figure 4.11 are as follows:
-sudo nano /etc/redis/redis.conf # open and edit the Redis configuration
-sudo systemctl restart redis-server #me-restart Redis service after configuration is changed
-sudo systemctl redis-server status # check the active status of Redis service
Figure Shows the output of the sudo systemctl command redis-server status after Redis Server is executed. From the results displayed, it can be explained as follows:

a. Service Status Line
"Active: active (running) since Sun 2025-10-26 17:57:25 WIB; 10 years ago"
Indicates that the Redis service has been successfully run and is running. This is a key indicator that the Redis configuration has been implemented correctly and that the service is running without errors.

1) Service Information
"Loaded:loaded(/usr/lib/systemd/system/redis-server.service; Enabled; preset: enabled)"
Indicates that the Redis service file has been properly loaded by systemd and set to be automatically activated whenever the system is booted (status enabled).

2) Redis Additional Information
"Status: "Ready to accept connections"

ISSN [Online]: 2776-6411

This message indicates that Redis is ready to receive a connection from the client via TCP port 6379 — according to the configuration that was done in the previous section.

3) Resource Information
"Memory: 3.3M (peak: 3.8M) CPU: 60ms"
This data shows that Redis runs very lightly with memory usage of about 3–4 MB, confirming the efficiency of Redis in-memory caching in the early stages of operation.

4) Process and Logging Information
"Main PID: 112726 (redis-server)
CGroup: /system.slice/redis-server.service
└─112726 /usr/bin/redis-server 0.0.0.0:6379"
Shows the primary process ID (PID) of Redis Server and the binding address 0.0.0.0:6379, which means Redis now listens for connections from the entire network interface, not just from localhost. This proves that the configuration changes to the bind and protected-mode parameters have been applied correctly.

4.5. Systemd Activity Log

"Oct 26 17:57:25 dc1cache01 systemd[1]: Starting redis-server.service...
Oct 26 17:57:25 dc1cache01 systemd[1]: Started redis-server.service."
Indicates that the Redis service has been successfully started without errors or warnings.
From the results of the verification, it can be concluded that Redis Server:

a. It has been successfully loaded and running with the modified configuration on the redis.conf file.
b. Listen for connections on port 6379 and can receive connections from the internal network (10.100.27.0/24).
c. Ready to use for the next stage, i.e. using redis-cli. Connectivity testing between Virtual Machines (DC1CLIENT01 → DC1CACHE01)

With the status of "Ready to accept connections", Redis on DC1CACHE01 has fully functioned as an active server cache that will be used for distributed caching experiments on VMware environments.

4.6. Verify Redis Access from Client

Once Redis Server is successfully configured and actively running on VM DC1CACHE01 (10.100.27.171), the next step is to verify access from the client side. This step aims to ensure that the network configuration and parameters that have been set on Redis Server (such as bind 0.0.0.0 and protected-mode no) actually allow the client to connect and communicate data directly between Virtual Machines within VMware's internal network. In the implemented distributed cache architecture, the DC1CLIENT01 VM (10.100.27.172) serves as the client node that will make a data request to Redis Server. This verification process includes two main stages, namely:

a. Installation of client-side components (Redis Tools) on client VMs, and
b. Testing the connectivity between the client and the server uses the redis-cli utility.

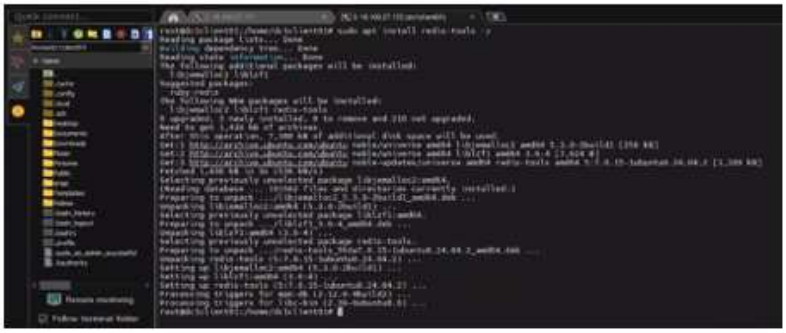


Figure 5.Redis Tools Installation Process on DC1CLIENT01

ISSN [Online]: 2776-6411

Figure Shows the installation process of Redis Tools on a VM DC1CLIENT01 using the command:
"sudo apt install redis-tools -y"

The command is used to install the redis-tools package, which is a set of utilities that provide tools to access, test, and monitor Redis Server. One of the important components of this package is the redis-cli, which is used to send commands and receive responses from Redis over a TCP/IP network connection. From the visible output, the installation process went smoothly - the system managed to download and install dependencies such as libjemalloc2 and libtinfo6 from the official Ubuntu (archive.ubuntu.com) repository. Setting up redis-tools (5:7.0.15-1ubuntu0.24.04.2). indicates that the installation was successful without any errors and that Redis Tools is ready to use. This is the first step of the verification process, as the redis-cli will be used to test the communication to the Redis Server in the next stage.

Figure 6. Verify Redis Connection from Client to Server

The image shows the connection test between the DC1CLIENT01 and DC1CACHE01 using the command:
"redis-cli -h 10.100.27.171 ping"

Explanation:

a. The -h parameter indicates the destination IP address, which is Redis Server on DC1CACHE01 (10.100.27.171).
b. Ping commands are a basic connectivity test that Redis uses to ensure that the server is active and can respond to commands from the client.

The results that appear are:
"PONG"

Indicates that Redis Server has successfully received and replied to requests from clients, proving that network connectivity between VMs is working properly. The PONG response is also an indicator that the bind and protected-mode configurations on Redis Server have been implemented correctly, so that the server can be accessed by other machines within the VMware subnet (10.100.27.0/24). Based on the results on it can be concluded that:

a. Redis Tools is successfully installed and serves as a client interface to access Redis Server.
b. Redis Servers in DC1CACHE01 can be reached and responded to well by DC1CLIENT01, as evidenced by the results of PONG.
c. Communication between Virtual Machines runs normally without network problems.

This stage proves that the Redis-based distributed caching system is functionally connected between Virtual Machines and is ready for use for performance testing and memory access latency analysis.

ISSN [Online]: 2776-6411

4.7. Running an external data source on the host

The program is created using the Python programming language by utilizing the Redis module as a temporary storage system (cache) and the Requests module to make data requests to the HTTP server. The purpose of this experiment was to analyze the comparison of response time between retrieving data derived from the Redis cache and retrieving data directly from an external source (server). The program defines the fetch_data() function that is in charge of retrieving data with the following steps:

a. Initialization of start time: The program records the start time of the process to calculate the total execution time.

b. Cache checking:

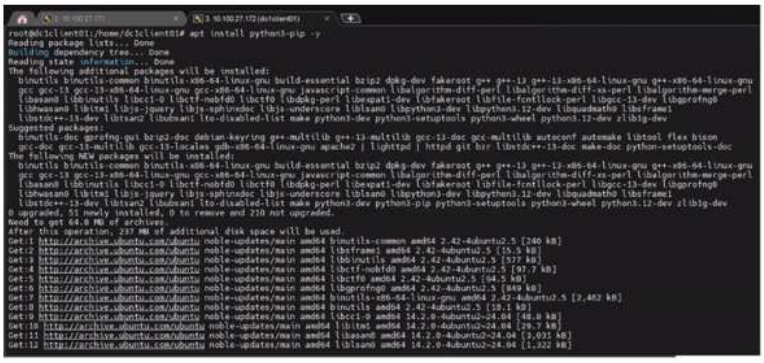
1) If the key named "data:external" already exists in Redis, then the data will be retrieved directly from the cache using r.get(cache_key).
2) The program then displays the message "Data from cache" as an indicator that the data is being retrieved from Redis.

c. Data retrieval from external sources:

1) If the key is not already in the cache, then the program will send an HTTP request using requests.get(url) to an external address.
2) The response data is then saved to Redis using r.set(cache_key, resp.text).
3) The program displays the message "Data fetched from source and cached" to indicate that the data has recently been cached.

d. Response time calculation:

1) After the data retrieval process is complete, the end time is recorded (end = time.time()).
2) The time difference (end - start) is calculated to determine the duration of the execution, and then displayed in the format "Response time: x.xxx seconds".
3) In this scenario, run the experiment 10 times with the command used:
times = []for i in range(10):t = fetch_data()
times.append(t)
print(f"Average response time: {sum(times)/len(times):.5f} seconds")



Scenario 2 Testing Images - Redis Cache

This step aims to set up a Python programming environment on the Ubuntu operating system that will be used to run the Redis performance testing program. The installation is done using the Advanced Package Tool (APT) on Ubuntu-based systems with the following command:

"apt install python3-pip -y"

On this view, it can be seen that the system:

a. Download the package from the official Ubuntu repository (archive.ubuntu.com).
b. Install various dependencies such as libexpat1-dev, python3-setuptools, and python3-wheel.

ISSN [Online]: 2776-6411

c. Displays additional disk space information (approximately 237 MB).

Scenario 2 Testing Images - Redis Cache

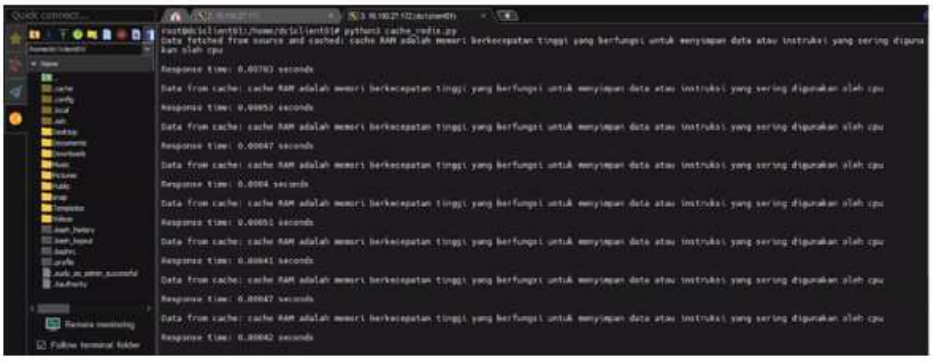
This stage aims to install the Redis library so that the Python program can communicate with the Redis server. This library provides an interface (API client) to perform operations such as storing, retrieving, and deleting data in Redis. Installation is carried out using the following command on the terminal:

"pip3 install redis --break-system-packages"

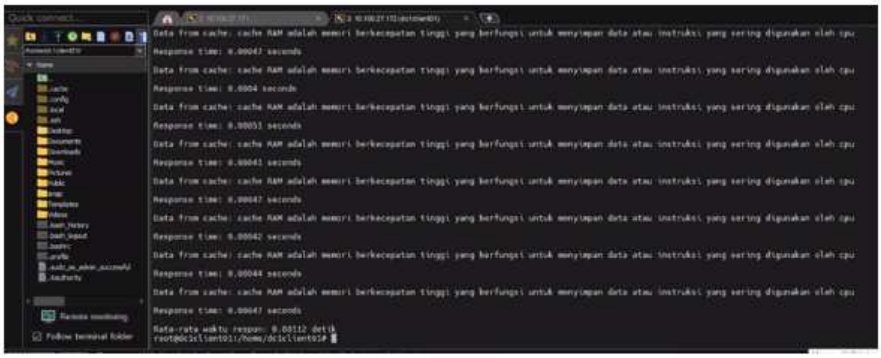
The --break-system-packages option is used because the installation is run as a root user outside of the virtual environment, so pip is allowed to modify the built-in Python package system. The installation process starts with:

a. Download the file redis-7.0.0-py3-none-any.whl from the Python Package Index (PyPI) repository.
b. Verifying package metadata (approx. 10 KB).
c. Install the Redis version 7.0.0 package to the system.
From the terminal display results, the system displays the message:
"Successfully installed redis-7.0.0"

With the successful installation of the Redis library, the Python environment is now ready to run a pre-built caching test program (cache_redis.py).



Scenario 2 Result Image - Redis Cache (1)



Scenario 2 Result Image - Redis Cache (2)

This stage aims to test the performance of the caching system using Redis in accelerating the response time of data collection.

ISSN [Online]: 2776-6411

The experiment was carried out by running the Python script cache_redis.py 10 iterations to compare data access times from external sources and from the Redis cache. Commands are executed using the following commands in the terminal:

"Python3 cache_redis.py"

At the beginning of execution, the program will:

a. Retrieve data from an external URL (http://10.100.27.92:8080/data.txt).
b. Saves the results to Redis as a cache.
c. On the next execution, the same data will be retrieved directly from Redis without accessing an external server.

4.8. Test Results

From the results of the above test, it can be observed that:

a. The first attempt takes about 0.007 seconds, as the data is retrieved from an external server (HTTP request).
b. The next experiment only takes between 0.0004 – 0.0005 seconds, as the data is retrieved from the Redis cache stored in memory (RAM). The average overall response time was 0.00112 seconds, indicating a very significant increase in access speed.

Table 1. Redis cache stored in memory (RAM)

Execution	Respone Time (detik)
1	0.00053
2	0.00047
3	0.0004
4	0.00051
5	0.00041
6	0.00047
7	0.00042
8	0.00042
9	0.00044
10	0.00047

From these results, it can be concluded that the use of Redis as a cache is able to speed up the data retrieval process by more than 90% compared to direct retrieval from external sources. This shows that caching mechanisms are very effective in systems that frequently access repetitive data, as they can:

1. Reduced response time,
2. Saves network load,
3. Improve overall system efficiency.

4.9. Comparison of Scenario 1 and 2 Results

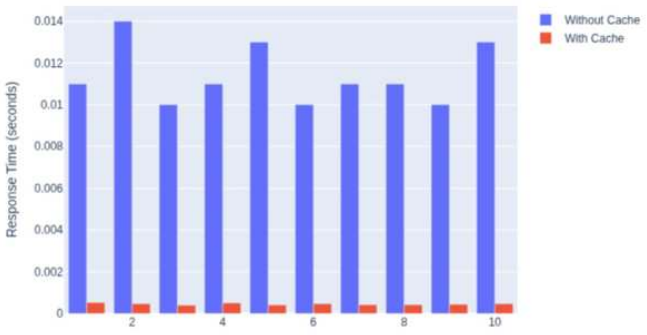


Figure 1. Results Comparison Images

ISSN [Online]: 2776-6411

The following figure shows the comparison of response times between the cached data retrieval process (blue color) and with Redis cache (red color) at ten program executions. The X-axis indicates the number of executions (1–10), while the Y-axis indicates the response time in seconds. From the graph, you can see a very striking difference between the two access methods with the following details.

a. No Cache

1) It is characterized by a blue rod.
2) This happens because every time a program is executed, the system has to make an HTTP request (request) to an external source to retrieve the data.
3) This process requires additional time for network connection and data parsing.

b. Cache Redis

1) Marked with a red rod.
2) In this method, the data that has been retrieved is temporarily stored in memory through Redis.
3) On-stream access directly retrieves data from the cache so there is no need to access the network, resulting in a much faster response time.

V. Conclusion

Based on the results of research and testing on the implementation of a distributed cache architecture between Virtual Machines using Redis on the VMware vCenter platform, it can be concluded that the distributed caching system was successfully implemented using two virtual machines, consisting of one VM as a Redis cache server and one VM as a tester client, where Redis stores frequently accessed data to achieve much faster response times compared to systems without caching. Performance testing conducted through 10 data request executions showed that the average response time without cache was 0.0113 seconds, while with Redis cache it decreased to 0.00046 seconds, indicating a performance improvement of approximately 97.6%, which proves that in-memory caching mechanisms like Redis are highly effective in reducing memory access latency between virtual machines. Additionally, Redis helps improve resource utilization efficiency by reducing the load on external data sources since repeated data can be retrieved directly from cache memory, thereby saving I/O resources and accelerating inter-VM communication. Throughout the testing process, the system demonstrated stable performance without connectivity disruptions or communication errors between nodes, and Redis consistently handled repetitive requests with low and stable response times, leading to the conclusion that implementing a distributed caching architecture using Redis in a VMware virtualization environment significantly enhances system efficiency, minimizes memory access latency, and accelerates data exchange between virtual machines.

References

Arifin, M., & Sari, R. (2022). "Analysis of HTTP Server Performance on Data Access Response Time Using Cache Test Methods." Journal of Information and Computer Technology, 8(2), 45–53.

Kurose, J. F., & Ross, K. W. (2021). Computer Networking: A Top-Down Approach (8th ed.). Pearson.

Nugroho, E. (2021). Computer Operating Systems and Networks. Andi Offset, Yogyakarta.

Python Software Foundation. (2024). http.server — HTTP servers. Accessed from https://docs.python.org/3/library/http.server.html

Redis Ltd. (2024). Redis Documentation. Accessed from https://redis.io/docs/

RFC 2616. (1999). Hypertext Transfer Protocol – HTTP/1.1. Internet Engineering Task Force (IETF).

Stallings, W. (2020). Data and Computer Communications (11th ed.). Pearson Education.

Tanenbaum, A. S., & Wetherall, D. (2019). Computer Networks (6th ed.). Pearson.

ISSN [Online]: 2776-6411

The curl Project. (2024). curl – Command line tool and library for transferring data with URLs. Accessed from https://curl.se/docs/

Ubuntu Documentation. (2024). Installing software packages using apt. Accessed from https://help.ubuntu.com/

VMware, Inc. (2024). vSphere Documentation. Accessed from https://docs.vmware.com/en/VMware-vSphere/

Welling, L., & Thomson, L. (2017). PHP and MySQL Web Development (5th ed.). Addison-Wesley Professional.