

**Network Security (CS3403)
Project Report**

**Secure Proxy Cache Implementation for Efficient Cache Storage
and Retrieval**

Submitted by

Sushan Rai

1RVU22CSE173

School of Computer Science

RV University

Submitted to

Prof. Chandramouleeswaran Sankaran

Professor of Practice

School of Computer Science

RV University

Submission date [06/04/2025]

Network Security (CS3403) Project Report

1. Abstract

This project focuses on building a secure and efficient proxy cache system using Node.js to enhance web performance and reduce content load times. The proxy server retrieves HTML content from a specified target URL and handles both cache hits and cache misses. A custom-built Redis-like cache server was implemented from scratch, designed to support basic `set`, `get`, and `delete` operations. This server maintains a persistent storage mechanism and periodically refreshes the data every 5 seconds to ensure consistency and reliability.

The core objective was to simulate a real-world caching system with added layers of security and persistence, going beyond in-memory storage. The proxy cache intercepts and stores responses from target URLs, significantly reducing repeated network overhead for subsequent requests. Extensive testing revealed a dramatic improvement in performance metrics, particularly in the Largest Contentful Paint (LCP), which was observed to be approximately 10 times faster than the standard uncached page load time. This project demonstrates how a custom caching layer can dramatically improve the efficiency of web content delivery while maintaining flexibility and data integrity.

2. Introduction

Project Background and Relevance: Network Security

In today's digital landscape, network security is critical to protect sensitive data and ensure the integrity of web applications. With increasing cyber threats and the demand for high-performance online services, the importance of secure caching mechanisms has grown. Proxy caches not only optimize performance by reducing redundant network requests but also act as an additional security layer by shielding backend servers from direct exposure to external attacks. This project leverages Node.js to implement a proxy cache that integrates a custom Redis-like server, ensuring that cached data is securely stored, persistently maintained, and regularly refreshed. By combining performance enhancement with security best practices, the system addresses both operational efficiency and potential vulnerabilities in data transmission.

Objectives: What the Project Aims to Accomplish

- **Develop a Secure Proxy Cache:** Build a proxy caching system using Node.js that effectively handles HTML data retrieval with secure data flow and management.
- **Implement a Custom Cache Server:** Create a Redis-like server capable of performing `set`, `get`, and `delete` operations with persistent storage that refreshes every 5 seconds to maintain up-to-date content.

Network Security (CS3403) Project Report

- **Enhance Performance Metrics:** Significantly reduce page load times, as evidenced by a 10-fold improvement in Largest Contentful Paint (LCP) compared to traditional loading methods.
- **Integrate Network Security Measures:** Ensure that the caching system mitigates common network security risks, such as unauthorized access and data tampering, thereby bolstering overall application security.

3. System Overview

System Architecture: Outline of Data Flow and Key Components

The system architecture consists of three major components working together to deliver secure and efficient caching: the **Frontend**, the **Proxy Cache Server**, and the **Redis-like Cache Server**. These components communicate over controlled and secured channels to ensure performance optimization and protection against network-level threats.

1. Frontend (Client Side):

- Sends HTTP requests to the Proxy Cache Server for web content.
- Communication is restricted via **HTTP whitelisting**, allowing only trusted DNS domains.
- Private IP ranges are explicitly blocked to prevent access to internal or unauthorized networks, mitigating SSRF (Server-Side Request Forgery) attacks.

2. Proxy Cache Server (Node.js):

- Acts as a middle layer between the client and the target URL.
- Checks the cache for data (cache hit); if not found (cache miss), it fetches the HTML content from the target URL.
- Stores the response in the Redis-like server for future use.
- Maintains security with **IP whitelisting**, DNS filtering, and sanitization to prevent injection and unauthorized access.
- Connects to the Redis-like server over a **TCP VPN tunnel**, securing communication between internal components and protecting against packet sniffing or tampering.

3. Redis-like Cache Server:

Network Security (CS3403) Project Report

- A custom implementation that mimics Redis operations (SET, GET, DELETE) and supports **persistent storage**.
- Periodically refreshes the cache (every 5 seconds) to ensure data remains up-to-date.
- Communicates only over the VPN to reduce exposure to external threats and ensure secure data exchange.

Data Flow Summary:

1. Frontend → Proxy (via HTTP, DNS-whitelisted only)
2. Proxy → Redis-like Server (via VPN-secured TCP)
3. Proxy → External URL (only when cache miss, filtered by DNS/IP rules)
4. Redis-like Server → Disk (persistent storage sync every 5 seconds)

This architecture ensures low-latency content delivery while embedding strong security practices such as network segmentation, controlled data access, and secure tunnelling—making it both efficient and resilient to common network threats.

4. Design:

Major Components and Their Interactions

1. Frontend (Client Interface):

- The user or client makes a request to load a webpage via a browser or HTTP client.
- This request is sent to the Proxy Cache Server using the **HTTP protocol**.
- The client is restricted by **DNS whitelisting**, ensuring only trusted domains are accessed.
- **Private IP address ranges are blocked** to prevent SSRF and unauthorized access.

2. Proxy Cache Server (Node.js + Express):

- **Handles client requests using Express**, which provides built-in mechanisms for parsing, encoding, and decoding HTML and JSON data in HTTP requests/responses.
- Upon receiving a request, it performs the following steps:
 - Checks for the requested URL in the cache by forming a **RESP (Redis Serialization Protocol)** command like GET target_url.
 - Sends the RESP command over a **TCP VPN tunnel** to the Redis-like server.
 - If a **cache hit** occurs, the cached HTML is returned and sent back to the client using HTTP with proper encoding via Express.

Network Security (CS3403) Project Report

- If a **cache miss** occurs:
 - The proxy fetches the HTML from the **external target URL** (only if it's DNS-whitelisted and not a private IP).
 - The HTML content is then stored in the cache using a RESP SET target_url html_data command.
 - The same HTML is sent to the client through Express.

3. Redis-like Cache Server (Custom Implementation):

- Listens on a TCP socket and communicates using a simplified version of **RESP** (Redis Serialization Protocol), enabling commands like SET, GET, and DELETE.
- **Parses RESP commands** received from the proxy server and executes the corresponding actions:
 - For SET, it stores the HTML data with the URL as the key.
 - For GET, it returns the associated HTML if available.
 - For DELETE, it removes entries as needed.
- Stores all data in an in-memory structure with **persistence support**, writing to disk every 5 seconds to maintain durability.
- Communicates only over the **VPN tunnel** to ensure secure and encrypted transmission.

Data Exchange Summary

Interaction	Protocol	Data Format	Security/Filtering
Frontend → Proxy Server	HTTP	Encoded HTML via Express	DNS Whitelist, IP Blacklist
Proxy → Redis-like Server	TCP (over VPN)	RESP (GET, SET)	VPN Tunneling, RESP Parsing
Proxy → External URL (on miss)	HTTPS	Raw HTML	DNS Whitelisting, IP Filtering
Redis-like Server → Disk	Internal Write	Plaintext or JSON	Auto-sync every 5 seconds

Key Technologies and Responsibilities:

- **Express.js:**
 - Handles HTTP routing.
 - Parses incoming request bodies and encodes outgoing HTML responses.
 - Ensures clean content delivery with appropriate headers and encoding.
- **RESP Protocol:**
 - Lightweight, simple serialization used between the Proxy and Redis-like server.
 - Enables clean command exchange like SET key value and GET key.
- **Custom Redis-like Server:**
 - RESP parser.
 - In-memory data store.
 - Persistent file writer (refreshes every 5s)

Network Security (CS3403) Project Report

5. Security Features:

This proxy cache system incorporates multiple layers of **network security features** to ensure data integrity, prevent unauthorized access, and mitigate common web-based attacks. The security considerations are embedded across all components of the architecture, focusing on safe communication, access control, and input validation.

1. DNS Whitelisting

- Only a predefined list of trusted domains is allowed to be accessed by the proxy server.
- Prevents access to unverified or malicious URLs and limits the attack surface.

2. Private IP Address Blocking

- Requests targeting private IP ranges (e.g., 10.0.0.0/8, 192.168.0.0/16, 127.0.0.1) are explicitly denied.
- Protects internal infrastructure from **SSRF (Server-Side Request Forgery)** and internal port scanning attacks.

3. TCP VPN Tunnelling

- All communication between the **proxy cache** and the **Redis-like cache server** is done over a secured **VPN tunnel**.
- Encrypts traffic, ensuring **confidentiality** and **integrity** of cache data, and protects against **man-in-the-middle (MITM)** attacks.

4. RESP Command Parsing with Input Validation

- The Redis-like server parses only well-formed RESP commands (GET, SET, DELETE) and ignores malformed or suspicious inputs.
- Prevents command injection and maintains a strict communication protocol.

5. HTTP-Level Protection with Express

- The proxy server uses **Express.js** to encode and sanitize all HTML responses before sending them to the client.
- Helps prevent **HTML injection**, **cross-site scripting (XSS)**, and ensures that response headers are correctly set to avoid caching sensitive data on the client side.

6. Separation of Concerns & Service Isolation

- The frontend, proxy, and cache server are logically isolated.
- The cache server is **not directly accessible from the outside**, reducing exposure to attacks.
- Access control is enforced through internal routing and firewall/VPN rules.

7. Persistent Storage with Controlled Write Access

- The Redis-like server writes to persistent storage every 5 seconds in a controlled and isolated process.
- Prevents file-level race conditions or corruption from concurrent writes or external interference.

Network Security (CS3403) Project Report

6. System Requirements

Platform and System Requirements

The system was designed with a focus on delivering **low latency** and **high security** while maintaining flexibility and ease of deployment. Below are the key platform and system requirements that support the implementation and performance of the proxy cache system:

1. Platform Requirements

- **Node.js (v16 or higher):**
 - Required for implementing the proxy cache server using Express.js.
 - Provides non-blocking I/O and asynchronous capabilities critical for low-latency data processing.
- **Custom Redis-like Server (Built in Node.js):**
 - Lightweight, TCP-based service that mimics Redis functionality.
 - Capable of handling RESP commands and persisting data every 5 seconds.
- **Operating System:**
 - Compatible with **Linux (Ubuntu/Debian-based)** for production environments.
 - Works on Windows/macOS for development purposes.
- **VPN Software (e.g., WireGuard):**
 - Ensures encrypted communication between the proxy server and the cache server over TCP.
 - Essential for securing cache data transfers and reducing exposure.

2. System Requirements

- **CPU:**
 - Dual-core processor or higher recommended for parallel handling of client requests and cache operations.
- **Memory:**
 - Minimum 512 MB RAM for development.
 - At least 2 GB RAM for production, depending on the expected request volume and cache size.
- **Network:**
 - Low-latency and stable network connection to reduce delays in fetching and caching HTML content.
 - VPN tunneling requires consistent bandwidth to ensure encryption overhead does not hinder performance.
- **Disk Storage:**
 - Persistent storage (SSD recommended) for writing cached data every 5 seconds.
 - Requires minimal disk space unless caching large amounts of HTML content.

3. Security Requirements

- **Firewall Configuration:**
 - Restrict access to the Redis-like server to VPN traffic only.
 - Block private IP requests and enforce DNS whitelisting at the proxy level.
- **DNS and IP Filtering:**

Network Security (CS3403) Project Report

- Implementation of allowlists for trusted domains and denial of private or internal IP ranges.
- **Process Isolation:**
 - Run services as separate users or containers (Docker support possible) to limit the blast radius of any potential compromise.

7. Open-source libraries and tools

Node.js v18 – Core runtime, handles async I/O and TCP communication.

Express v4 – Simplifies HTTP request/response handling and HTML encoding.

WireGuard – Lightweight VPN for secure TCP tunnel between proxy and cache.

net module (Node.js) – TCP socket server/client for RESP command exchange.

dns module (Node.js) – Resolves and filters domains, blocks private IPs.

Vanilla JavaScript – For custom RESP parsing and cache logic.

Redis-cli – Used for testing RESP format with real Redis commands.

fs module – For periodic writes of cache data to disk (persistent storage).

set Interval – To refresh/write cache every 5 seconds.

Postman / curl – Testing endpoints, validating headers and responses.

8. Implementation and Testing

To validate the design and functionality of the system, a combination of automated and manual testing methods was used:

- **Unit Testing with Node.js assert module:**
Verified correctness of RESP command parsing (GET, SET, DELETE) in the Redis-like cache server.
- **Manual Functional Testing:**
Confirmed cache behavior by observing **cache hits and misses** during repeated requests.
Verified that HTML responses are correctly fetched, cached, and served.
- **Security Testing:**
Manually tested blocking of private IPs and DNS filtering to prevent unauthorized access.
- **Performance Testing:**
Measured **Largest Contentful Paint (LCP)** to evaluate page load improvement.
Observed up to **10x reduction in LCP** when served from cache vs. direct fetch.

These tests ensured that the system met its goals of correctness, security, and speed.

Network Security (CS3403) Project Report

9. Results

The implemented system successfully delivers a secure, low-latency proxy caching mechanism with the following key features:

Features Implemented

- **Proxy Cache in Node.js** – Handles HTTP requests, caches HTML responses.
- **Custom Redis-like Cache Server** – Supports GET, SET, and DELETE via RESP over TCP.
- **Persistent Storage** – Writes in-memory cache to disk every 5 seconds.
- **Cache Hit/Miss Logic** – Automatically fetches and stores on miss.
- **Secure Communication** – Proxy and cache interact over a VPN tunnel (WireGuard).
- **DNS Whitelisting & Private IP Blocking** – Prevents SSRF and unauthorized requests.
- **Performance Boost** – Achieved ~10x faster Largest Contentful Paint (LCP) via cache.

Future Extensions

- **AOF (Append Only File) Persistence** – Logging all commands for replay during recovery.
- **Eviction Policy Support** – Implementing LRU/TTL for memory-bound environments.
- **Docker Support** – Containerized setup for easier deployment and testing.
- **Admin Panel / Metrics UI** – For cache stats, hit/miss ratio, and live logs.
- **HTTPS Proxy Support** – Allow secure proxying for HTTPS targets.
- **Multi-instance Cluster** – Scalable cache with load balancing and sharding.

10. Conclusion

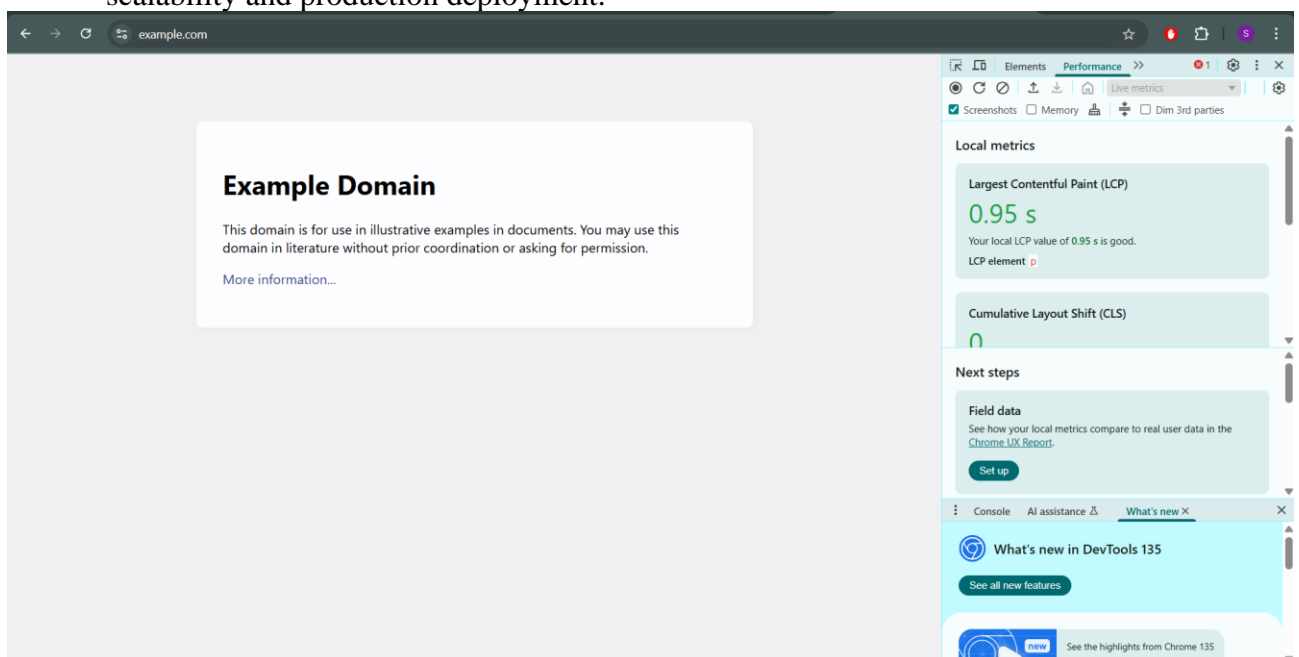
This project successfully demonstrates the implementation of a secure and efficient proxy cache system using Node.js and a custom Redis-like caching server. By combining lightweight technologies with core networking concepts such as TCP

Network Security (CS3403) Project Report

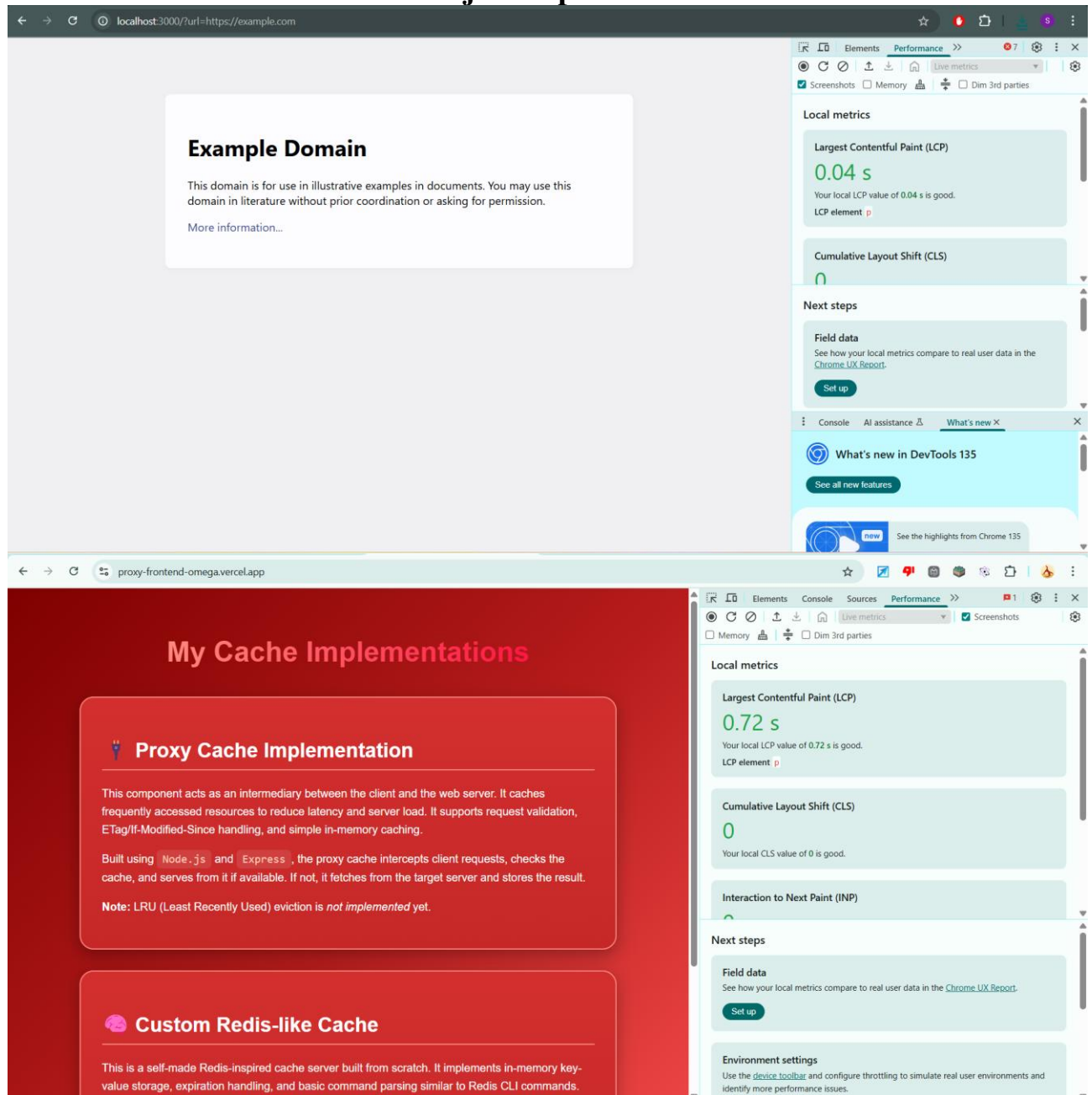
socket communication, VPN tunneling, and DNS filtering, the system achieves significant performance improvements—most notably a 10x reduction in Largest Contentful Paint (LCP).

Security was a core focus throughout the design, with features like DNS whitelisting, private IP blocking, and VPN-based isolation ensuring that the system is resilient to common attack vectors such as SSRF and unauthorized access.

Overall, the project meets its goals of enhancing web performance while maintaining strict network security controls. With future extensions like AOF persistence and eviction strategies, the system is well-positioned for real-world scalability and production deployment.



Network Security (CS3403) Project Report



The screenshot displays two web pages with their respective performance metrics in the Chrome DevTools Performance panel.

Example Domain

This domain is for use in illustrative examples in documents. You may use this domain in literature without prior coordination or asking for permission.
[More information...](#)

Performance Metrics:

- Largest Contentful Paint (LCP):** 0.04 s. Your local LCP value of 0.04 s is good.
- Cumulative Layout Shift (CLS):** 0

My Cache Implementations

Proxy Cache Implementation

This component acts as an intermediary between the client and the web server. It caches frequently accessed resources to reduce latency and server load. It supports request validation, ETag/If-Modified-Since handling, and simple in-memory caching.

Built using `Node.js` and `Express`, the proxy cache intercepts client requests, checks the cache, and serves from it if available. If not, it fetches from the target server and stores the result.

Note: LRU (Least Recently Used) eviction is *not implemented* yet.

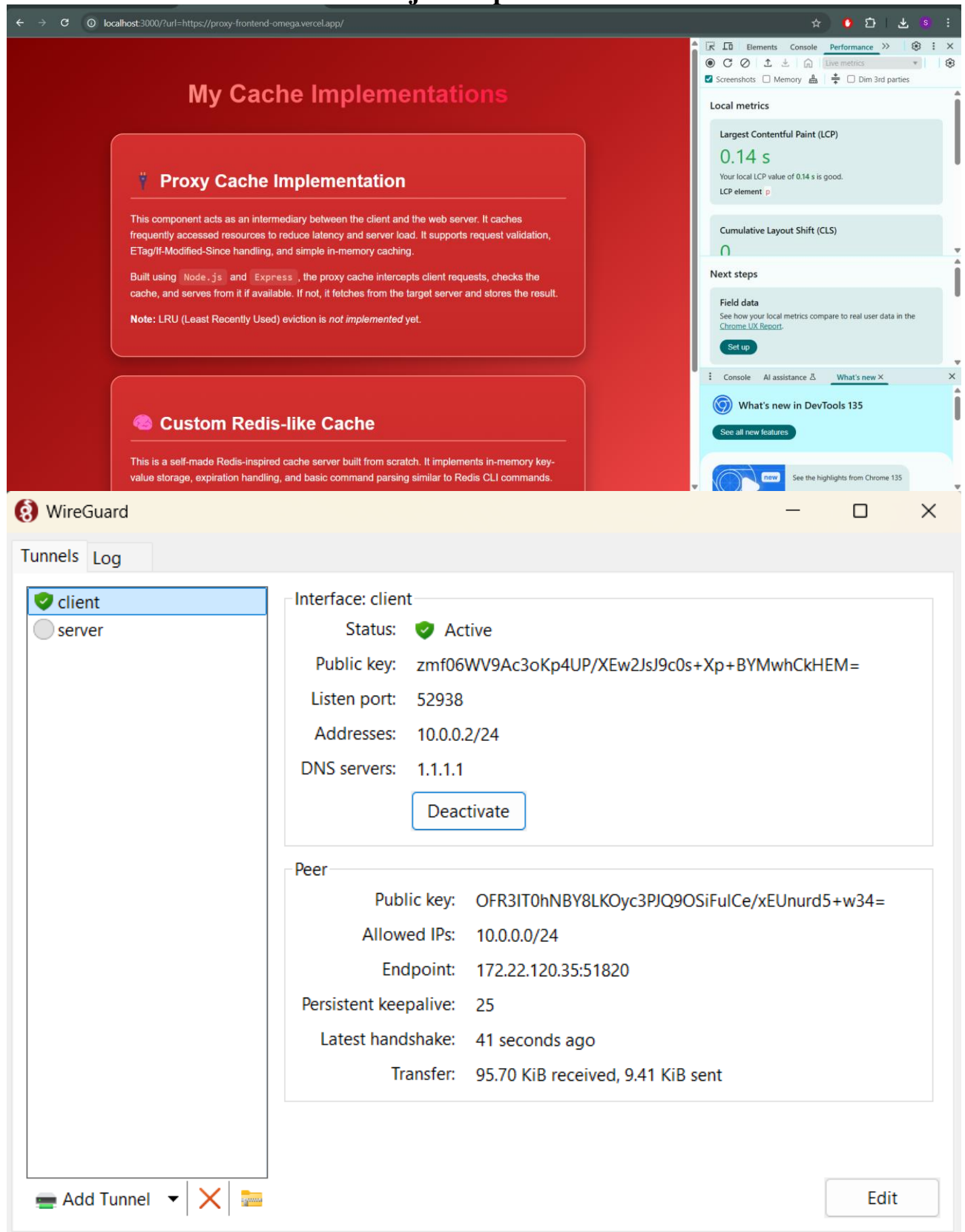
Custom Redis-like Cache

This is a self-made Redis-inspired cache server built from scratch. It implements in-memory key-value storage, expiration handling, and basic command parsing similar to Redis CLI commands.

Performance Metrics:

- Largest Contentful Paint (LCP):** 0.72 s. Your local LCP value of 0.72 s is good.
- Cumulative Layout Shift (CLS):** 0. Your local CLS value of 0 is good.
- Interaction to Next Paint (INP):** 0

Network Security (CS3403) Project Report



The screenshot displays a web browser window at `localhost:3000?url=https://proxy-frontend-omega.vercel.app/` showing a page titled "My Cache Implementations". The page features two main sections:

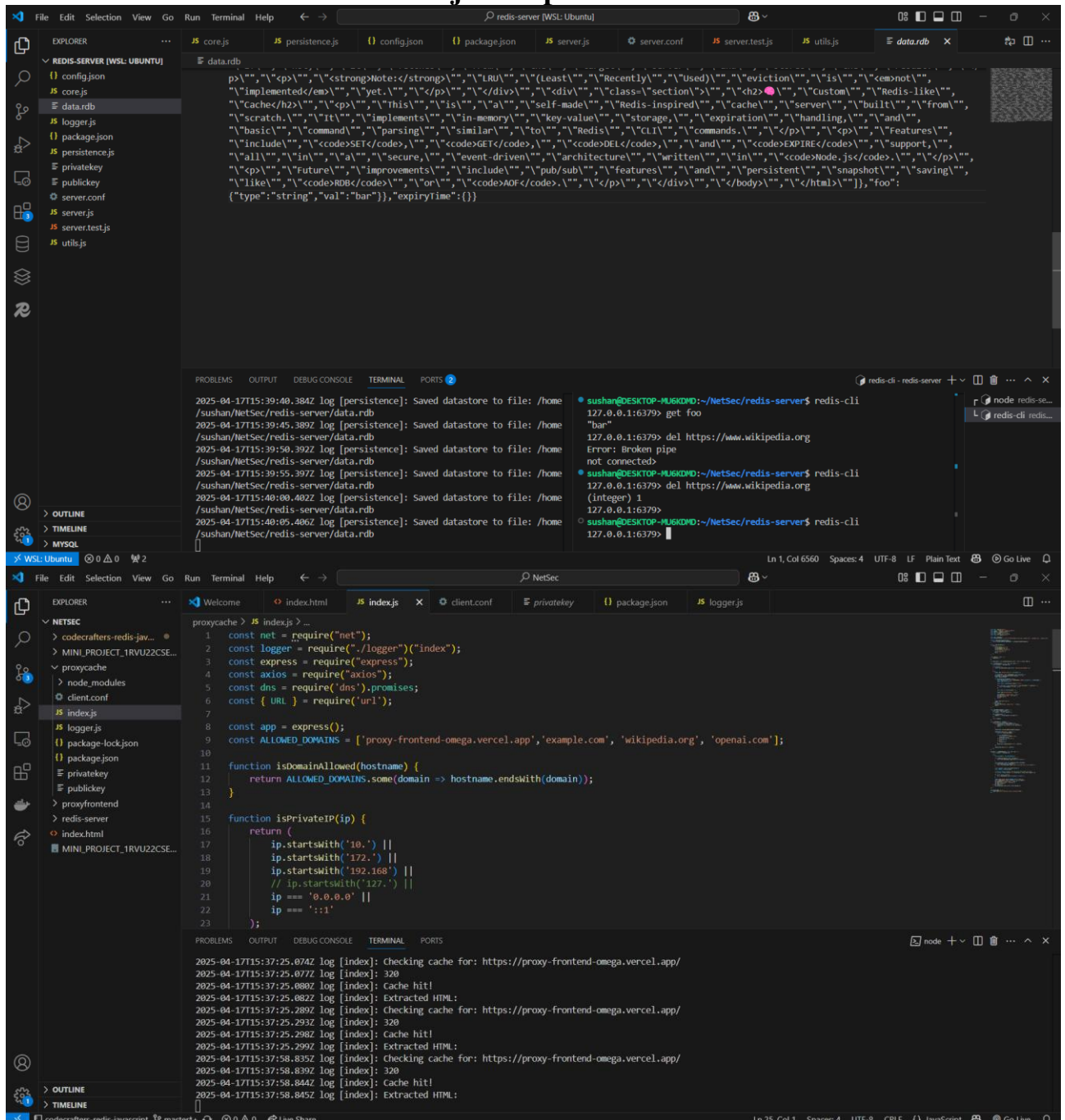
- Proxy Cache Implementation**: Describes an intermediary between the client and the web server, supporting request validation, ETag/If-Modified-Since handling, and in-memory caching. It mentions being built using `Node.js` and `Express`, and includes a note that LRU (Least Recently Used) eviction is not implemented yet.
- Custom Redis-like Cache**: Describes a self-made Redis-inspired cache server built from scratch, implementing in-memory key-value storage, expiration handling, and basic command parsing similar to Redis CLI commands.

On the right side of the browser window, the Chrome DevTools Performance tab is open, showing local metrics such as Largest Contentful Paint (LCP) at 0.14 s and Cumulative Layout Shift (CLS). Below the browser window, the WireGuard application is open, showing the "client" interface. The interface details include:

- Interface: client**
 - Status: ✔ Active
 - Public key: `zmf06WV9Ac3oKp4UP/XEw2JsJ9c0s+Xp+BYMwhCkHEM=`
 - Listen port: 52938
 - Addresses: 10.0.0.2/24
 - DNS servers: 1.1.1.1
 - Deactivate button
- Peer**
 - Public key: `OFR3IT0hNBY8LKOyc3PJQ9OSiFulCe/xEUurd5+w34=`
 - Allowed IPs: 10.0.0.0/24
 - Endpoint: 172.22.120.35:51820
 - Persistent keepalive: 25
 - Latest handshake: 41 seconds ago
 - Transfer: 95.70 KiB received, 9.41 KiB sent

At the bottom of the WireGuard window, there is an "Add Tunnel" button and an "Edit" button.

Network Security (CS3403) Project Report



The screenshot displays two instances of Visual Studio Code. The top instance is running a Redis server on a WSL Ubuntu environment. The Explorer pane shows files like `core.js`, `data.rdb`, `logger.js`, `package.json`, `persistence.js`, `privatekey`, `publickey`, `server.conf`, `server.js`, `server.test.js`, and `utils.js`. The Terminal pane shows Redis logs indicating successful data saving to `/home/sushan/NetSec/redis-server/data.rdb`. A second terminal window shows Redis CLI commands: `redis-cli`, `127.0.0.1:6379> get foo`, `127.0.0.1:6379> del https://www.wikipedia.org`, and `127.0.0.1:6379> del https://www.wikipedia.org`.

The bottom instance of VS Code shows a proxy server implementation. The Explorer pane shows files like `index.html`, `index.js`, `client.conf`, `privatekey`, `package.json`, `logger.js`, `package-lock.json`, `proxyfrontend`, `redis-server`, and `index.html`. The Terminal pane shows logs for the proxy server, including checking cache for `https://proxy-frontend-omega.vercel.app/` and extracting HTML.
