Architecture of Content Delivery Network

Anfimov Vladimir 2B3

6 December 2022

Abstract

An architecture of Content Delivery Network (CDN) is to be presented is this paper, with initial focus on the implementation details on different levels, such as the transport level where the system uses TCP/IP and the application level where HTTP/1.1 is used for plain text responses to the end user. The CDN system is made up of many nodes, each kept in a Docker container, with the role of caching and a proxy that distributes the workload on all machines depending on the algorithm used: Round robin or Least connection. Data propagation in the CDN takes place through the Pull method, so that the file is stored once its first request occurs, and its lifetime is limited by using the TTL (time to live) concept.

1 Introduction

Content Delivery Networks appeared due to the rapid need to access information on the Internet, as the monolithic architecture had become intractable. CDNs solve this problem by distributing data in several geographical areas depending on the required data flow, and the nodes that make up this network are both data storages and proxy servers.

The CDN implemented in this report is minimalist, but efficient in serving GET requests on the HTTP/1.1 protocol, where the response received from the server is in text/plain format. Clients can use a browser or a CLI (command line interface) such as CURL to make requests to the computer network. To upload new files in the network, namely in the origin node, the administrator will be able to use the supervisor panel, where the history of the network flow is also visible.

This paper continues with the chapter "Technologies" where it is explained why TCP/IP, HTTP/1.1, Docker and SQLite were necessary in the creation of the system. Furthermore, in the "Application Architecture" is presented the way in which the nodes interact with each other, with the proxy and the supervisor server. The chapter "Implementation details" includes key aspects about load balancing, cache policy and health checking system. The "Conclusion" ends with possible improvements for the CDN.

2 Technologies

The technologies used in the project can be divided into protocols, such as TCP/IP and HTTP/1.1, with a role in establishing the mode of communication in the network and imposing a standard, and applications, such as Docker, for configuring virtual machines, called containers, each with a SQLite database that serves as a cache.

2.1 TCP/IP

Transmission Control Protocol (TCP) provides a host-to-host communication by providing a channel for the communication needs of the applications. In order to work as expected, a CDN must return data without missing bytes, in this case TCP is ideal because it provides flow-control and reliable transmission of data by design. Because is categorized as connection-oriented, both client and server benefit from having the knowledge of its partner conectivity situation. TCP does not handle routing, thus Internet Protocol (IP) is used to provide routing of packets from a source address to a destination address.

2.2 HTTP/1.1

The Hypertext Transfer Protocol (HTTP) is an application layer protocol that sits on top of the transfer layer, which uses TCP/IP in the current CDN. HTTP/1.1 transfers data in plain-text messages [2] and includes in its headers information as verb, response status and even more. In this project the client makes a GET request from a browser (or a CLI as Curl) in order to "get" the requested page. Even if the 1.1 standard introduces the possibility to keep the connection alive, the server implemented in this report will close the connection after returning the information.

2.3 Docker

Docker is a platform that uses OS-level virtualization to deliver software in packages called containers [3]. These containers are lightweight and because of this a single machine could handle many of them. An application, like a server, can be deployed fast by creating an instance (a container) from an image which includes all the instructions and the binaries of the installation. In order to communicate with a server that is inside a container a TCP port is disposed. The CDN from this paper will use a container for the proxy and for each node.

2.4 SQLite

SQLite is a database engine that was built as a library which can be embedded in apps. It stores the whole database as a file and it was configured to require little to none configuration. It provides safe concurrent access from multiple threads (by locking the entire file) and in this CDN each container will have such a database.

3 Application Architecture

The CDN implemented in this paper is composed of the following components: a reverse proxy with the role of a load balancer, a lot of interconnected nodes that cache data about sites (URLs, content, TTL), and a server called *Supervisor* that manages the system at a high level. Each such component has a suite of services, some exposed externally for network communication, others only internally for decoupling and separation of concerns. Following is a presentation of the system components and the services available.

3.1 Reverse Proxy and Load Balancer

The client interacts directly with this component through a HTTP/1.1 connection and has a great power of abstracting the system, as it exposes a single IP address to the client. The *Proxy Service* interacts with the *Analysis System Service* to see the nodes exposed in the network, their workload and health status, following to establish a new TCP/IP connection to the optimal node to receive the content requested by the client. The latter service constantly receives this information from the nodes in the network. The two load balancing methods, Round Robin and Least Connection, will be explained in the chapter on implementations.

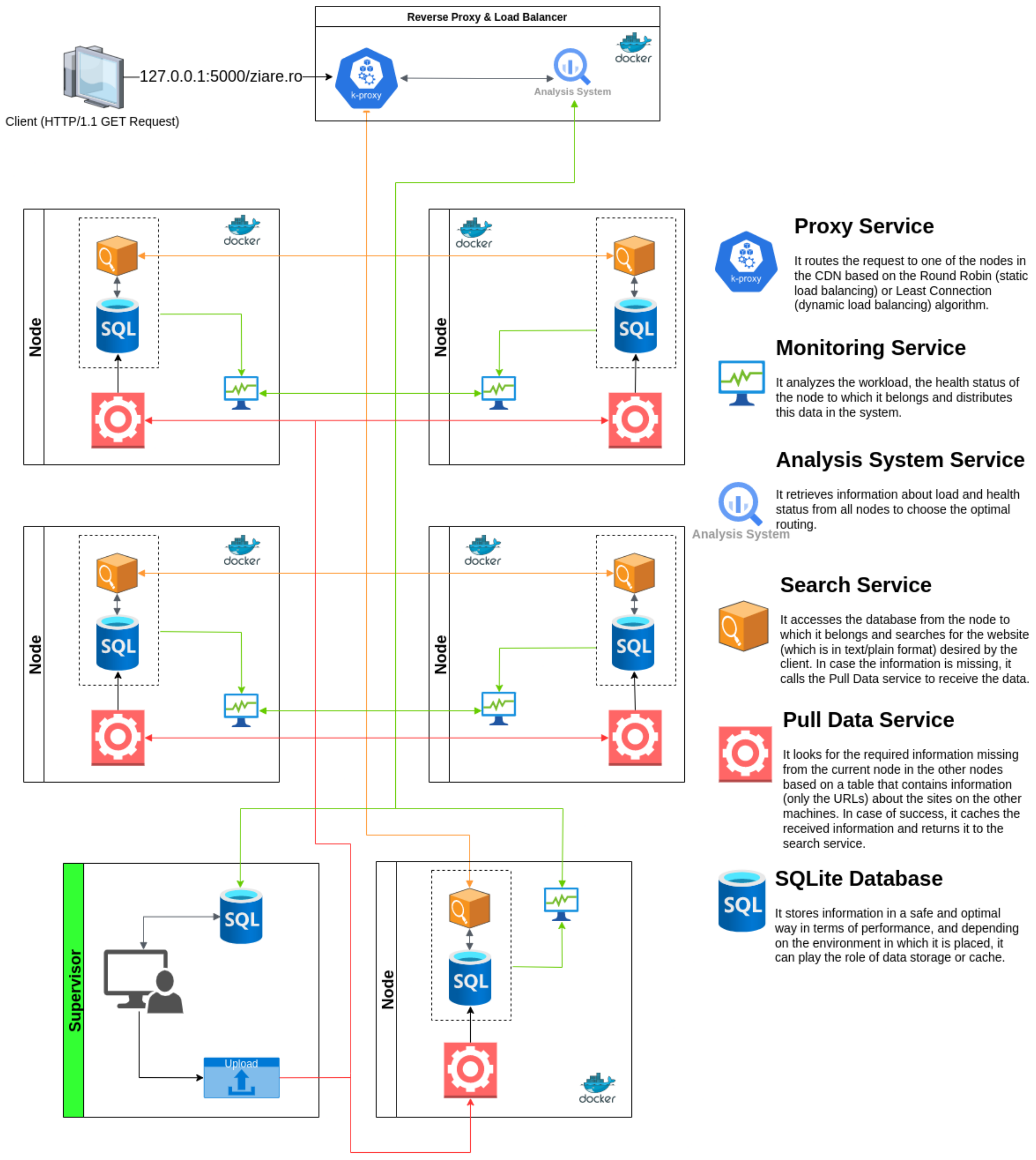
3.2 Node

A node is the most frequent component in the network. It runs in a minimalist virtual machine, along with a database that can cache URLs, page content and TTL data. There are three main services:

1. *Search Service* is an API that exposes content search functionalities based on the input parameters, namely the URL. If the information is not found, it calls *Pull Data Service.*
2. *Pull Data Service* uses a local table, which is constantly updated, to find the node where the content requested by the client is located. If the content is found on another machine, it will "pull" the data, return it to the search service and cache it in the local database.
3. *Monitoring Service* analyzes the number of connections and the health status of the component to which it belongs in order to inform the network of its situation. Thus, it regularly sends this information to the other components through long-term TCP/IP connections. To the Supervisor it also sends data about the served requests.

3.3 Supervisor

The supervisor is the control server that can start, configure and stop the CDN. It also has a history of requests from the network and has the functionality of uploading information (URL, content, TTL) in an arbitrarily chosen node. All these things can be accessed through a CLI that establishes communication with the control server.



Proxy Service

It routes the request to one of the nodes in the CDN based on the Round Robin (static load balancing) or Least Connection (dynamic load balancing) algorithm.

Monitoring Service

It analyzes the workload, the health status of the node to which it belongs and distributes this data in the system.

Analysis System Service

It retrieves information about load and health status from all nodes to choose the optimal routing.

Search Service

It accesses the database from the node to which it belongs and searches for the website (which is in text/plain format) desired by the client. In case the information is missing, it calls the Pull Data service to receive the data.

Pull Data Service

It looks for the required information missing from the current node in the other nodes based on a table that contains information (only the URLs) about the sites on the other machines. In case of success, it caches the received information and returns it to the search service.

SQLite Database

It stores information in a safe and optimal way in terms of performance, and depending on the environment in which it is placed, it can play the role of data storage or cache.

Figure 1: CDN Architecture Overview

The architecture of the application consists in the separation of responsibilities in a distributed network that comes with the following benefits:

1. *Quick access* to information because the data is duplicated on several nodes
2. *Reduced downtime* because when a node goes down there are others that can serve the requests
3. *High scalability* due to the fact that it is easy to add more nodes to the network

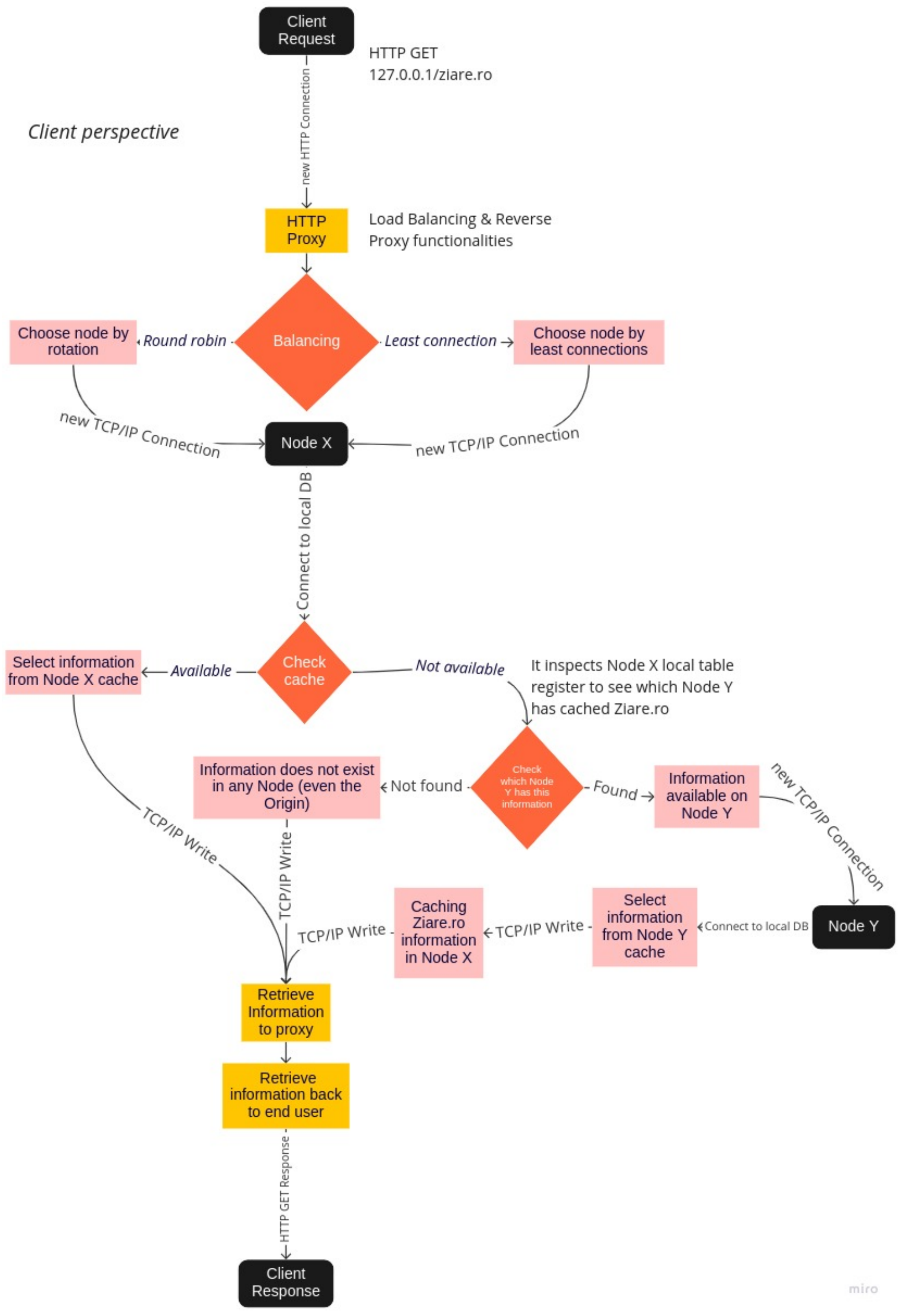

Client Request
HTTP GET
127.0.0.1/ziare.ro
Client perspective
new HTTP Connection
HTTP Proxy
Load Balancing & Reverse Proxy functionalities
Choose node by rotation
Round robin
Balancing
Least connection
Choose node by least connections
new TCP/IP Connection
Node X
new TCP/IP Connection
Connect to local DB
Select information from Node X cache
Available
Check cache
Not available
It inspects Node X local table register to see which Node Y has cached Ziare.ro
Information does not exist in any Node (even the Origin)
Not found
Check which Node Y has this information
Found
Information available on Node Y
new TCP/IP Connection
TCP/IP Write
TCP/IP Write
TCP/IP Write
Caching Ziare.ro information in Node X
TCP/IP Write
Select information from Node Y cache
Connect to local DB
Node Y
Retrieve Information to proxy
Retrieve information back to end user
HTTP GET Response
Client Response
miro

Figure 2: Client Flow Perspective

4 Implementation details

This chapter includes the implementation of some important mechanisms of the CDN. For a better understanding pseudocode snippets have been introduced.

4.1 Load Balancing

Load Balancing is implemented in the Proxy component and can use either the Round Robin (RR) or Least Connection (LC) algorithm to make optimal routing. Both algorithms iterate through a list containing tuples of the form (node port, timestamp of last health check, number of connections). Each node regularly sends such a tuple to the proxy, so that if the timestamp variable has not been updated for 3 seconds, the load balancer will consider that the respective node is inactive. RR takes each node in cyclic order for routing, while LC chooses the node with the fewest active connections.

```
port = -1
minimumConnections = MAX
foreach node in nodesList:
    if now() - node.timestamp < 3 and node.connections < minimumConnections:
        port = node.port
        minimumConnections = node.connections
response = await newTCPConnectionTo(port)
```

Figure 3: Least Connection Algorithm

This list is continuously modified in another thread of the proxy that receives health checks from each node through long-term TCP connections.

```
loop {
    node = await waitForMessage()
    if node not in nodesList:
        nodesList.add(node)
    else:
        index = nodesList.findByPort(node.port)
        lock(nodesList[index])
        nodesList[index].connections = node.connection
        nodesList[index].timestamp = node.timestamp
        unlock(nodesList[index])
}
```

Figure 4: Health checking thread

Initially, if a node has not sent the health check for 3 seconds, it will be ignored during load balancing, but if more than 15 seconds pass without a ping, a job (from another thread) will remove it from the list of candidates.

```
InactiveNodesRemover() {
    for node in nodesList:
        if now() - node.timestamp >= 15:
            lock(nodesList)
            nodesList.remove(node)
            unlock(nodesList)
}
```

Figure 5: Inactive nodes remover

4.2 Cache Policy

In this project, the Cache Pull method is implemented. The supervisor can choose a node X in which to load a tuple of form *(URL, content, TTL)*. After the upload, the entire network receives a simplified tuple *(URL, TTL, node X)* so that all nodes are kept up to date with the state on other machines. Having this knowledge, any node can do the optimal routing if it does not find the required content in the local cache.

```
request = await waitForRequests()
if request.url in localDB.cache:
    content = localDB.cache.select(content).where(url)
else:
    node = findNodeWithCachedContent(url)
    (content, TTL) = pullContent(node)
    localDB.cache.add(content, TTL)
await sendResponse(content)
```

Figure 6: Searching for content and caching in case of miss

The *findNodeWithCachedContent* method queries the local database and chooses the node that has the most recently updated content. Each node has a scheduler that starts at a regular interval to check if there is information in the cache that should be deleted as a result of the expiration determined by the TTL field.

5 Conclusion

This project represents a minimalist architecture of a CDN and there are many possible improvements that could be made. For example, a node could be organized in several databases with duplicate information, which would greatly increase the speed of access to the cache. Also, another caching method is Push, which is more efficient at the first page load. This is due to the fact that when a page is uploaded in a node, it will distribute the content to all nodes in the network, which reduces the number of cache misses.

In conclusion, the project wants to introduce the basics of a distributed system with obvious performance benefits compared to monolithic models that cannot manage such a large number of clients.

References

[1] Hypertext Transfer Protocol Wiki, https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

[2] HTTP/1.1 vs HTTP/2: What's the Difference?, https://www.digitalocean.com/community/tutorials/http-1-1-vs-http-2-what-s-the-difference

[3] Docker Wiki, https://en.wikipedia.org/wiki/Docker_(software)

[4] SQLite Wiki, https://en.wikipedia.org/wiki/SQLite

[5] Types of load balancing algorithms, https://www.cloudflare.com/learning/performance/types-of-load-balancing-algorithms/

[6] HTTP Server, https://medium.com/from-the-scratch/http-server-what-do-you-need-to-know-to-build-a-simple-http-server-from-scratch-d1ef8945e4fa

[7] TCP/IP Wiki, https://ro.wikipedia.org/wiki/TCP/IP

[8] Docker Containers, https://www.docker.com/resources/what-container/