Library and Nginx System Architecture Review

Software Systems Architecture

Group 7 – Class 3

Master in Informatics and Computing Engineering

Dinis R. dos Santos Bessa de Sousa	up202006303@fe.up.pt
João Rola Reis	up202007227@fe.up.pt
Pedro Manuel da Silva Gomes	up202006322@fe.up.pt
Ricardo André Araújo de Matos	up202007962@fe.up.pt

Part 1 - Library architecture review

Assessment Rubric
Assignment (which architecture): Library

Which team are you reviewing: T25

What is your team: T37

Topic	Strengths and suggestions for the architects
Diagrams	Good
Clarity	Very Good
Consistency	Very Good
Completeness	Fair
Sufficient Level of detail	Good
Text Description	Very Good
Clarity	Very Good
Consistency	Very Good
Sufficient?	Good
Correctness	Fair
Anything missed?	Fair
Will it work?	Fair
Presentation	Very Good
Summary	Good

Explanation:

- **Diagrams**

  There is only one diagram. It is a UML, which is clear and consistent. The diagram does not seem sufficient to represent the system, representing only a logic level, as many doubts remain (what are the physical means used, how can the parts of the system interact with each other). For instance, there's ambiguity regarding the scaling of servers and the integration process with other library search systems (is it a global system? or is the library system for only one library?). Furthermore, the design of the MVC architecture remains unclear when examining the class diagram.

- **Text description**

  The text is simple, clear, while the description of the entities is consistent with the diagram . However, it remains unclear, for example, how the design of the system was inspired by the MVC architecture. Some of the information could be in another diagram maybe, like the use scenarios, as it would be easier to understand.

- **Correctness**

  There is no reference to some of the considerations, like considerations 2, 3, and 4 (from the Hints and Considerations of the Library System Architecture assignment). It remains unclear what are the different ways to use the system (only mentions interaction in the library) and there are no considerations about separation of concerns or coupling.

- **Presentation**

  The report is clean.

- **Summary**

  The report lacks the sufficient information to implement the design, being only a starting point, but more information is needed.

Part 2

2.1 Nginx introduction

Nginx is an open source web server that focuses on high performance, high concurrency and low memory usage. It can also be configured as a reverse proxy, a load balancer, and a cache.

Servers have to deal with huge amounts of clients. Two decades ago, the major cause of concurrency was slow clients. A server would rapidly pull the source to send to the client, however bad connections would stall the request and server. Because of that, it would be advantageous to spawn new processes. However, with the advance of technology it became necessary to maintain millions of connections always open to receive notifications.

A web server cannot afford to allocate a separate process or thread for each connection due to the cumulative resource overhead per process and the performance impact of creating and deleting processes, which can eventually lead to poor performance due to thread thrashing on excessive context switching.

2.2 Nginx Architecture

Nginx was written with an architecture that is much more suitable for nonlinear scalability in both the number of simultaneous connections and requests per second than their competitors. The system uses multiplexing and event notifications, assigning specific tasks to separate processes within a limited number of single-threaded workers for efficient connection processing.

Worker processes accept new requests from a shared “listen” socket and execute a highly efficient run-loop to process thousands of connections each. The way connections are distributed to workers is OS-dependent (epoll on Linux, kqueue on BSD-based OSes, etc). Overall, the key principle is to be as non-blocking as possible. The worker run-loop relies heavily on asynchronous task handling, implemented through modularity, event notifications and callback functions. Module invocation is extremely customizable, as it is performed through a series of callbacks using pointers to the executable functions.

NGinx follows the master-slave architecture. The master is responsible for job allocation to the workers. The **master process** runs privileged operations (reading configurations, binding to ports, creating child processes,creating, binding and closing sockets, …).

Meanwhile, the **worker processes** handle network connections, read and write content to disk and communicate with upstream servers. They wait for events on the listen sockets and when an event arrives by new incoming connections these connections are assigned a state machine (HTTP state machine, raw TCP, SMTP …). These state machines tell NGINX how to process a request.

Inside the state machine, each HTTP request passes through a sequence of phases. In each phase a distinct type of processing is performed on the request. Module-specific handlers can be registered in most phases, and many standard nginx modules register their phase handlers as a way to get called at a specific stage of request processing. Phases are processed successively and the phase handlers are called once the request reaches the phase.

The NGX_HTTP_CONTENT_PHASE orchestrates the generation of responses to client requests. When the configuration specifies a handler, requests in the content phase are directed to this content handler. If the handler is not set, requests are routed to handlers defined in the main configuration. These content handlers generate response content, which is then passed through filters for further processing such as compression or rewriting, before being sent back to the client.

The postpone filter is used for subrequests. Subrequests are a very important mechanism for request/response processing. For example, a subrequest can be used for client authorization. Additionally, with subrequests nginx can return the results from a different URL than the one the client originally requested (internal redirects). Not only can filters perform multiple subrequests and combine the outputs into a single response, but subrequests can also be nested and hierarchical. A subrequest can perform its own sub-subrequest, and a sub-subrequest can initiate sub-sub-subrequests.

The **cache loader** checks the on-disk cache items and populates nginx's in-memory database with cache metadata. This is necessary to allow nginx instances to work with the files already stored on disk. The **cache manager** is mostly responsible for cache expiration and invalidation. Cache items might be used to serve static files, content cache, etc.

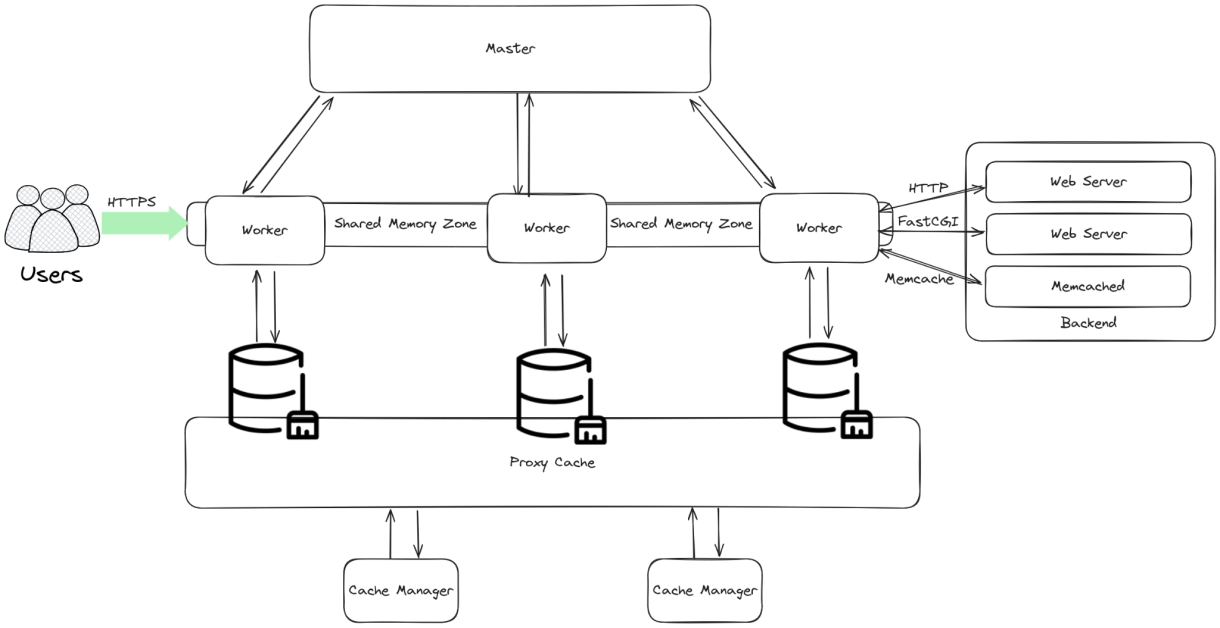

Figure 1: System Architecture

2.3 Architecture Patterns

- Master-slave:

Nginx runs several processes in memory. One (and only one) of the processes is the *master*, and the rest of them are *workers*.

The *master* process is responsible for reading and validating configuration; create, bind and close sockets; starting, terminating and maintaining the configured number of *worker* processes; as well as other actions that are related with configuration of the system.

We consider this to be a Master-slave Pattern as the *master* node controls the operations of the *worker* nodes which manage the computations and store the results.

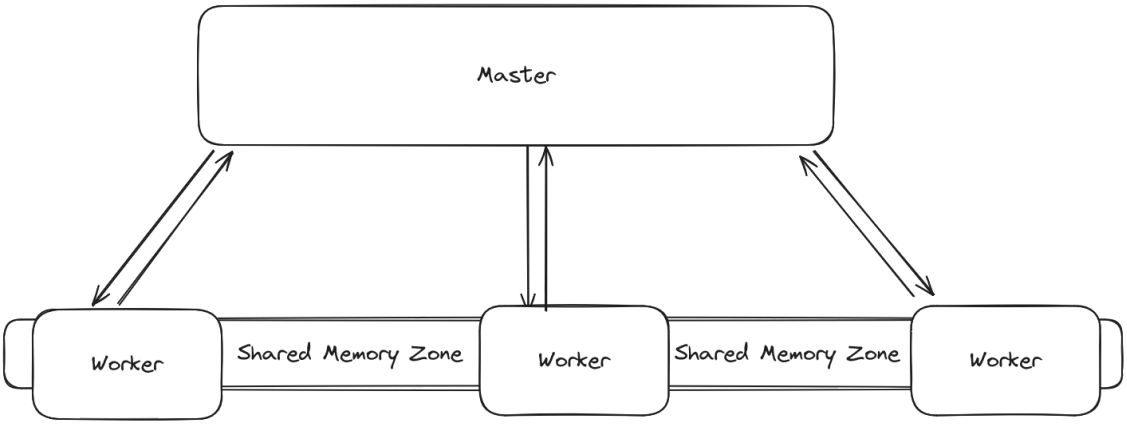

Figure 2: Master-slave pattern

- Shared repository:

The Shared Repository pattern, implemented through the Shared Memory Zone in Nginx, is used to share the state between the multiple worker processes.

For example, in the context of **DDoS attack mitigation**, the Shared Memory Zone serves as a central repository for tracking incoming requests, IP addresses, and request rates. By storing this information, Nginx can swiftly identify and mitigate DDoS attacks by applying rate limiting and request throttling measures, leveraging the shared data across worker processes to maintain a consistent defense strategy.

Additionally, in distributed environments with load balancing requirements, the Shared Memory Zone facilitates health monitoring and load balancing decisions. For instance, in scenarios where backend servers may go offline or experience issues, the Shared Memory Zone allows workers to quickly detect changes in server status and adjust traffic routing accordingly, ensuring high availability and reliability of web services.

Through these applications, the Shared Repository pattern in the Shared Memory Zone enhances security, scalability, and performance in Nginx deployments, addressing critical requirements in web server management and defense against cyber threats.

Figure 3: Shared Repository pattern

- Pipes and Filters:

Filters do the task of manipulating the output produced by a handler. A filter gets called, starts working, and calls the next filter until the final filter in the chain is called. It works much like a Unix pipeline. In figure 4, the response filters could include 3 filters: an image filter to resize an image, a gzip compression filter to compress the image and lastly a server-side includes filter to include the size of the compressed resized image in the response.

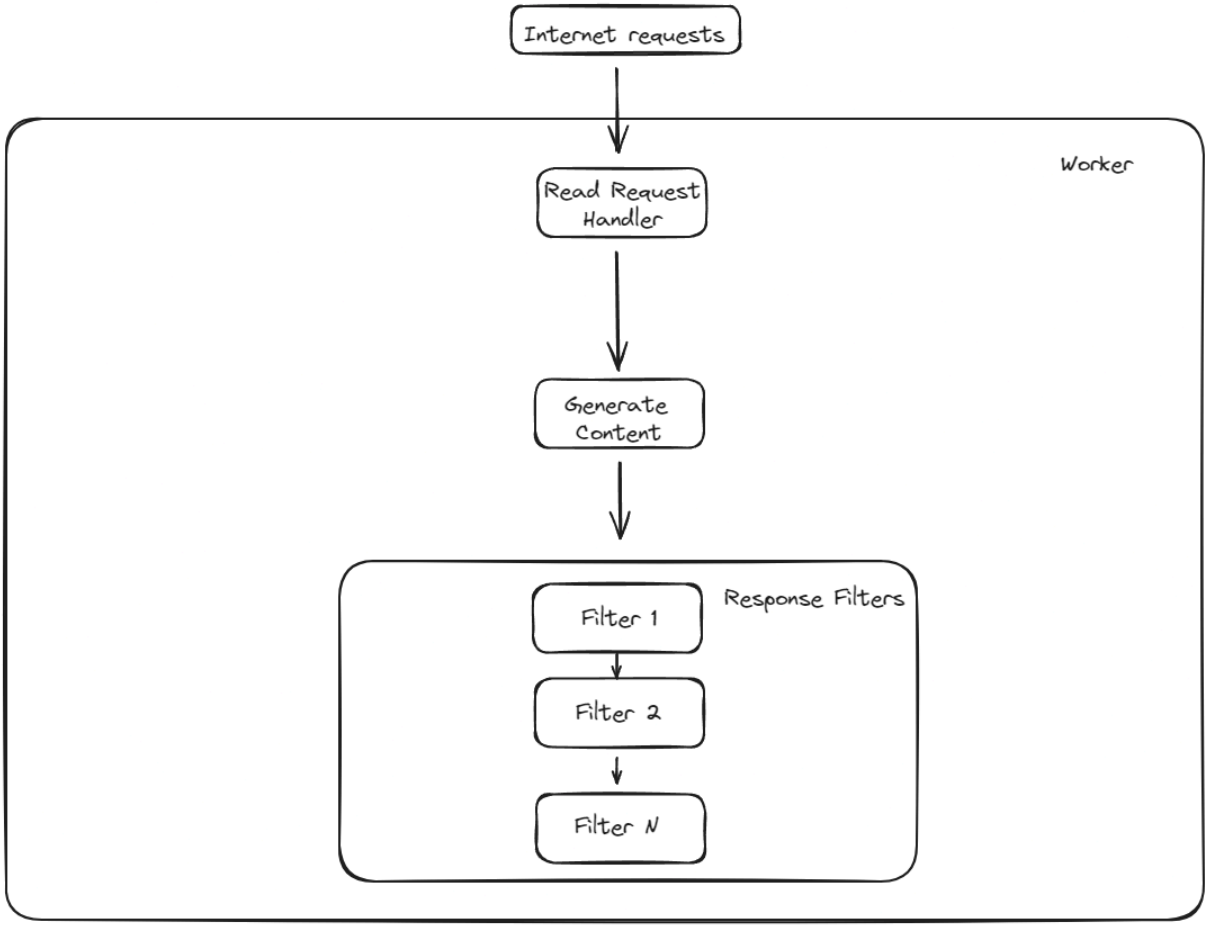

Figure 4: Pipes and Filters pattern

- Microkernel:

The core of nginx handles essential tasks like maintaining a run-loop, managing network protocols, and providing a runtime environment by executing the appropriate sections of modules on each stage of request processing.

Most application-specific features, such as serving web pages or acting as a reverse proxy, are implemented as separate modules. These modules extend the server's capabilities and can be dynamically loaded as needed.

In the microkernel pattern, a minimalistic core provides essential services, while additional functionalities are implemented as separate modules or processes. Nginx follows this pattern with its lightweight core and modular extensions.

By separating core functionality from application-specific features, nginx promotes modularity and flexibility, adhering to the principles of the microkernel architecture. This architecture enables easy extension and customization of the server without impacting its core stability or performance.

2.4 Quality attributes of the system

- Performance

The primary objective behind the creation of Nginx was to tackle the performance bottleneck caused by Apache's inability to handle thousands of simultaneous connections effectively. Nginx was designed with a distinct architecture centered around events. This event-driven approach proved instrumental in overcoming performance hurdles, enabling it to manage tens of thousands of concurrent connections efficiently. Additionally, Nginx introduced various features such as reverse proxy with load balancing, compression, and caching, further enhancing system performance.

- Scalability

As previously mentioned, Nginx was primarily developed to address performance concerns. Its *modular, event-driven, asynchronous, single-threaded, non-blocking architecture* offers users the ability to efficiently utilize server resources and seamlessly scale them as their websites grow, all while maintaining cost-effectiveness.

- Extensibility

Nginx exhibits strong extensibility through its support of loadable modules, which enables users to dynamically extend its functionality without modifying the core codebase. The flexible configuration system further enhances extensibility, empowering users to fine-tune Nginx according to their unique requirements.

- Maintainability

Maintainability in Nginx is upheld through adherence to C-style convention for the configuration files, facilitating readability and ease of maintenance for developers. This commitment to standardized coding practices promotes efficient collaboration and reduces the likelihood of errors or inconsistencies, ultimately enhancing the server's long-term maintainability. Additionally, the clarity of the configuration aids in troubleshooting and debugging efforts, enabling developers to identify and rectify issues swiftly.