Faculty of Engineering of the University of Porto

Homework 06

Understand Someone Else's Architecture

M.EIC010 - Software Systems Architecture

1MEIC03 - T32

Professors

Ademar Aguiar

Neil Harrison

Students

Ana Rita Baptista de Oliveira - up202004155@edu.fe.up.pt

Diogo Alexandre da Costa Melo Moreira da Fonte - up202004175@edu.fe.up.pt

João Paulo Moreira Araújo - up2020004293@edu.fe.up.pt

Tiago Nunes Moreira Branquinho - up202005567@edu.fe.up.pt

Part 1

Assessment Rubric
Assignment: HW 06 Part 1 - Understanding HW 05 architecture of other team

Which team are you reviewing: T21

What is your team: T32

Topic	Strengths and suggestions for the architects
Diagrams	
Clarity	There are no legends in any diagram, but the labels in each relation are good to help understanding the flows. The sequence diagrams are good to understand example scenarios.
Consistency	In the first diagram, interface blocks are different from the others. Why? What do they represent in specific?
Completeness	In the first diagram, authentication is not represented, raising questions about how the selection between Patron Interface or Staff Interface is made.
Sufficient Level of detail	In the first diagram, after the Inventory Management Component is represented the Electronic Resources Component (using the label "Manages using"), but in the text description it is mentioned that there are Library and Electronic Resources. Shouldn't this be splitted in those two, at least? Same applies to the left part of the diagram, that after the Catalog Management Component, only the Library Resources are represented. In our perspective, both Library and Electronic Resources should be represented on both sides (Catalog and Inventory).
Text Description	
Clarity	The description lacks some detail about each component.
Consistency	Consistent.
Sufficient?	The description of the Reservation System and the Notification using External Services lacks a lot of details to understand the architecture.
Correctness	
Anything missed?	An anonymous user is not allowed to search through the library catalog, why? In our opinion, an anonymous user should be able to search items, but not reserve them. Analyzing the text description and the second diagram, we realized that the Staff manages the reservation system, but is this viable? About the Notification Service, there are no notifications using their own system, to alerts of reservations for example? Where is the Collaboration Module between libraries?
Will it work?	There are details and components missing, such as the ones regarding the late delivery fees. It may work, but not with all the features requested.
Presentation	
Summary	In the current state, we are not confident that we can implement the design as it is described, due to the abstractions and the lack of certain details that missed out on the request page

Part 2

Introduction

In this report, we delve into the architecture of **Graphite**, an open-source time-series metrics monitoring system. Its choice as the subject of the report is motivated by its unique strengths - Graphite's architecture allows for efficient handling of time-series data, as its use of network-based architecture patterns and a push approach to data collection ensures scalability and efficiency. Furthermore, Graphite's quality attributes such as scalability, efficiency, and reliability make it a robust system for metrics monitoring. These strengths collectively emphasize Graphite's suitability for diverse environments and its adaptability to varying workloads.

In the following sections, we will start by summarizing Graphite's architecture, then we will identify and elaborate on the architecture patterns used in Graphite and discuss the important quality attributes of the system. This comprehensive analysis aims to provide a deeper understanding of why Graphite is a preferred choice for metrics monitoring in various domains.

Graphite's Architecture Overview

Graphite stands out in the realm of open-source time-series metrics monitoring systems, owed largely to its architectural design. At its core, Graphite embodies a sophisticated infrastructure designed to efficiently manage and visualize time-series data, exhibiting a feat that makes it a preferred choice across diverse domains.

Essentially, Graphite's architecture comprises three pivotal components:

- **Data Storage (Whisper) -** At the core of Graphite lies Whisper, a robust database library engineered to handle the storage of time-series data. Diverges from conventional database paradigms, opting instead for a file-based approach that offers unparalleled efficiency and scalability. Written in Python, just like the remaining components, Whisper empowers applications to directly interface with its storage mechanisms, facilitating read and write operations, crucial for managing high-frequency data streams. Its data format, encapsulated within each data point as **[**"**name**" "**value**" "**time in seconds**"**]**, embodies simplicity without compromising on performance, ensuring that Graphite doesn't bottleneck even among the most demanding data workloads.

- **Backend Daemon (Carbon)** - Graphite’s backend process is called Carbon, a resilient daemon built on top of the Twisted framework. Carbon assumes the role of ingesting and organizing incoming data points, cataloging them within a hierarchical structure for efficient retrieval and analysis. Designed with scalability in mind, Carbon can handle a multitude of client connections with ease, thanks to the integration with the Twisted framework. Its streamlined plaintext protocol offers clients a straightforward avenue for data submission, ensuring that Graphite remains agile and responsive even amidst surges in data volume.
- **Web Application and Real-Time Visualization** - Completing Graphite's architecture is its web application interface, a gateway to real-time visualization and analysis. Seamlessly integrating data from Whisper’s storage repository and Carbon’s buffered data points, the web application empowers users with a versatile suite of graphing capabilities. Leveraging a URL-based API, users can effortlessly summon customized graphs tailored to their specific needs, complete with an array of display options and data manipulation functionalities. Through its capability to integrate multiple data sources and facilitate real-time visualization, the web application solidifies Graphite’s standing as a potent tool for constructing dynamic, data-driven dashboards tailored to high-volume environments.

In essence, Graphite’s architecture stands as a testament to its adaptability, scalability, and reliability in the realm of metrics monitoring systems. By integrating Carbon, Whisper, and its web application interface, Graphite can join efficiency and versatility, ready to tackle the challenges posed by modern data ecosystems. As data and organizations continue to traverse the digital landscape, Graphite remains an up-to-date tool that offers insights and visualization capabilities, crucial for navigating the complexities of real-time time-series data with efficacy.

Architecture Patterns

In software engineering, architectural patterns provide reusable solutions to common problems. Whether software architects are consciously aware of it or not, they regularly rely on architectural patterns when designing software, and for good reason - patterns tend to offer a structured approach to designing a system that has previously worked well in other similar scenarios.

The following are the 3 main architectural patterns that we were able to find in the chapter regarding the Graphite system.

1. Client-Server

The first pattern we identified in the paper was the **Client-Server Pattern**. This is a common pattern that involves two main components: clients and servers. Clients make requests to the servers, while servers receive, process, and return responses to requests sent by clients.

While not clearly explicit in the paper, we believe this pattern is more explicitly present in sections 7.1 and 7.2, where it introduces the main components of the system and explains the back-end. Another clue could be the fact that Graphite's backend is built on Twisted, which "enables carbon [the server] to efficiently talk to a large number of clients and handle a large amount of traffic with low overhead".

In Graphite, this pattern is used to handle the communication between parties. In this case, the clients are the multiple monitoring agents that collect metrics and data, and send it to the server component. Carbon, the server component, with the help of whisper, stores this data so it can be later used to generate graphs.

2. Shared Repository

The **Shared Repository Pattern** was the second pattern found in the paper. It involves using a central data repository for communication between components in systems. The different components of the system have access to this repository and can store and/or retrieve information as needed.

Once again, this pattern is not explicit in the text, but using sections 7.2 and 7.3, which explain how the backend and frontend work, and also Figure 7.2 of the paper, it is possible to understand the presence of this pattern.

In Graphite, the Shared Repository Pattern is apparent in the way data is stored and accessed by multiple components using whisper. The carbon component uses whisper to store the data collected by the client applications, while the webapp uses whisper to read and create graphs from the data. Essentially, there is a centralized data storage that can be accessed by multiple components using whisper as an aid.

3. Broker

Lastly, the **Broker Pattern** was identified in the Graphite system. This pattern consists of introducing a middleman, i.e. a broker, to a system with the objective of decoupling components.

This pattern is not explicit in the chapters, but section 7.9.2 highly implies a broker. In section 7.9, the idea of clustering is introduced and it seems like a way to improve mostly, but not only, scalability. However, this created a problem when different clients sent data to servers. To fix this, a broker was introduced.

In Graphite, the pattern appears to be present in the tool carbon-relay, which is used to, through a series of rules and rule evaluations, correctly send data regarding certain metrics to the correct servers. This pattern ends up being a good complement to the Client-Server pattern when clustering becomes involved since it facilitates the distribution of metrics among various independent carbon instances.

Diagrams

There are two types of actors considered in this representation: ordinary users and clients, which represent machines or sensors responsible for the gathering of the required data, saved in the format of Data Points, which consist of a timestamp paired with a value.

The left part of the diagram represents the writing of data. First of all, a Client needs to establish a TCP connection with Graphite's backend daemon, Carbon, to enable the sending of data points. The carbon-cache is responsible for receiving such information and sending it to a server within the Whisper logical component. The carbon-relay receives data (equally to the carbon-cache part) but instead of storing it, this optional abstraction is responsible for applying a set of rules to it, identifying the servers where that information should be stored. In this way, the process of duplicating input can be achieved, though synchronization issues are not covered (thus not replication). When the data reaches the Whisper module, a write operation is triggered, and the system proceeds with verifying whether the write translates into a create or an update function. After that, the operation takes place on the File System.

On the other hand, Carbon isn't used in the process of reading data. The Webapp component is crucial to this procedure, as Users interact with it. The Composer UI communicates with the API Endpoint enabling the sending of requests to Whisper, where a read operation is triggered. If the file is found, it is fetched from the File System and the data points are returned to the Webapp via the API Endpoint. After that, the information is saved in the Memcached optional mechanism, implemented for performance purposes. This reduces the number of requests made to whisper since copies of data can be kept on the webapp storage. These can be passed to the Graph Generator that produces PNG images based on the points read, which are further displayed on the Composer UI.

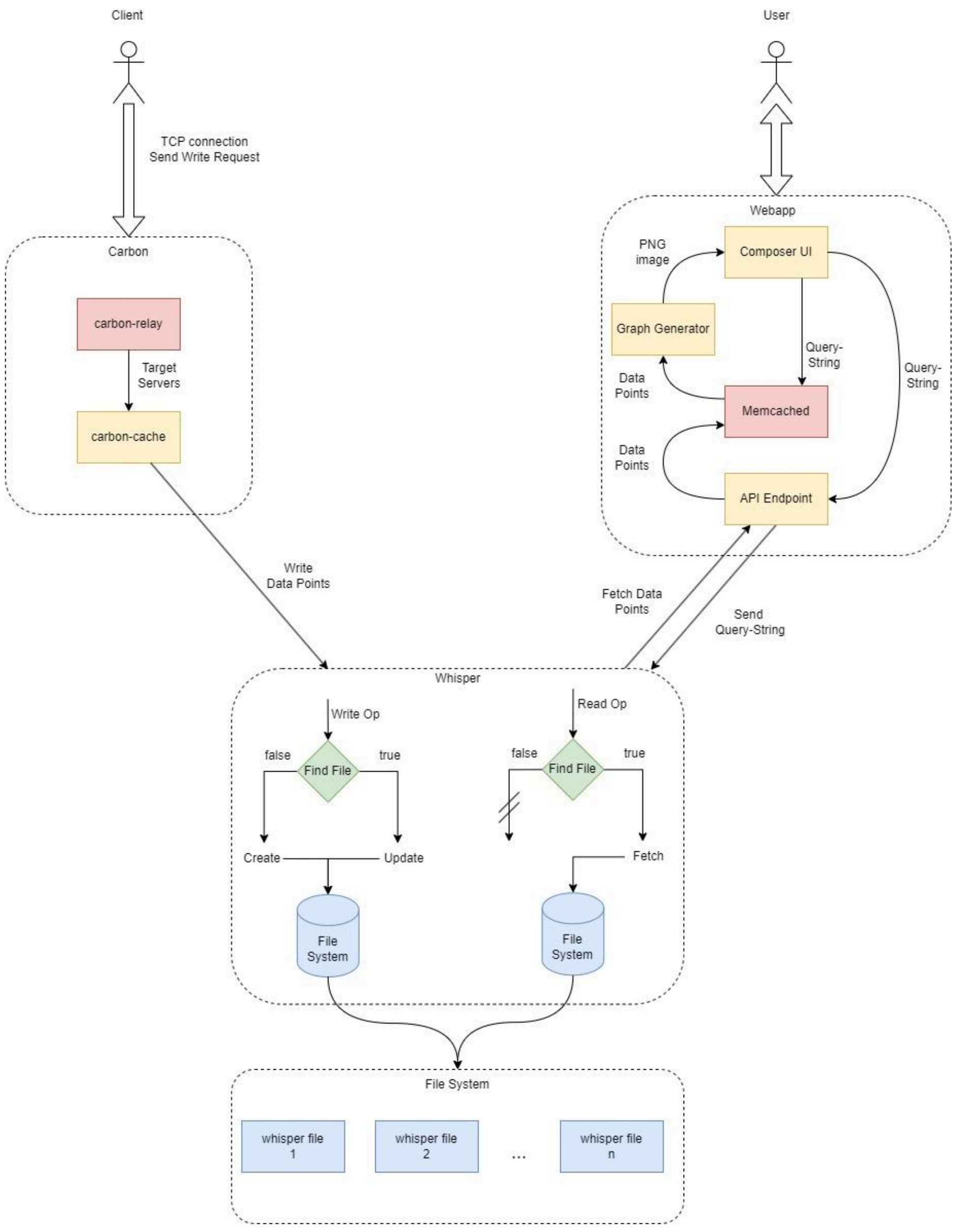

Figure 1. System diagram

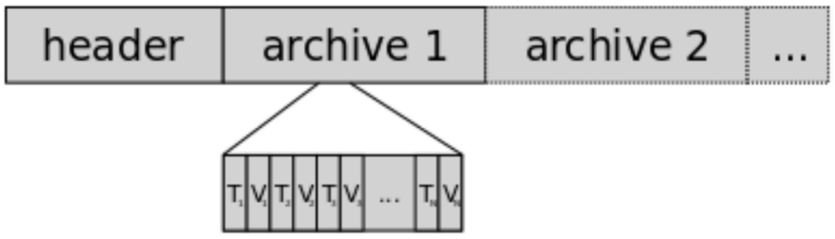

Figure 2. Structure of a Whisper File

Quality Attributes

The main quality attributes of Graphite were already explored in the overview and architecture patterns section to justify the choices of Graphite's architecture and design, but this section distills them further for easier comprehension:

- **Scalability**: Graphite is designed to scale both vertically and horizontally to accommodate large volumes of data. It can handle a high number of writes per second and can be clustered across multiple servers for load balancing and redundancy.
- **Performance**: Graphite's performance is a critical attribute, given its role in real-time data monitoring. It is optimized for fast data ingestion and retrieval, ensuring minimal latency in metrics reporting and graph rendering.
- **Reliability**: The reliability of Graphite is paramount, as it is often used in production environments where consistent uptime is required. Its modular architecture allows for parts of the system to fail without affecting the overall availability of the service.
- **Efficiency**: Graphite's components are designed to handle operations with minimal resource usage, ensuring that the system can manage large volumes of data without excessive consumption of computing resources.

These attributes are essential for Graphite's operation as a monitoring tool, ensuring that it can reliably store, process, and present time-series data at scale and high performance, with capabilities to provide its users with real-time visualization of its data.

Conclusion

In conclusion, Graphite's architecture embodies efficiency, scalability, reliability, and performance. Through **Whisper** for storage, **Carbon** for processing, and the **web application** for visualization, Graphite offers a robust platform for dynamic, data-driven dashboards. Utilizing key patterns like Client-Server, Shared Repository, and Broker, Graphite demonstrates adaptability and scalability. With its quality attributes of scalability, performance, reliability, and efficiency, Graphite proves its suitability for real-time data monitoring in diverse environments. Overall, Graphite stands as a testament to effective architecture, providing users with a reliable solution for managing time-series metrics with precision and efficacy.