Survey of software architecture V0.0

Name of system: Graphite
Reviewer: Claude Richard
Date: November 7th, 2011
Author of software: Chris Davis, initially at Orbitz Worldwide
Author of book chapter: Chris Davis
Five star rating of book chapter: **** (4)

Purpose of system:

Graphite's purpose is to do two simple things with numeric time-series data: store them, and display them in graphs. It must do this as a web application serving a large amount of clients, therefore performance was the primary issue when designing Graphite. It needs to be able to store large amounts of data as fast as possible, and use this data to render large amounts of graphs using data that is as recent as possible.

The project was started by Chris Davis when he was working at Orbitz Worldwide.

Basic metrics

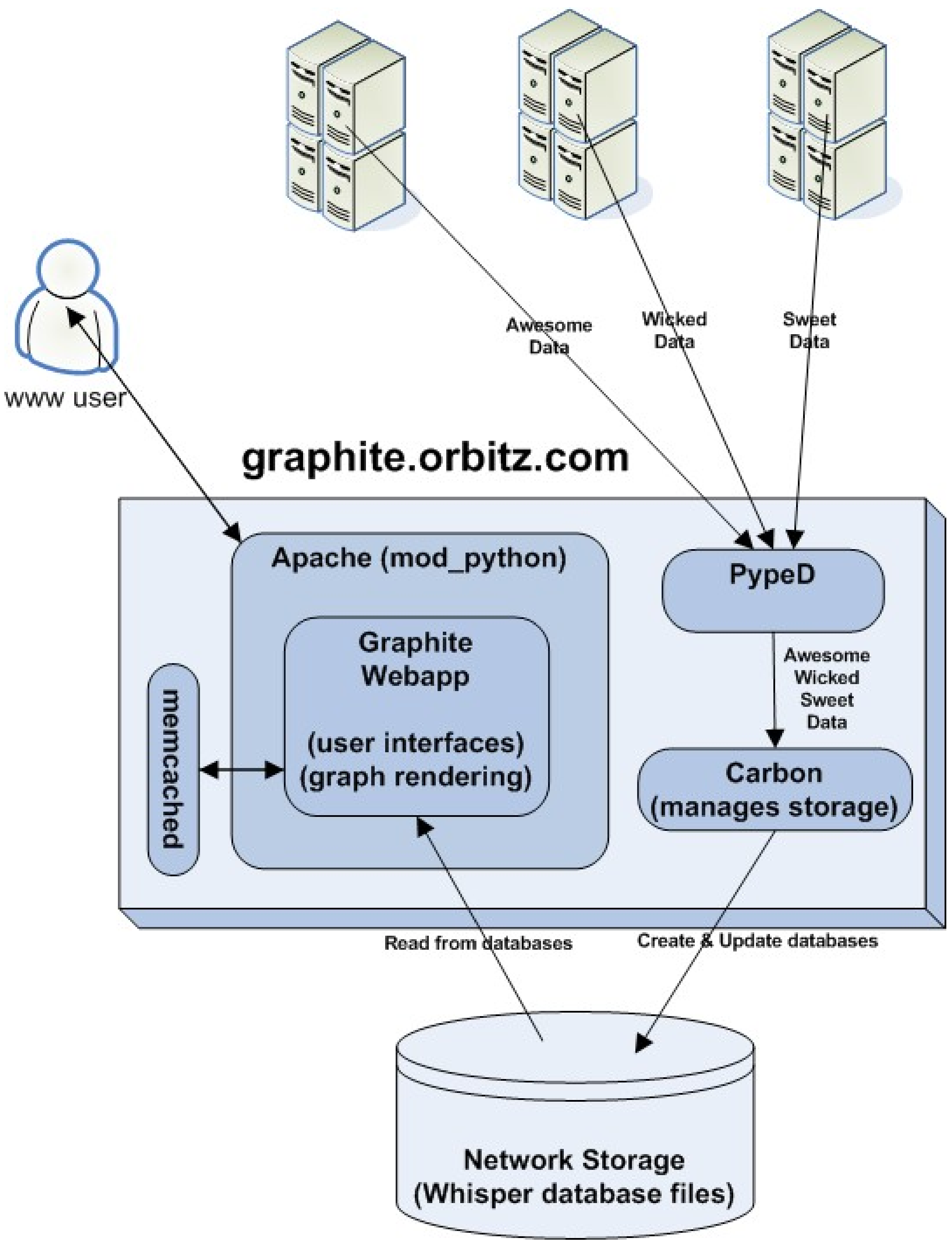
KLOC: 96.556 Javascript, 24.316 CSS, 12.816 Python
Project start-up: 2006
Number of major releases: 0 (version 1.0 planned for 2012)
Number of developers: 55 active members
Size of user community or number of installations: (not found)
Major stakeholders: Orbitz Worldwide
Use of concurrency: Carbon is split into 3 processes since Python has problems with multi-threading within the same process.
Implementation language: Python
Supporting software: Carbon is built on Twisted, ExtJS is used for the webapp, and the following are required in order to install the software:

- python2.4 or greater
- cairo (with PNG backend support)
- mod_python
- django
- python-ldap (optional - needed for ldap-based webapp authentication)
- python-memcached (optional - needed for webapp caching, big performance boost)
- python-sqlite2 (optional - a django-supported database module is required)
- bitmap and bitmap-fonts required on some systems, notably Red Hat

http://graphite.wikidot.com/installation

High level architecture

Diagram of software architecture



http://graphite.wikidot.com/high-level-diagram

The arrows denote data flow.

High level scenarios

Several computers together produce time-series data, and want to insert it into the server. They do this by calling methods in PypeD, which handles clustering of servers. PypeD then gives the data to carbon, which contains code to manage storage. Carbon, using whisper, writes the data to the database files.

When a user wants to access data in the form of a graph, she uses the Graphite webapp. She inputs a URL specifying the type of the data and its location. The webapp, again using whisper, reads the database files. The server produces a graph from that data according to the specifications in the URL, and sends the graph to the user for the webapp to display.

Data structures or algorithms

Whisper:

Whisper defines a special file format to store time-series data, and includes code to read from/write to these files.

Carbon:

Carbon listens for data from clients and tries to store it on disk as quickly as possible using whisper. There is a buffering mechanism that allows writing more data per write operation to the hard drive in order to reduce hard drive seeking times.

If Carbon uses too much RAM and therefore doesn't leave any RAM for the kernel for I/O buffering, then hard drive seeking times will increase insanely. In order to prevent this, there are configurable limits on cache size and rate-limits for whisper operations.

Webapp:

When retrieving data, the webapp gets whisper data stored on disk, as well as data buffered in carbon, and combines the two in order to get data that is as recent as possible. It uses this combines data to make the graph that the user requested. The webapp also caches graphs in the web server for a small amount of time to avoid rendering too many identical graphs.

Control flow and/or data key to the architecture if any

The data flow is more or less one-directional: Client -> carbon -> whisper files -> webapp -> user. The only modification of the data happens in the graph rendering in the webapp. The data is permanently stored in whisper files so that it can later be read by the webapp (possibly by several users).

Architectural style:

One-directional data flow:

See control flow section.

Layered:

The client that makes requests to store data sends data to the database server. Carbon handles these requests to store data for multiple clients, and uses whisper to store the data in the filesystem of the server. Therefore, there is a layering hierarchy: Carbon -> Whisper -> Filesystem. Clients can only access Carbon, by sending strings of data to the server such as this one: "Servers.www01.cpuUsage 42 1286269200". Similarly, the webapp uses whisper to read the filesystem, so the hierarchy is: Webapp->Whisper->Filesystem. Again, a user cannot access Whisper or the filesystem directly.

Polymorphism:

Some polymorphism is used for the *find* method in Whisper in order to be able to store data in different formats, e.g. Gzipped Whisper files.

Major evolutionary changes:

The author designed the architecture of Graphite gradually. As performance problems arose, he looked for improvements in design rather than low-level optimization.

Performance bottlenecks:

Rendering a graph is the most expensive part of a graph request. Therefore, a bottleneck can occur if a webapp server receives large amounts of graph requests. The author experienced this bottleneck once his users starting using dashboards, and he dealt with it by caching the graphs on the server and making them expire after a few seconds. These few seconds alleviated a large burden from the server, since the majority of graph requests for dashboards were for graphs that were already rendered a few seconds before.

If carbon can't write all the data it receives fast enough, it will cause problems. A buffering mechanism in Carbon is used to allow each write operation to write more data, thereby reducing hard drive seek times and improving the writing rate that Carbon can handle.

Real time:

I/O operations by carbon on the servers must be as fast as possible. Carbon has various tweaks in design to accomplish this.

The webapp servers must be able to produce large amounts of graphs for the users. Some tweaks in design are used for this as well, for example caching the graphs for a small amount of time to prevent excessive re-rendering of identical graphs.

Notation for architecture: Box and arrow
Methodology: not clear

Appendix:
Kruchten's eight context attributes applied to Brown/Wilson systems

Kruchten attributes described in his slides 17-24 of
https://files.me.com/philippe.kruchten/sbz0ma
Also in: https://files.me.com/philippe.kruchten/1q00nw

1. **Size:** Small (12 to 13 KLOC)
2. **Criticality:** Med (company critical
3. **Age of system:** M (startup in 2006)
4. **Rate of change:** Med
5. **Business model:** Started In-house (Orbitz Worldwide hired Chris Davis as a contractor to make this software), became Open-Source in 2008
6. **Stable architecture:** Lo (stable)
7. **Team distribution:** Lo in early years, now VH since it's open-source
8. **Governance:** Lo