Cross Over

High Level requirements

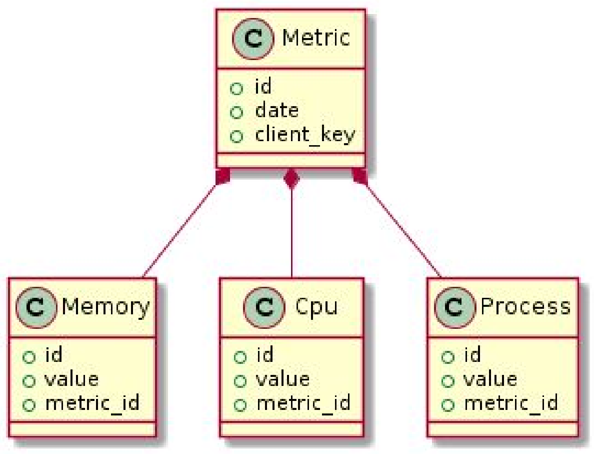
The system is composed of two parts; one client application running on the monitored machines and one server application, receiving the data from the clients.

Client: The client must gather the system usage: percent CPU usage, percent RAM usage and the quantity of processes currently running. This analyze must be done every 5 minutes, and send to the server application.

Server: The server must reception all the clients' information and must be able to support at least 100 of clients. It must store the system usage into a Relational Database and send an email to the client if their system usages is superior to their configured limits.

High Level presentation of the data model

Database

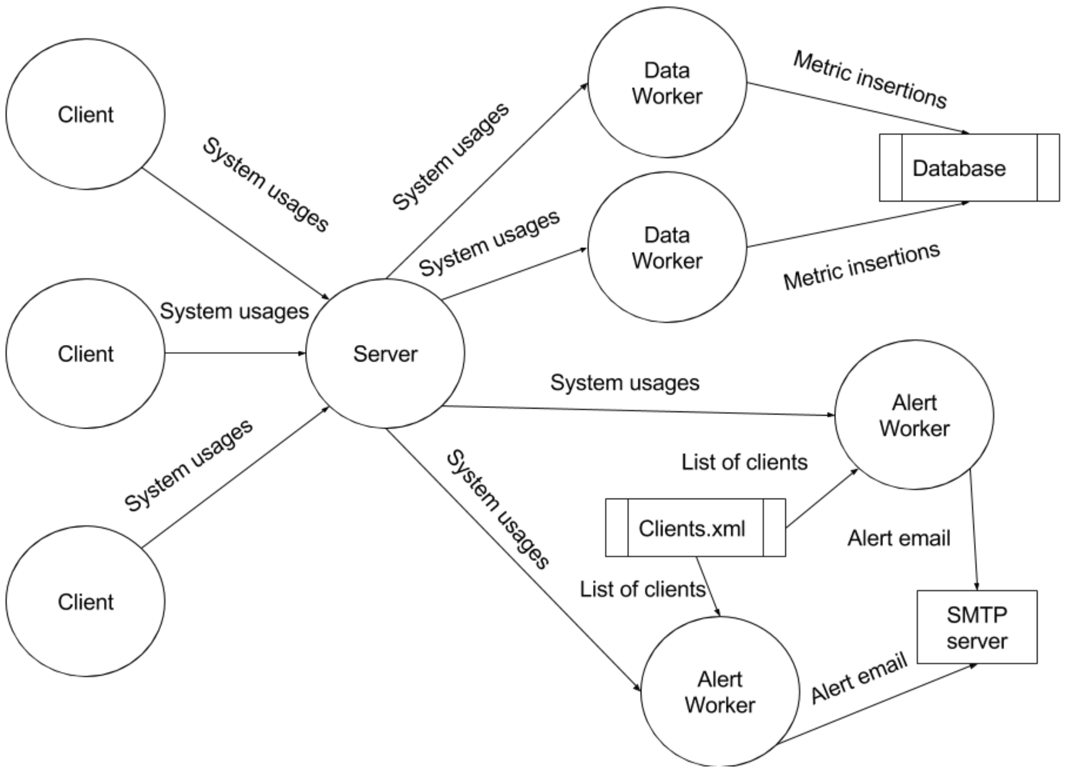


Metric: A metric unit describe a time T system usages for a client X.

Memory, Cpu, Process are 3 kinds of system usages. Each of them has an unique data and belong to a metric unit.

One of the interesting part in this design, is its evolutive style. At this moment we only support 3 kinds of system usage, but in the future we might need more.

Architecture



There are 3 main task in the project: Sending the system usages, saving those usages, sending an alert if required.

The client are the producer of this system, they will measure their system usages and then send those information to the server.

The server is composed in 3 part, and has 2 main task: saving the system usages and sending an email. It is designed as a microservices, with a proxy system, called: Server; this server will forward a system usage to a DataWorker and to a AlertWorker.

The DataWorker will be in charge to update the database with the received system usage.

The AlertWorker will find the client configuration matching with the received message, and verify if the system usage are below of the configured limits. If the system usage are above the limits, it will ask the SMTP server to send an email.

Detailed Architecture

Load balancing

This project was designed to be scalable, technically the Workers can be separated into independent processes on several machine. The only restrictions will be to be able to connect to the Server.

Each kind of Workers bind on one specific port, I used a design pattern of PUSHER / PULLER to provide a load balancing between the running instances.

Serialization

To exchange the data between the clients, the server and the workers; I used the ProtocolBuffer. I choose this technology to reduce the amount of data transferred between the clients and the server.

This system is a closed communication between computers, nobody should have to read the transferred data; hence a protocol binary can be used.

Moreover it does provide a backward compatibility, so if we desire to add another system usage, we could add another field and we won't require to update the running clients.

Threading

The server is currently designed in a standalone version, with an embedded set of DataWorkers and AlertWorkers.

I used a thread pool to run the workers instead of a dynamic creation of workers. I choose this option to limit the system resources. Each worker will open one socket to listen the Server and will open another file descriptor either to communicate with the Database or to read the content of clients.xml.

A thread pool will avoid to crash our system if we are under an high pressure.