System Activity Monitor

Architecture Design Document

Juan Belón Pérez

Computer Engineer

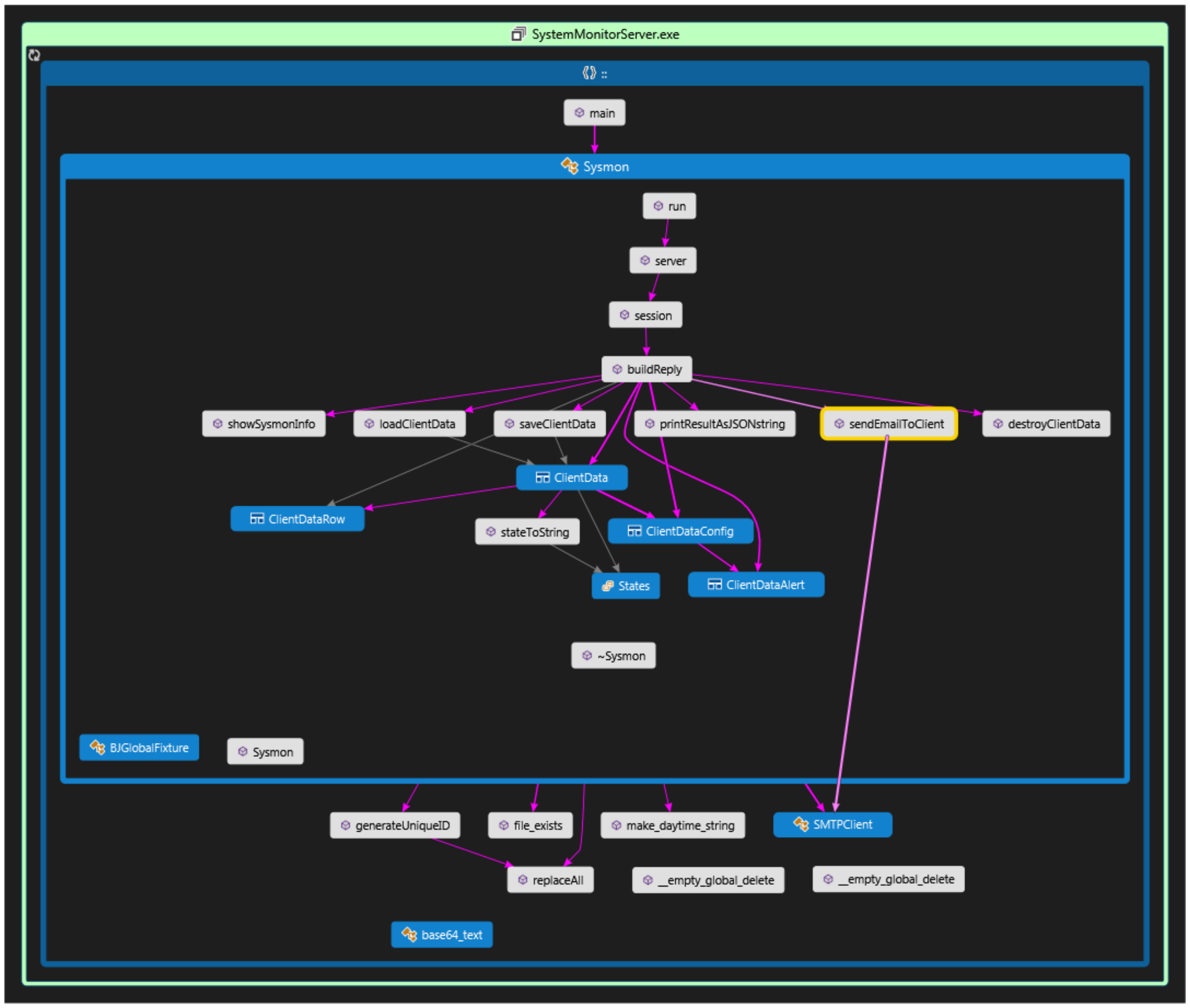
hola@programadorphp.org

1. Server web service design

This Code map is useful to understand the flow of the server program. We call via console command :

./sysmon_server <PORT> &

and it will run a loop waiting for incoming connections from clients, for each new socket the function session operates in an infite loop until the peer is closed, building replies to the incoming messages of the clients.



The first message to process from the client in the server, is "create", that will return to the client an unique session identificator used to store data in a single file that is the database for this client from this moment, until the socket connection is closed. The data is saved in binary format, but it's generated in the client and sent in a json string using the http protocol.

The server is designed to be installed on a single central machine in the same intranet, but it could be anywhere on the internet. It receives the statistics from the client machines and stores them into a relational database along with client key, that is retrieved from the XML of clients configurations. This database is a single file <session_id>.dat

Example of XML of clients configurations:

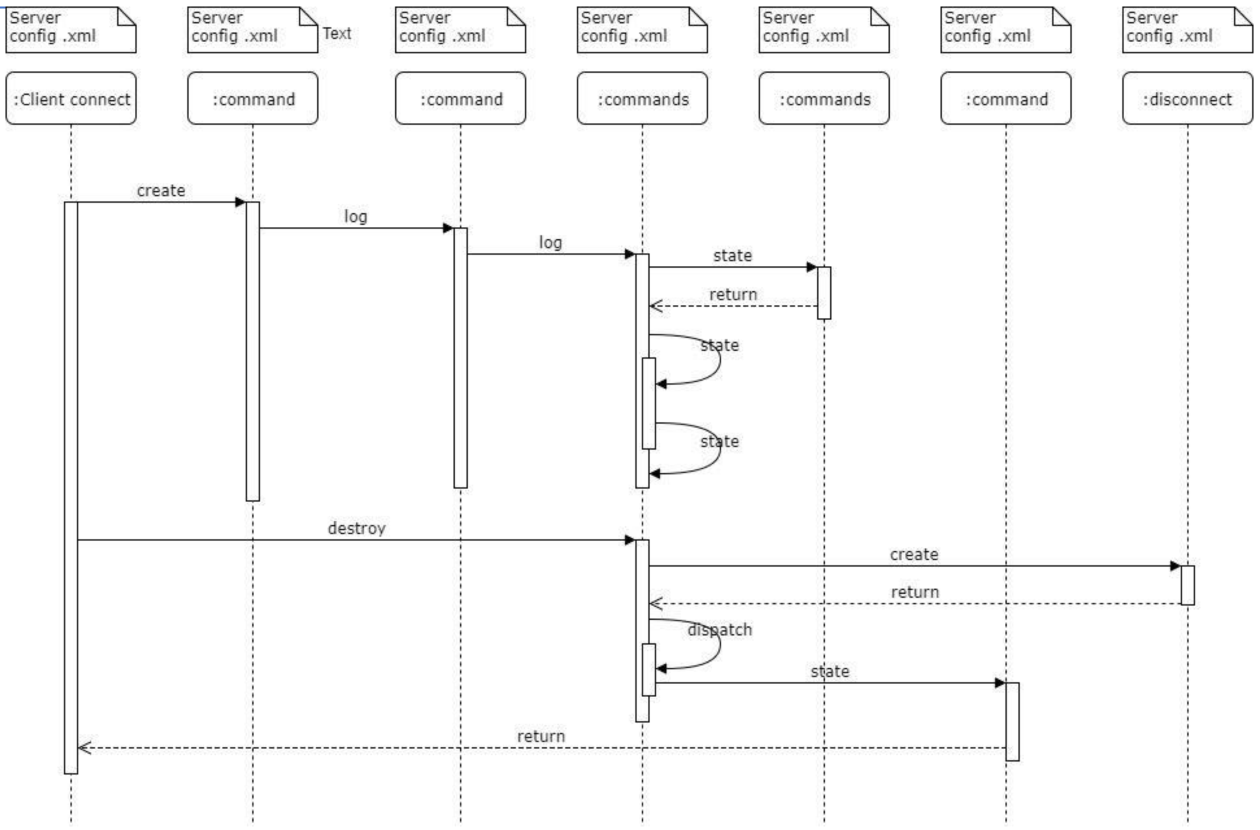
```
<client key="xxx" mail="asa@asda.com">
  <alert type="memory" limit="50%" />
   <alert type="cpu" limit="20%" />
   <alert type="processes" limit="50" />
</client>
```

limit - means the value beyond which the alert is triggered.

The server receives data collected by multiple clients and based upon the "alert" configuration of each client sends a mail notification. The notification is sent to the client configured email address using SMTP. Use a simple text mail format with some detail about the alert.

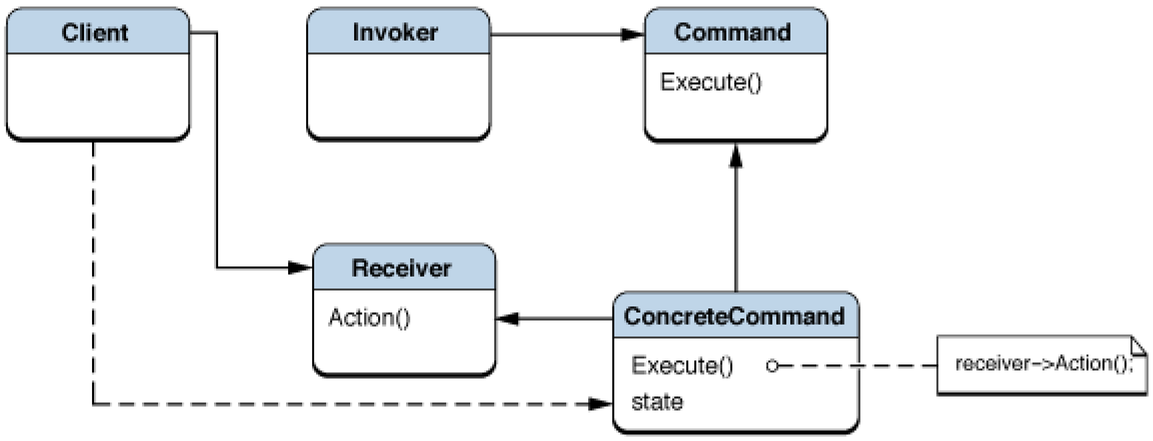
To achieve this functionality each session will read the xml file to store the new config parameters of the client ,not only using the key to authenticate but the list of alerts.

In the sequence diagram we can watch what happens in a normal run of the clients against the server. The server is listening, a client connects ,the server reads the config xml file for the key and creates a session, it returns the session and change the state of the client and store it in a database file, the client can now use a different command, log, the server saves the data in the database, using the unique session id , this goes on until the clients disconnects from the server, the next time a client will use a new session id, we can search for all the sessions of a client using the config key and do other things if we want with the data, filter, process,and so on.



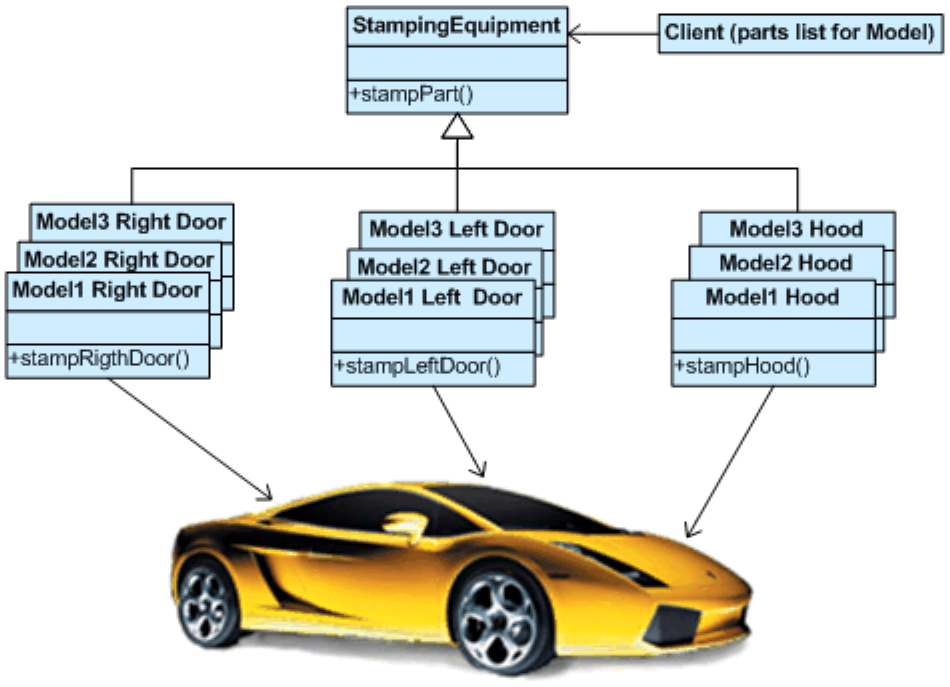
2. Command Factory Pattern

Because of the behaviour of the alerts parameters are very similar, we could use a factory pattern with an interface <Command> that requires to implement the shared methods needed to check the client incoming statics against the configuration alerts of the xml, so let's check this CommandFactory in the diagram



The client is asking to the server for an action, the first one is to create a session, then , through the CommandFactory we can choose what action execute in the server, and then send back a reply to the receiver (client).

The problem is that it gets complicated to maintain and change and it’s hard to scale, so, it could be better if we just use an abstract factory of types of commands that we can replace?



Maybe, but only if we change the nature of the commands, for example, instead of limit, add a different type of operation.

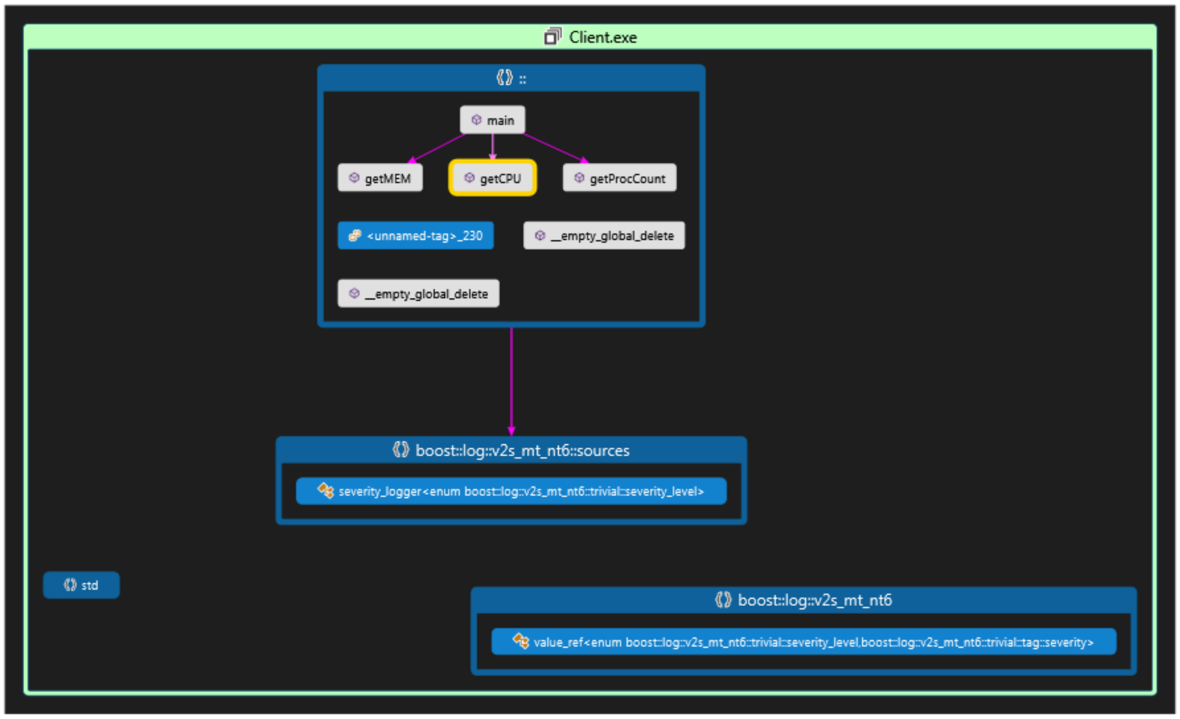
3. Model View Controller

You are used to see this design pattern in websites,but we are using it all the time for Input -> **Insertions/Modifications/Reads -> Outputs** just to agregate data to the database.

Using a JSON we can choose the command, for example, {“action”: “log”, “type”: “cpu”}, this simple string represents a controller action that will trigger a model operation and will return another view as result (log).

4. Client program

This is code diagram for the client


Client.exe
::
main
getMEM
getCPU
getProcCount
<unnamed-tag>_230
__empty_global_delete
__empty_global_delete
boost::log::v2s_mt_nt6::sources
severity_logger<enum boost::log::v2s_mt_nt6::trivial::severity_level>
std
boost::log::v2s_mt_nt6
value_ref<enum boost::log::v2s_mt_nt6::trivial::severity_level,boost::log::v2s_mt_nt6::trivial::tag::severity>

How does it works?

It will ask you for a server address, a port, and a key.

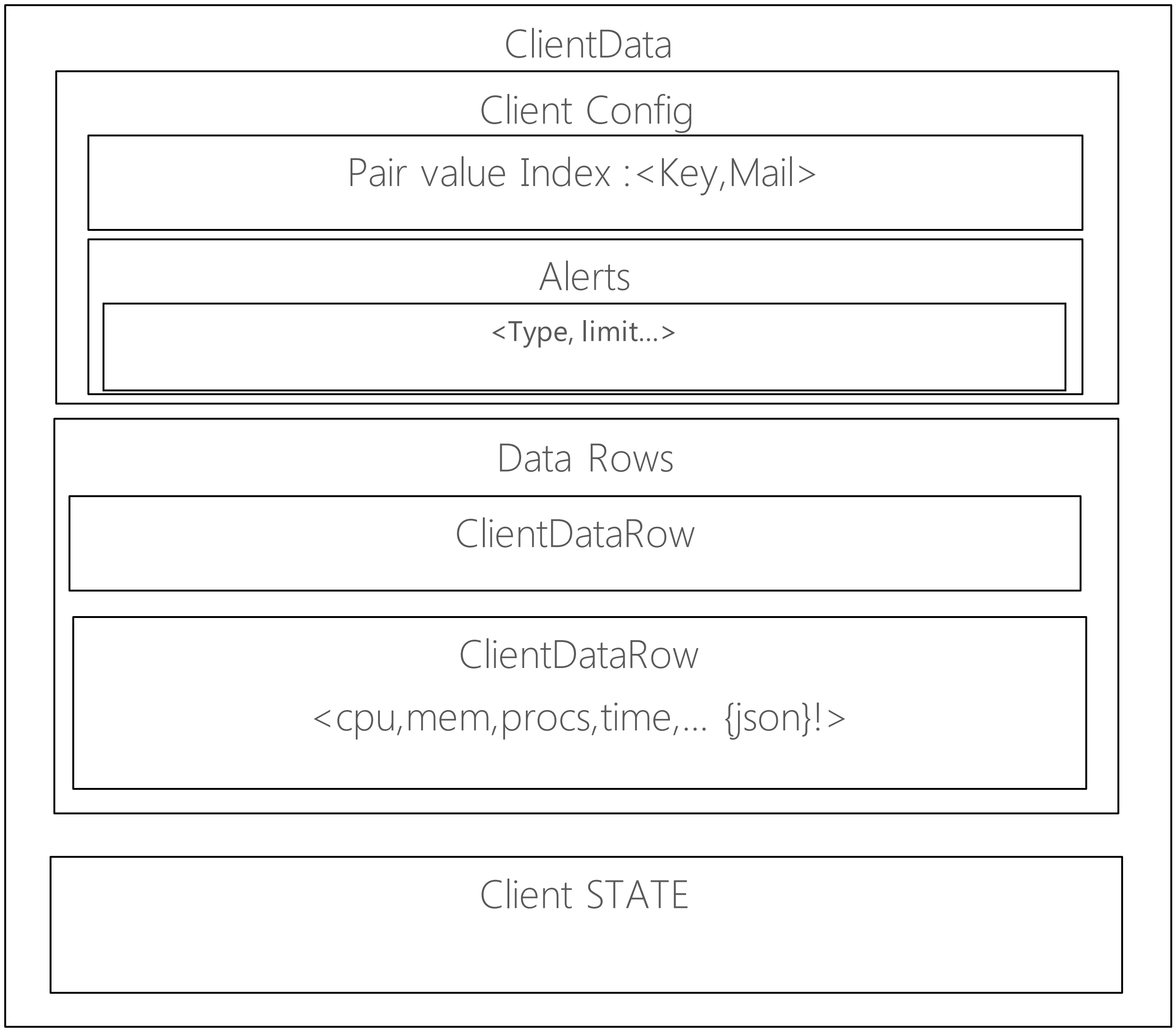
Using the command line it will be something like this:

./sysmon_client <HOST> <PORT> <KEY>

Once you run the program it will try to connect to the host on that port and sends the key with a "create" action in a json to the server. If everything goes ok, the server will reply with an unique session id that represents the name of the database only for this session where it will be stored all the lines of the time spaced logs of statics.

5. Client Data

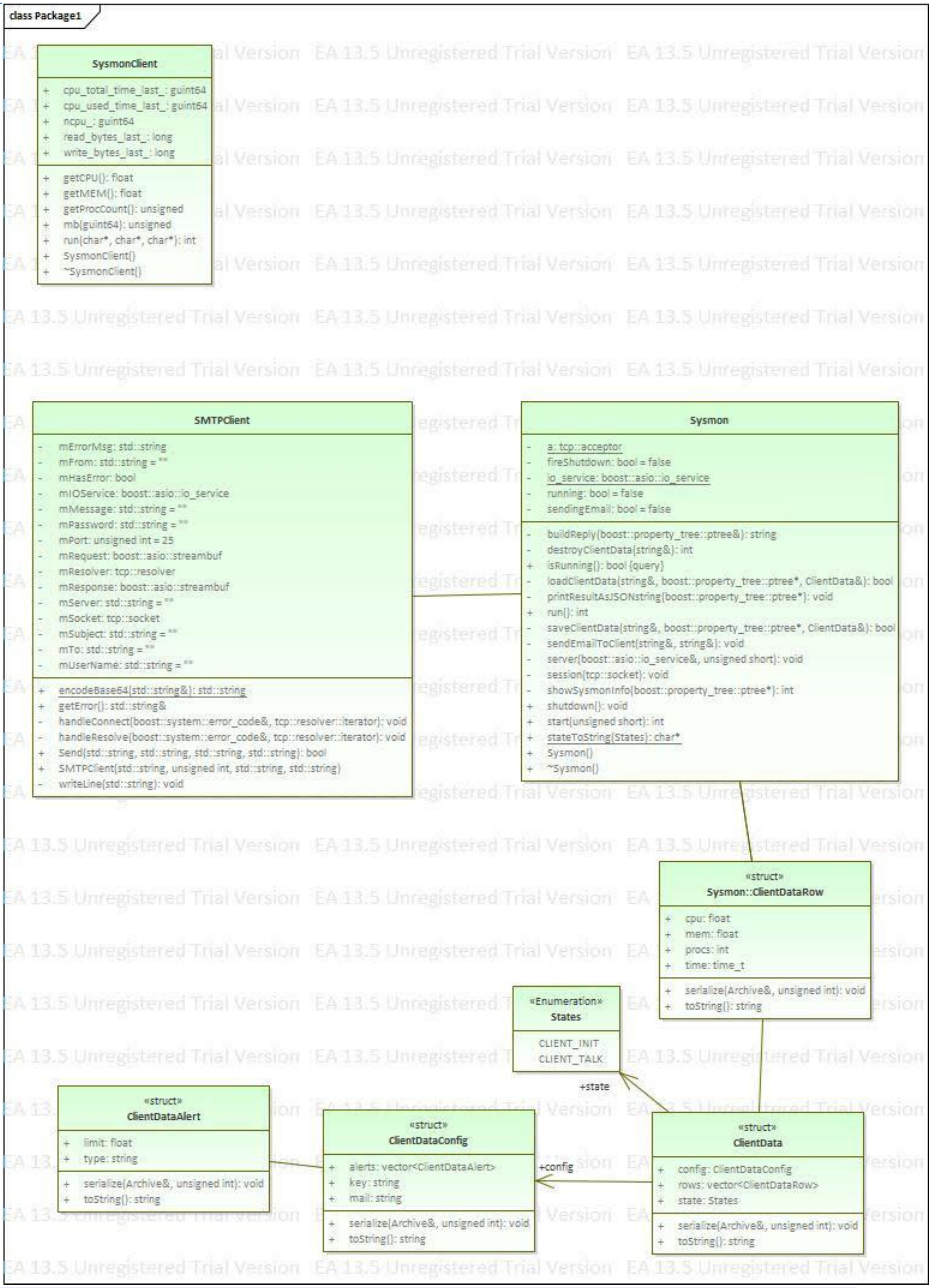
The data of the client stored in each file as database per session is composed of



6. Source choices

UML Diagram of the classes used in the source code (C++)

We have the data model represented for all the most important pieces of the client/server of the system monitor architecture.


class Package1
SysmonClient
+ cpu_total_time_last_: guint64
+ cpu_used_time_last_: guint64
+ ncpu_: guint64
+ read_bytes_last_: long
+ write_bytes_last_: long
+ getCPU(): float
+ getMEM(): float
+ getProcCount(): unsigned
+ mb(guint64): unsigned
+ run(char*, char*, char*): int
+ SysmonClient()
+ ~SysmonClient()
SMTPClient
- mErrorMsg: std::string
- mFrom: std::string = ""
- mHasError: bool
- mIOService: boost::asio::io_service
- mMessage: std::string = ""
- mPassword: std::string = ""
- mPort: unsigned int = 25
- mRequest: boost::asio::streambuf
- mResolver: tcp::resolver
- mResponse: boost::asio::streambuf
- mServer: std::string = ""
- mSocket: tcp::socket
- mSubject: std::string = ""
- mTo: std::string = ""
- mUserName: std::string = ""
+ encodeBase64(std::string&): std::string
+ getError(): std::string&
- handleConnect(boost::system::error_code&, tcp::resolver::iterator): void
- handleResolve(boost::system::error_code&, tcp::resolver::iterator): void
+ Send(std::string, std::string, std::string, std::string): bool
+ SMTPClient(std::string, unsigned int, std::string, std::string)
- writeLine(std::string): void
Sysmon
- a: tcp::acceptor
- fireShutdown: bool = false
- io_service: boost::asio::io_service
- running: bool = false
- sendingEmail: bool = false
- buildReply(boost::property_tree::ptree&): string
- destroyClientData(string&): int
+ isRunning(): bool {query}
- loadClientData(string&, boost::property_tree::ptree*, ClientData&): bool
- printResultAsJSONstring(boost::property_tree::ptree*): void
+ run(): int
- saveClientData(string&, boost::property_tree::ptree*, ClientData&): bool
- sendEmailToClient(string&, string&): void
- server(boost::asio::io_service&, unsigned short): void
- session(tcp::socket): void
- showSysmonInfo(boost::property_tree::ptree*): int
+ shutdown(): void
+ start(unsigned short): int
+ stateToString(States): char*
+ Sysmon()
+ ~Sysmon()
«struct»
Sysmon::ClientDataRow
+ cpu: float
+ mem: float
+ procs: int
+ time: time_t
+ serialize(Archive&, unsigned int): void
+ toString(): string
«Enumeration»
States
CLIENT_INIT
CLIENT_TALK
+state
«struct»
ClientDataAlert
+ limit: float
+ type: string
+ serialize(Archive&, unsigned int): void
+ toString(): string
«struct»
ClientDataConfig
+ alerts: vector<ClientDataAlert>
+ key: string
+ mail: string
+ serialize(Archive&, unsigned int): void
+ toString(): string
+config
«struct»
ClientData
+ config: ClientDataConfig
+ rows: vector<ClientDataRow>
+ state: States
+ serialize(Archive&, unsigned int): void
+ toString(): string
EA 13.5 Unregistered Trial Version