GridAdmin

Dan Alexandru

Faculty of Computer Science, Iasi, Romania
dan.alexandru@info.uaic.ro,
web home page: https://github.com/xR86/rc-project

Abstract. GridAdmin offers a solution for orchestration of VMs - both on managed servers and cloud servers (like AWS), with a client-server architecture, the server acting as a command and control server, and the client offering a web interface...

Keywords: networking, Linux, C, Python, AWS

1 Introduction

GridAdmin offers a solution for network administration by sending commands through a Python GUI to a C master server, that uses ssh to pass the commands to the slave machines (or respective API available for cloud platforms).

Targeted functionality similar to Python's Fabric (http://www.fabfile.org/).

2 Technologies used

2.1 Server-side

The project uses C for the concurrent TCP server. Use of libssh (https://www.libssh.org/) will be employed on the server (master), along with openssh-server on the slave.
If needed, the C server could be extended with python modules.

2.2 Client-side

The project uses Python and PyQt4, the interface being modeled with web technologies such as HTML, CSS, Javascript, Angular.js. This will grant easier interface creation, and a more rich interface, (example of frameworks doing this: Apache Cordova - https://cordova.apache.org/ / Ionic, Electron). One example of web app that runs on desktop is Slack (https://slack.com/downloads/windows).

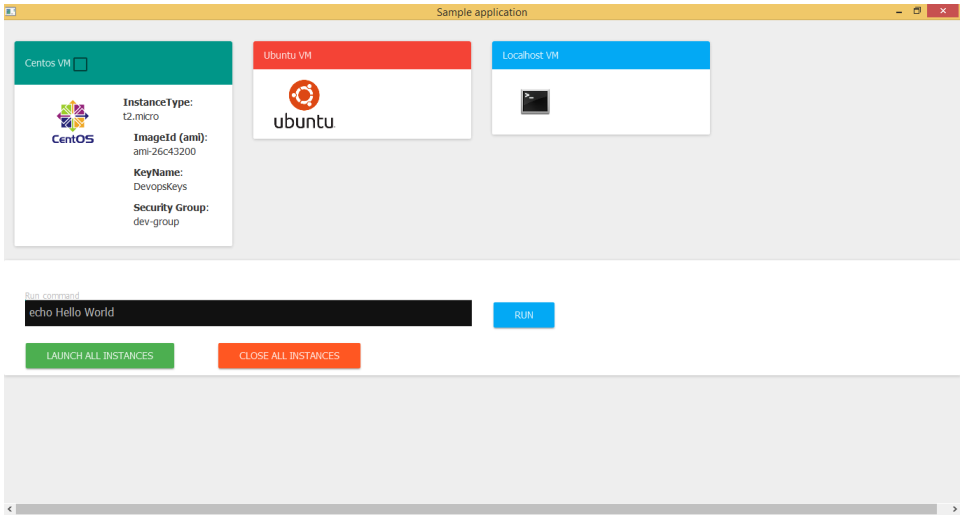

Fig. 1. Client demo

3 Application Architecture

The application makes use of the client-server model for interfacing and master-slave model for control.

3.1 Client-server model

In this project, the **client** is the Python GUI that can be remote or on the same machine as the server. The client connects to the C **server** to run bulk commands (to multiple machines) or to run one or more commands on one machine.
The server will ssh to various machines, execute commands from client and return the output to the client.

3.2 Master-slave model

In this project, the **master-slave model** acknowledges that there is a server that has ssh access to various machines (public-key authentication and/or password authentication), and can run almost anything on them (if root mode allowed in **sshd_config**).

3.3 Application layer diagram

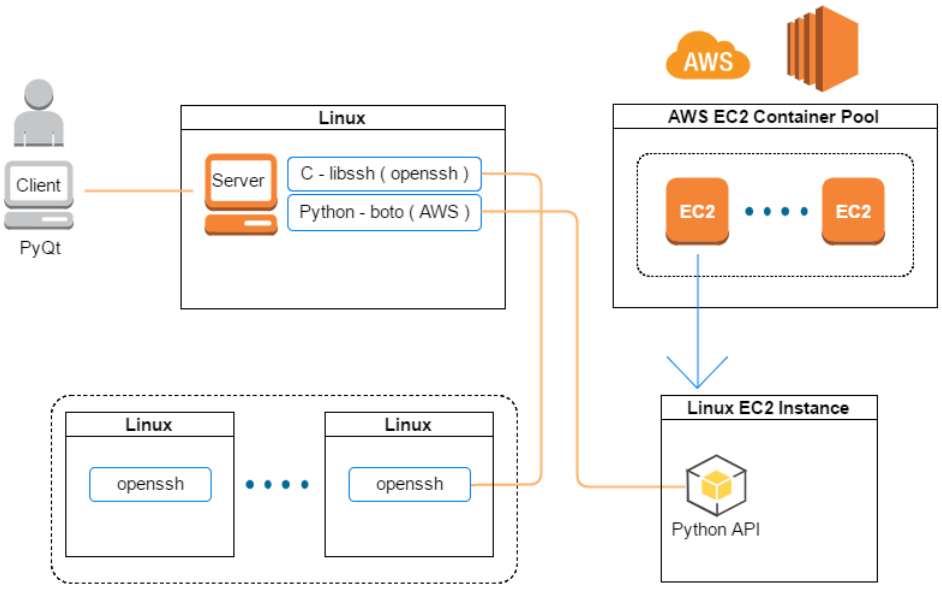

Fig. 2. Application Layer architecture - The client exchanges data with the server, that will either be C + Python (extending) or C and Python separately (libssh and boto) , the server then connects either to various machines (with openssh pre-installed) or to a AWS EC2 instance (through the API).

4 Implementation details

4.1 Application flow

The application flow is:

- user deploys various instances/servers
- user saves a config file and deploys the master server. The config file will contain **IP:user:password/pubkeyName**, and may contain the following machine types:
  - containers (eg: Docker)
  - VMs - ssh to host, then use VBoxManage to port forward (and/or use bridged mode) and make a mapping (and use the API) between forwarded ports and desired VMs (eg: **VBoxManage modifyvm rc-box1 –natpf1 "ssh,tcp,,3022,,22"** - for VM called "rc-box1", forward guest port 22 to host port 3022) - automated through Vagrant
  - servers
  - EC2 instances (or equivalent)
- user then opens the GUI client, and checks initially the status of all the machines in the config file

– master server runs: **ssh-copy-id <remoteuser>@<remotehostIP >** (meaning, it exchanges keys with the remote slaves, so it could establish the ssh connection later)
– master server checks the status of the machines
– after this, you could send various commands within the interface to selected machines, and have the differentiated output and logs sent back to you

4.2 Relevant code

Some of the relevant code snippets for this project will be explained here.

The work will be split in 3 parts

– client
– server and configs (e.g.: **sshd_config** file)
– automatization (Vagrant, APIs, supporting different archetypes of machines)

```
# establishing VM variables
api_version = '2'
provider   = 'virtualbox'
box        = 'ubuntu/trusty64'
hostname   = 'rc-box'
ip         = '192.168.50.13'
ram        = '2048'

# configuring Vagrant deployment
Vagrant.configure(api_version) do |config|
  # configure variable
  config.vm.define "rc" do |rc|
  end

  # creating new VM with specified parameters
  config.vm.box = box
  config.vm.host_name = hostname
  config.vm.network :private_network, ip: ip
end
```

Fig. 3. Vagrantfile snippet - automating VM deployment

```c
int main() {

    ssh_session session; // creates a new session (connection)
    ssh_channel channel; // creates a new channel (data)
    int rc, port = 22; //status flag and port variable
    char buffer[1024]; //buffer for ssh responses
    unsigned int nbytes; //

    printf("Session...\n");
    session = ssh_new();
    if (session == NULL) exit(-1);

    ssh_options_set(session, SSH_OPTIONS_HOST, "localhost"); //localhost
        //192.168.100.2
    ssh_options_set(session, SSH_OPTIONS_PORT, &port);
    ssh_options_set(session, SSH_OPTIONS_USER, "xR86"); //xR86 //labsi

    printf("Connecting...\n");
    rc = ssh_connect(session);
    if (rc != SSH_OK) error(session);

    printf("Password Autentication...\n");
    //rc = ssh_userauth_password(session, "NULL", "password"); // second
        is username, third is password
    rc = ssh_userauth_publickey_auto(session, "xR86", NULL);
        //automatically looks in .ssh/ keys
    if (rc != SSH_AUTH_SUCCESS) error(session); // if auth fails, the
        program stops here

    printf("Channel...\n");
    channel = ssh_channel_new(session); //connection established
    if (channel == NULL) exit(-1);

    printf("Opening...\n");
    rc = ssh_channel_open_session(channel);
    if (rc != SSH_OK) error(session);

    printf("Executing remote command...\n");
    rc = ssh_channel_request_exec(channel, "ls -l"); //system call in
        remote machine
    if (rc != SSH_OK) error(session); //cmd failed

    printf("Received:\n");
    nbytes = ssh_channel_read(channel, buffer, sizeof(buffer), 0);
    while (nbytes > 0) {
        fwrite(buffer, 1, nbytes, stdout);
        nbytes = ssh_channel_read(channel, buffer, sizeof(buffer), 0);
    }

    free_channel(channel); //EOF, close and deallocate
    free_session(session); //disconnect and deallocate memory

    return 0;
}
```


Fig. 4. C libssh snippet

5 Conclusions

The solution could be extended to be:

- **cloud provider agnostic** (especially IaaS), offering support for Google Cloud Platform and/or Microsoft Azure)
- duplicating the **server and hosting it in different geographical locations** and infrastructure backbones - for availability / legal reasons
- offering **specific commands in the interface** such as on-the-fly snapshots, backup rotation of logs (backup rotation scheme), bulk close/sleep/open, grouping after type (VM/instance/etc)

.

Solution will be made available at the following address:

`https://github.com/xR86/rc-project`