Relatório da Etapa 2 do trabalho de Linguagens de Programação

José Luiz - Pedro Graça

UFRJ, 13 de Novembro de 2020

Introdução

O software é um sistema de gerenciamento de máquinas. Este é composto por 4 partes:

Gerente da infra estrutura: É o usuário que, através do programa, controlar as máquinas conectadas remotamente.

FrontEnd: Interface de texto que permite o controle dos clientes por parte do servidor.

Backend: Servidor que submete comandos aos clientes e recebe o resultado destes comandos.

Cliente: Qualquer máquina que possui o software instalado e está se comunicando com o servidor.

Casos de Uso

Os casos de uso estão de acordo com as opções solicitadas pelo gerente no frontend. Dessa forma vamos orientar os casos de uso de acordo com cada uma das opções de execução oferecido pelo programa.

Listar clientes:

Retorna a lista de todos os clientes conectados. Essa opção é executada quando o gerente informa, selecionando pelo número da função chamada de “FrontListClients”.

Abrir shell:

Abre um shell no cliente para o uso de comandos pelo gerente. É necessário informar o número de identificação do cliente, qual endereço rodará rodará o shell e a porta de conexão com esse endereço.

Essa opção é executada quando o gerente informa, selecionando pelo número da função chamada de "FrontPopShell" e ele informa corretamente (quando solicitado) os dados imediatamente supracitados.

Enviar arquivo:

Copia um arquivo do servidor para um cliente especificado. É necessário informar o nome do arquivo (o caminho relativo, com relação ao binário, pode ser informado) e o número de identificação do cliente. Ele copiará o arquivo informado e jogará no arquivo uploads.

Essa opção é executada quando o gerente informa, selecionando pelo número da função chamada de "FrontSendFile" e ele informa corretamente (quando solicitado) os dados imediatamente supracitados.

Receber arquivo:

Copia um arquivo do cliente para o servidor. É necessário informar o número de identificação do cliente e informar o nome do arquivo, este terá que estar dentro da pasta files. Se o gerente desejar solicitar um arquivo do cliente fora desse diretório ele terá que “Abrir shell” desse cliente e copiar o arquivo para essa pasta files fixa.

Essa opção é executada quando o gerente informa, selecionando pelo número da função chamada de "FrontReciveFile" e ele informa corretamente (quando solicitado) os dados imediatamente supracitados.

Executar programa:

Roda um terceiro executável do cliente para o servidor. É necessário informar o número de identificação do cliente e informar o nome do binário, este terá que estar dentro da pasta files.

Essa opção é executada quando o gerente informa, selecionando pelo número da função chamada de "FrontExecProgram" e ele informa corretamente (quando solicitado) os dados imediatamente supracitados.

Instalar persistência:

Tenta copiar o próprio cliente para a pasta de inicialização do sistema para garantir que o programa sempre rodará ao iniciar a máquina. É necessário informar o número de identificação do cliente.

Essa opção é executada quando o gerente informa, selecionando pelo número da função chamada de "FrontInstall” e ele informa corretamente (quando solicitado) os dados imediatamente supracitados.

Ajuda:

Informa todos os comandos existentes e os explica.

Essa opção é executada quando o gerente informa, selecionando pelo número da função chamada de "Help”.

Exit:

Termina a execução do programa.

Essa opção é executada quando o gerente informa, selecionando pelo número da função chamada de "Exit”.

Implementação

Backend - respostas TCP asíncronas:

O servidor http jamais retorna uma resposta sem nenhum comando para o cliente. Isto é pensado de forma a evitar a técnica de pooling e consequentemente o desperdício de banda pelo cliente. Para isso ocorrer, utilizamos uma abordagem semelhante ao problema de concorrência dos produtores e consumidores.

```
    def getCmd(self):
        self.cvcmd.acquire()
        while self.cmd==None:
            time.sleep(1)
            self.cvcmd.wait()
        #consume cmd
        result=self.cmd
        self.lastcmd=result
        self.cmd=None
        self.cvcmd.release()
        return result
def setCmd(self,cmdList,fname=None):
        self.cvcmd.acquire()
        #produce cmd
        self.cmd=cmdList
        self.cvcmd.notify()
        self.cvcmd.release()
```

O servidor sempre que o cliente pede por um comando, chama este método da class cliente, mas ele espera que o comando seja válido para retornar fazendo a resposta http esperar por tempo indeterminado.

Cliente e backend - autenticação JWT

Utiliza-se um *json web token* para garantir que apenas usuário autorizados pelo administrador possam fazer parte da rede. O JWT é um token em json, mas assinado pelo servidor através do algoritmo HS256, impedindo que dispositivos não autorizados forgem um token válido, desde que a senha para a assinatura seja segura.

```
JWT_SECRET="secret" #implementação server side
def encodeJwt(token):
    return jwt.encode(token,JWT_SECRET,'HS256')
def decodeJwt(token):
    try:
        result=jwt.decode(token,JWT_SECRET,'HS256')
        return result,True
    except jwt.exceptions.PyJWTError:
        return None,False
    jsonresponse = curl.simplePerform("/register",&statusCode);
    if (statusCode==200){
        d.Parse(jsonresponse.c_str());
        jwtToken=string("x-auth: ")+d["x-auth"].GetString();
        return true;
    }
    //client side
```

Client: Biblioteca LibCurl e wrappers

A biblioteca LibCurl é amplamente utilizada na linguagem C para trabalhar com requisições web, entretanto o seu uso é bastante verboso. Tendo em vista um uso mais simples, criamos uma classe wraper para ela, chamada de simpleCurl e conseguimos generalizar as checagens e parâmetros comuns a múltiplos métodos em um só, facilitando o reuso de código.

```
string simpleCurl::simplePerform(const char * subpath,long
*statusCode=nullptr,const char **headers=nullptr,FILE *fp=nullptr){
    string result;
    CURLcode res;
    struct curl_slist *headerList=NULL;
    //build header list
    if (headers!=nullptr){
        for(unsigned i=0;headers[i]!=NULL;i++){
            headerList = curl_slist_append(headerList, headers[i]);
        }
    }
    //set common options
    curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headerList);
    curl_easy_setopt(curl, CURLOPT_SSL_VERIFYPEER, 0L);
    curl_easy_setopt(curl, CURLOPT_SSL_VERIFYHOST, 0L);
    curl_easy_setopt(curl, CURLOPT_URL, (SERVER_URL + subpath).c_str() );
    if (fp!=nullptr){
        curl_easy_setopt(curl, CURLOPT_WRITEDATA,fp);
    }
```

```
    else{
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION,
httpManager::writeCallback);
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, &result);
    }
    //perform and check the request
    res = curl_easy_perform(curl);
    if (statusCode!=nullptr){
        curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, statusCode);
    }
    if(res != CURLE_OK){
        throw runtime_error( string("curl perform error: ") +
curl_easy_strerror(res));
    }
    curl_slist_free_all(headerList);
    return result;
}
```

O método acima configura a maioria dos parâmetros necessários para as requisições https mais simples retornando inclusive a resposta em uma std::string podendo ser usado para post,get e download de arquivos. Simples como:

```
simpleCurl curl;
jsonresponse = curl.simplePerform("/register",&statusCode);
```

frontEnd: classe CPyObject

A api do Python pode ser um pouco complicada de usar. O exemplo mais simples para chamar uma única função possui cerca de 70 linhas. Buscando facilitar o uso dos objetos do tipo `PyObject` incorporamos as rotinas mais comuns dentro de uma classe chamada `CPyObject`. Nos baseamos nesta postagem: https://www.codeproject.com/Articles/820116/Embedding-Python-program-in-a-C-Cplusplus-code

E na documentação do python3 https://docs.python.org/3/extending/embedding.html e melhoramos a classe original, adicionado checagens de ponteiros e exceções tornando o código mais legível.

```
class CPyObject
{
private:
    PyObject *p;
public:
    CPyObject(PyObject* _p)
    {
        if (!_p){
            throw std::runtime_error("PyObject returned null. aborting");
        }
        p=_p;
    }
   ....
```

Dessa forma foi possível referenciar objetos do tipo `CPyObject` de forma segura apenas em uma linha como em:

```
manager::manager()
{
    pModule =  PyImport_Import(PyUnicode_FromString("front"));
    master = PyObject_GetAttrString(pModule, "master");
}
```

Assim podemos referenciar o módulo e a variável de dentro de um arquivo .py de forma simples:

- Cliente - rapidJson - Biblioteca utilizada para manipular as respostas do backend no cliente. Permite montar uma árvore de objetos a partir de uma string json.
- Forntend c++ Biblioteca requests - Uma biblioteca bem popular para realizar requisições HTTP. Foi utilizada no frontEnd para que o administrador tenha um canal de comunicação web com o servidor.
- Flask - microframework para servidor web. Foi utilizado para a ponte de comunicação entre admin e clientes na forma de uma API rest.

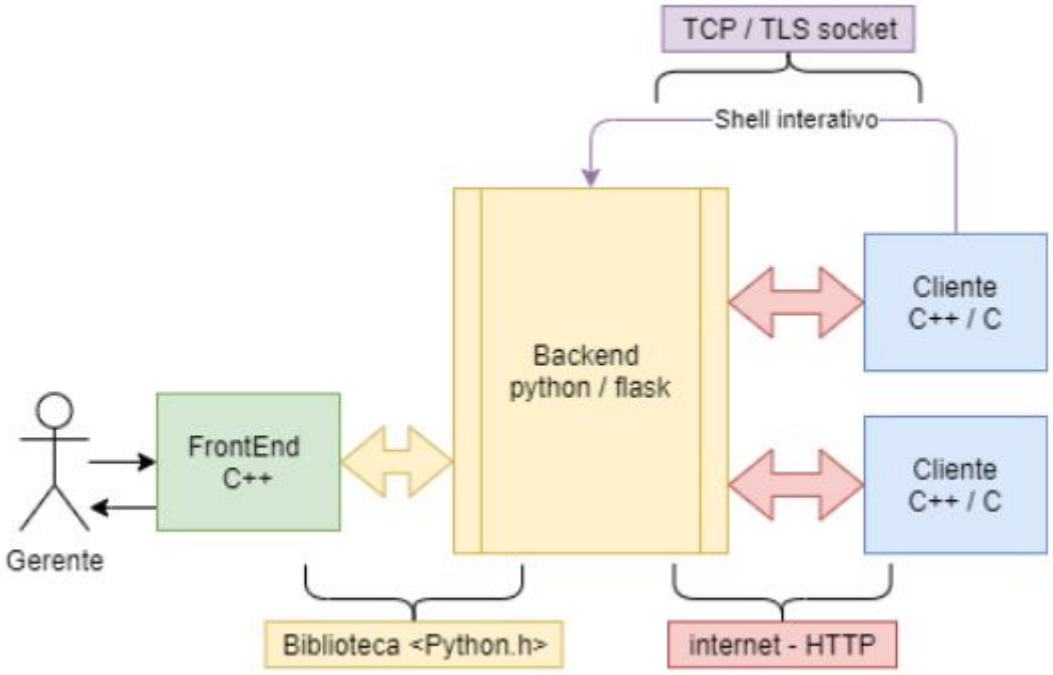
Desafíos enfrentados

A proposta inicial do projeto era que o servidor e frontend estivessem no mesmo programa. Durante o primeiro projeto, tanto o backend quanto o cliente se comportaram bem no protocolo de comunicação. No backend haviam duas threads. Uma para a api em flask e a outra para a iteração do usuário com a classe manager, que replicava os comandos para os clientes.

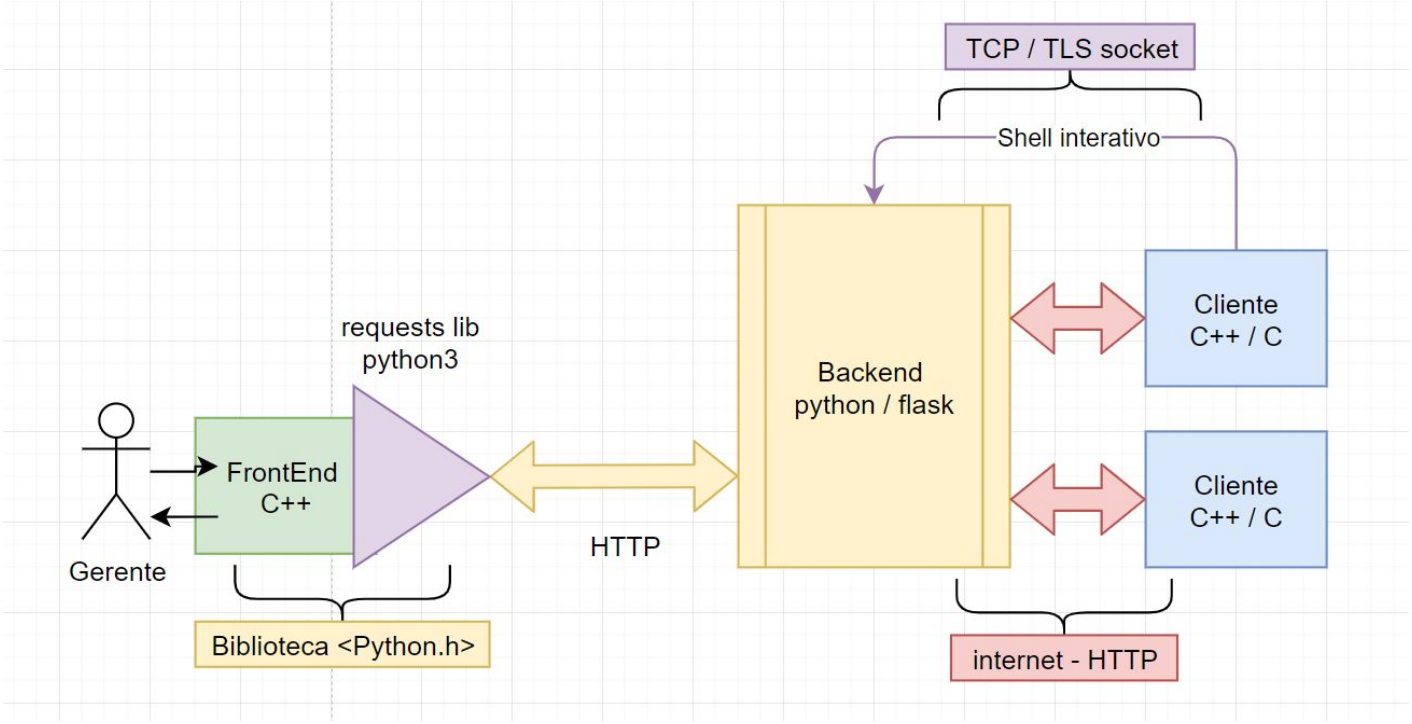
Todavia ao implementar a interface com python, nenhuma requisição estava mais respondendo. Apesar do módulo threading estar sendo utilizado, o nosso processo estava sendo ocupado apenas por IO do cout/cin , não havia fluxo de execução no servidor, o banner de inicialização do backend era impresso (em partes) apenas quando o código em C chamava um método em python. O servidor estava em *starvation ,* que parece fazer sentido, O python não implementa paralelismo, mas concorrência. Apenas uma thread pode estar tomando conta da execução por vez, isso se deve a trava global do interpretador, o GIL (https://realpython.com/python-gil/). Durante uma primeira solução a ideia foi passar as threads para c++, uma para o servidor e outra para o administrador, mas o resultado foi claro enquanto a thread do servidor rodava, toda chamada para o python pela segunda thread retornava em segfault (talvez tenhamos bagunçado o GIL?).

Uma possível solução seria a utilização da biblioteca de multiprocessing: Ao invés de criar mais de uma thread, ele cria outro processo de tal forma que o paralelismo é possível. Todavia como os espaços de endereços são diferentes, as variáveis não são automaticamente replicadas e não existe o suporte para replicação de objetos complexos, o que impossibilitou o seu uso.

A solução foi abandonar a ideia de concorrência no mesmo processo, e separar o backend do administrador. Criando uma rota no backend apenas para acesso do admin e adaptando o frontend em python/C++ para enviar requisições web para ele.



Projeto inicial



Projeto final

Estado de cada módulo

Devido a sua simplicidade, a interface do FrontEnd c++ com o python está funcional, bem como a sua comunicação com o backend em flask, o que está no escopo mínimo do projeto. Todavia embora as funções do cliente parcialmente implementadas, a lógica de comunicação com o servidor não está funcionando corretamente. Eles são capazes de se registrar na rede, mas ao receber alguns comandos do servidor, ficam em um estado travado.O método de shell reverso interativo é o que melhor opera por enquanto, a conexão chega a uma máquina remota mas não mostra resultado.