Instituto Superior Técnico

Programação de Sistemas

Definição de Arquitectura de Sistema e Protocolo de Comunicação para Laboratórios Remotos

Grupo :
Manuel Santos nº 76128
Pedro Rossa nº 87344

Professor:
João Nuno De Oliveira e Silva

2º Semestre de 2021

Introdução

Os laboratórios remotos são ferramentas de aprendizagem importantes que permitem o acesso a experiências reais remotas que, pela natureza dos fenómenos estudados, são difíceis de implementar localmente. As principais características destas experiências que dificultam a sua implementação localmente são o perigo de alguns fenómenos de interesse, e.g. a radioactividade, ou a precisão de medição necessária e complexidade da montagem experimental, o que está correlacionado com o custo da mesma. Outros tipos de experiências, ainda, são de natureza intrinsecamente remota por necessitarem da existência de aparatos experimentais geograficamente distribuídos.

Neste trabalho, desenvolvido no âmbito da cadeira de Programação de Sistemas sob a orientação do Prof. João Nuno de Oliveira e Silva, foi definida a arquitectura de sistema de um laboratório remoto que disponibiliza um conjunto de experiências através da *world wide web* aos utilizadores.
O sistema foi desenhado com base na experiência adquirida no actual laboratório remoto do IST, o e-lab. Os *raw requirements* considerados foram:

- Os utilizadores deverão poder ligar-se a uma aparato experimental, configura-lo e receber os dados gerados pelo mesmo durante a execução experimental pedida.
- Todos os utilizadores ligados deverão receber os dados que a experiência está a gerar.
- Os utilizadores deverão ser capazes de agendar execuções experimentais no aparato.
- Os utilizadores deverão ser capazes de reservar tempo experimental no aparato.
- Os utilizadores deverão ser capazes de aceder aos resultados de execuções experimentais realizadas no passado.
- Os utilizadores deverão ser capazes de se ligar aos aparatos experimentais utilizando dispositivos electrónicos comuns, como PC's, smartphones ou tablets.

De forma a melhor compreender as implicações destes *requirements* foi implementado um protótipo de parte do sistema em Python e utilizado o aparato experimental de testes da experiência Pêndulo Mundial do e-lab, localizado no IST.

1 Arquitectura de Sistema

Apresenta-se na figura 1 a arquitectura definida para o sistema.

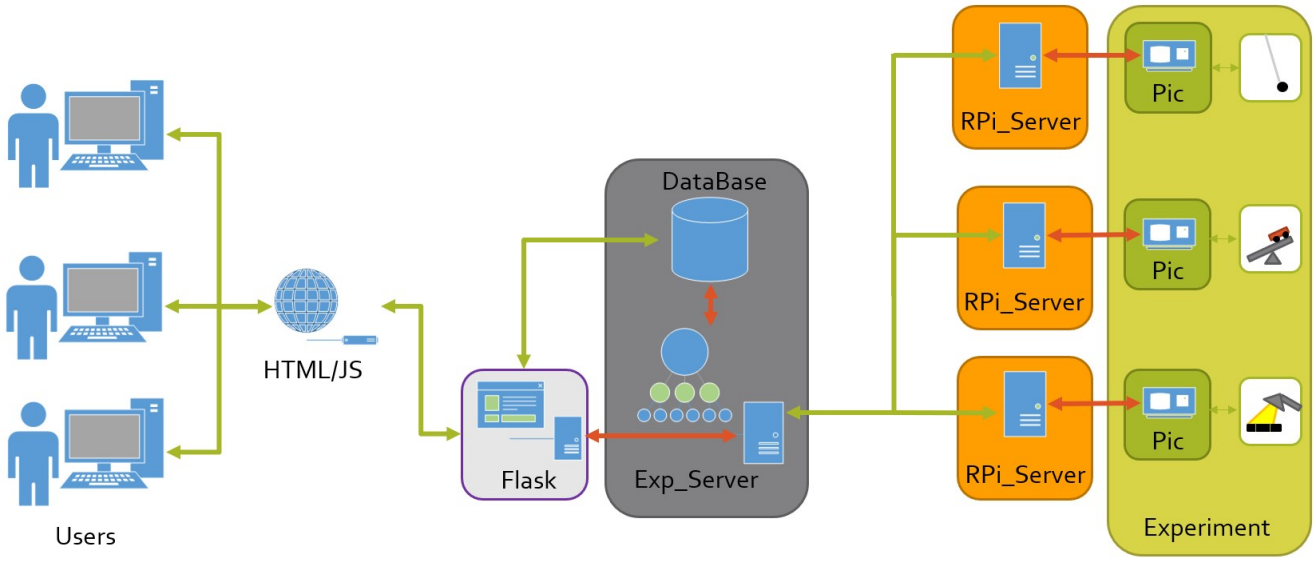
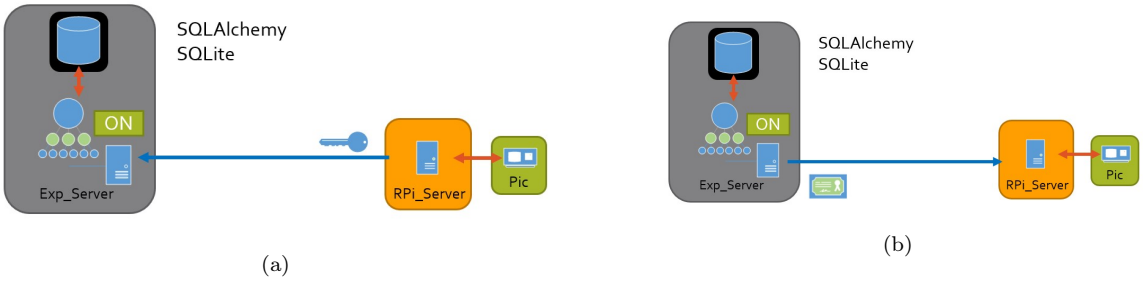


Figura 1: Arquitectura do sistema definida. É possível observar as diferentes camadas de software que separam os utilizadores, do lado esquerdo da imagem, dos aparatos experimentais, do lado direito da imagem.

Na arquitectura definida, os utilizadores comunicam com um servidor central, nomeado *Exp_Server*, através de páginas web.
Este servidor *Exp_Server* contém a base de dados com os resultados das experiências já realizadas e as execuções experimentais e agendamentos de tempo experimental pedidos pelos utilizadores através da interface web. O servidor *Exp_Server* é também responsável por intermediar a comunicação em tempo real entre os utilizadores e as experiências, trocando pedidos de configuração, pedidos e respostas de estado e dados gerados entre os utilizadores activamente ligados aos aparatos e os aparatos.
Junto dos aparatos experimentais um computador, nomeado *RPI_Server*, faz de interface entre o controladores dos aparatos experimentais e o servidor *Exp_Server*. Cada *RPI_Server* está ligado e representa um, e apenas um, aparato experimental. É importante notar que este componente foi nomeado *RPI_Server* por ser, em princípio, implementado através do uso de um Raspberry Pi, um computador que tem módulos de comunicação necessários para comunicar nativamente com os microcontroladores usualmente utilizados para controlar os aparatos. No entanto, este componente pode ser implementado de outras formas sendo apenas estritamente necessário que comunique segundo o protocolo - e.g. uma experiência virtual com dados gerados através de um modelo numérico pode implementar este componente com um computador normal sem interfaces especializadas. A comunicação entre todos os componetes é feita pela internet com a excepção da comunicação entre o *RPi_Server* e o aparato experimental, onde a comunicações é feita através do protocolo que o controlador do aparato implementar.

2 Protocolo de comunicação

Para iniciar a ligação entre o *RPi_Server* e o *Exp_Server*, quando é ligado o *RPi_Server* tenta-se conectar ao servidor *Exp_Server*. Se não tiver sucesso, este deverá esperar 10 segundos e voltar a tentar até conseguir realizar a ligação.

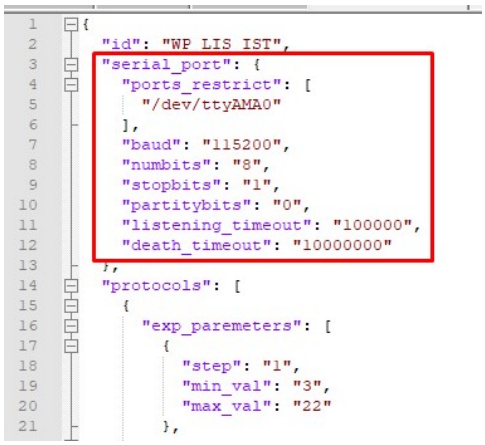


(a) (b)

Figura 2: O processo de autenticação do *RPi_Server*: (a) envio do Segredo e do seu ID; (b) envio da parte do *Exp_Server* da *config_file* da experiência a que o *RPi_Server* se encontra ligado.

Quando a ligação é estabelecida com sucesso o servidor *RPi_Server* envia para o *Exp_Server* o seu ID e um Segredo. Estas duas variáveis são de seguida verificadas do lado do servidor *Exp_Server* como forma simples de autenticar que a ligação é proveniente de uma máquina autorizada. Em caso de verificação bem sucedida, o servidor *Exp_Server* responde ao *RPi_Server* com mensagem codificada em formato JSON que descreve a experiência a ser servida pelo *RPi_Server*. Esta mensagem de configuração permite que o mesmo programa *RPi_Server* seja adaptável a diferentes experiências com, por exemplo, diferentes números de actuadores e sensores sem alterações no código fonte.

Com base nas informações contidas nessa mensagem, nomeada *config_file*, o *RPi_Server* procura a experiência no endereço indicado e tenta estabelecer comunicação com o controlador do aparato experimental. De forma a desacoplar o *RPi_Server* dos diferente protocolos de comunicação que cada controlador experimental poderá implementar, o *RPi_Server* deverá comunicar com os aparatos chamando um conjunto de métodos standard cuja implementação deverá ser adaptada de forma a ser adequada ao controlador alvo. A figura 3 mostra um *config_file* exemplo. Este contém a informação necessária para o *RPi_Server* se ligar a uma experiência ligada via UART, chamando os métodos de comunicação definidos no ficheiro fonte *interface.py*, figura 4.



```
{
  "id": "WP_LIS_IST",
  "serial_port": {
    "ports_restrict": [
      "/dev/ttyAMA0"
    ],
    "baud": "115200",
    "numbits": "8",
    "stopbits": "1",
    "partitybits": "0",
    "listening_timeout": "100000",
    "death_timeout": "10000000"
  },
  "protocols": [
    {
      "exp_paremeters": [
        {
          "step": "1",
          "min_val": "3",
          "max_val": "22"
        },
```

Figura 3: Exemplo de um *config_file* dentro do rectângulo vermelho encontram-se as especificações da *serial communication*.

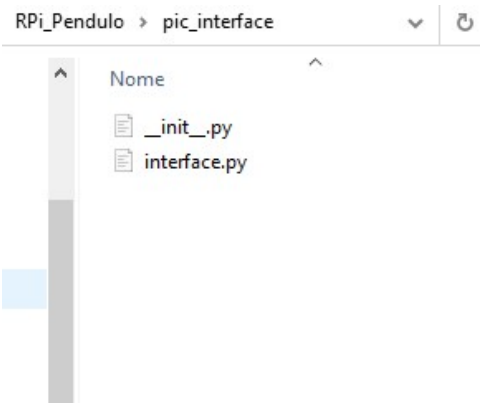


Figura 4: Localização do modulo *interface.py* responsável pela comunicação entre a *serial port* e o *RPi_Server*.

Feita esta configuração inicial, de ligação do *RPi_Server* ao *Exp_Server* e do *RPi_Server* ao controlador da experiência, o sistema está pronto para a troca de mensagens entre os componentes. As mensagens trocadas entre os componentes foram definidas seguindo um filosofia *tudo é uma string*. Assim, as mensagens trocadas são strings formatadas em JSON em que os campos do JSON codificam a mensagem a ser transmitida. O tipo de cada mensagem é identificado pelo campo *msg_id* do JSON da mensagem, sendo que para mensagem enviada a resposta, a existir, deverá ter *reply_id* com o mesmo valor. De forma a ser adaptável a vários tipos de aparatos experimentais, o sistema não guarda informação de estado do aparato

pelo que é possível enviar qualquer das mensagens definidas em qualquer altura. A correcta sequenciação de mensagens para a operação bem sucedida do aparato é responsabilidade de outras camadas de software controladas pelo user.

2.1 Mensagens enviadas *Exp_Server* → *RPi_Server*

2.1.1 Mensagem 1

Mensagem enviada para transmitir os parâmetros de configuração do aparato experimental para o *RPi_Server*. Os parâmetros deverão ser enviados no campo *config_file* formatados como JSON. O JSON de configuração deverá ser adaptado a cada tipo de aparato e ao código fonte de interface utilizado.

```
{"msg_id": "1","config_file": config_json}
```

Como resposta, o *Exp_Server* deverá receber uma das seguintes mensagens, indicando o estado de inicialização do aparato. Em caso de inicialização bem sucedida, a resposta deverá ser da seguinte forma:

```
{"reply_id": "1","status":"Experiment initialized OK"}
```

Em caso de erro, a resposta deverá ser da seguinte forma:

```
{"reply_id": "1", "status":"Experiment initialized NOT OK", "error": error_description}
```

Neste caso, o campo *error* poderá ser usado para descrever em detalhe o erro que ocorreu na inicialização, e.g. não foi possível encontrar a experiência na porta série configurada.

2.1.2 Mensagem 2

Mensagem enviada para transmitir os parâmetros de configuração de uma execução experimental definida por um utilizador. Os parâmetros da execução experimental deverão ser enviados no campo *config_params* codificados no formato JSON. O JSON que codifica os parâmetros de configuração deverá ser adaptado a cada tipo de aparato e ao código fonte de interface utilizado.

```
{"msg_id": "2","config_params": run_config}
```

Como resposta, o *Exp_Server* deverá receber uma das seguintes mensagens, indicando o estado de configuração e execução do aparato. Em caso de configuração e execução bem sucedida, a resposta deverá ser formatada da seguinte forma:

```
{"reply_id": "2","status":"Experiment Running"}
```

Em caso de erro, a resposta deverá ser da seguinte forma:

```
{"reply_id": "2", "status": status_description, "error": error_description}
```

Os campos *status* e *error* deverão ser usados para reportar em que estado ficou o aparato experimental depois do erro e especificar detalhadamente qual o erro que foi detectado, respectivamente.

2.1.3 Mensagem 3

Mensagem enviada para comandar a paragem da execução experimental em curso. O comportamento quando enviado para um aparato experimental já parado não é especificado.

```
{"msg_id":"3"}
```

Como resposta, o *Exp_Server* deverá receber uma das seguintes mensagens, indicando o estado do aparato. Em caso de paragem bem sucedida, a resposta deverá ser formatada da seguinte forma:

```
{"reply_id": "3","status":"Experiment Stopped"}
```

Em caso de erro, a resposta deverá ser da seguinte forma:

```
{"reply_id": "3", "status":"Experiment didn't stop", "error": error_description}
```

O campo *error* deverá ser usado para especificar detalhadamente qual o erro que foi detectado.

2.1.4 Mensagem 4

Mensagem enviada para comandar o reset do aparato.

```
{"msg_id":"4"}
```

Como resposta, o *Exp_Server* deverá receber uma das seguintes mensagens, indicando se foi possível fazer reset ao aparato com sucesso. Em caso de reset bem sucedido, a resposta deverá ser formatada da seguinte forma:

```
{"reply_id": "4","status":"Experiment Reseted"}
```

Em caso de erro, a resposta deverá ser da seguinte forma:

```
{"reply_id": "4", "status":"Experiment didn't Reset", "error": error_description}
```

O campo *error* deverá ser usado para especificar detalhadamente qual o erro que foi detectado.

2.1.5 Mensagem 5

Mensagem enviada para pedir o estado do aparato.

```
{"msg_id":"5"}
```

Como resposta, o *Exp_Server* deverá receber uma das seguintes mensagens, indicando se foi possível determinar o estado do aparato com sucesso. Em caso de determinação de estado bem sucedida, a resposta deverá ser formatada da seguinte forma:

```
{"reply_id": "5","status": exp_status}
```

Em caso de erro, a resposta deverá ser da seguinte forma:

```
{"reply_id": "5", "status":"Couldn't get status", "error": error_description}
```

O campo *error* deverá ser usado para especificar detalhadamente qual o erro que foi detectado.

2.2 Mensagens enviadas pelo *RPi_Server* → *Exp_Server*

2.2.1 Mensagem 6

Mensagem enviada para pedir o registo do *RPi_Server* no *Exp_Server*. Os campos *id_RP* e *segredo* deverão ser usados para codificar o id e o segredo associados ao *RPi_Server* que está a pedir registo.

```
{"msg_id":"6","id_RP": id ,"segredo": secret}
```

Em caso de registo bem sucedido a resposta deverá ser da forma:

```
{"reply_id":"6", "status":"Connected"}
```

Em caso de erro, a resposta deverá ser da forma:

```
{"reply_id":"6", "status":"Connection failed", "error": error_description}
```

O campo error deverá ser usado para especificar detalhadamente qual o erro detectado, e.g. o id enviado não existe na base de dados de experiências autorizadas.

2.2.2 Mensagem 7

Mensagem enviada para transmitir os resultados completos da execução experimental pedida. O campo *results* deverá ser usado para codificar, em formato JSON, os resultados experimentais completos. O formato do JSON depende do tipo de dados gerados pelo aparato.

```
{"msg_id":"7", "results": run_results}
```

2.2.3 Mensagem 8

Mensagem enviada para sinalizar um erro numa execução experimental que já tinha começado. Os campos *status* e *error* deverão ser usados para codificar o estado do aparato depois do erro e qual o erro que foi detectado, respectivamente.

```
{"msg_id":"8", "status": status, "error": error_id}
```

2.2.4 Mensagem 9

Mensagem enviada para transmitir, de forma assíncrona, o estado do aparato para o *Exp_Server*. Os campos *timestamp*, *status* e *current_config* deverão ser usados para codificar a hora a que a mensagem foi gerada, o estado actual do aparato e qual a configuração do aparato, respectivamente.

```
{"msg_id":"9", "timestamp": timestamp, "status": status, "current_config":current_config}
```

2.2.5 Mensagem 10

Mensagem enviada para transmitir, de forma assíncrona, um ponto experimental que não possa ser eficientemente codificado como texto, e.g. uma imagem, e que por essa razão é guardado num local acessível pelo *Exp_Server* em formato binário. Os campos *timestamp* e *id_dados_bin* deverão ser usado para codificar a hora a que a mensagem foi gerada e o endereço onde o cliente poderá obter os dados enviados pelo *RPi_Server*, respectivamente.

```
{"msg_id":"10", " timestamp": timestamp, "bin_data_address": data_address}
```

2.2.6 Mensagem 11

Mensagem enviada para transmitir, de forma assíncrona, um ponto experimental codificado como texto. Os campos *timestamp* e *data* deverão ser usado para codificar a hora a que a mensagem foi gerada e os dados experimentais a transmitir, formatados em JSON, respectivamente. A formatação dos dados experimentais é dependente do tipo de dados gerados pelo aparato e deverá ser apropiada ao mesmo.

```
{"msg_id":"11","timestamp":timestamp, "data": data_point}
```

3 Documentação das Funções

No protótipo criado, de forma a implementar as funcionalidades necessárias a um nível mais alto que o das mensagens individuais descritas anteriormente, foram definidas funções que realizam as tarefas necessárias para a operação do sistema chamando trocando quando necessário as mensagens apropriadas. Estas encontram-se com um *background* com corres diferentes para se poder fazer a seguinte distinção:

Interacção instantânea *Exp_Server* → *RPi_Server*
Controláveis pelo user *Exp_Server* → *RPi_Server* (Flask)
RPi_Server → *Exp_Server*
RPi_Server → Controlador da Experiência (Pic,...) Estas funções podem ser alteradas mas deve-se manter o retorno e a sintaxe das mesmas para assim manter o funcionamento do *RPi_Server* à 100%.

3.1 *Exp_Server*

`check_Experiment(id_Exp, segredo, conn)`

Parameters:

- id_Exp → IP Address da experiência.
- segredo → Password de ligação ao servidor.
- conn → Conexão do socket.

Returns:

- 0 → Conexão estabelecida.
- -1 → Segredo incorrecto.
- -2 → Experiência não existe na data base.

Return type: int

Verifica-se a entidade que se está a tentar ligar ao *Exp_Server* se encontra na base de dados de experiências autenticadas e se o segredo está correcto. Esta função sinaliza sobre esta informação e depois é enviado o *config file* em formato (JSON), código apresentado na secção 4, lista 1.

`ConfigureRP(conn, id_Exp)`

Parameters:

- conn → Conexão do socket.
- id_Exp → IP Address da experiência.

Esta função envia ao ID que se conectou e foi verificado como uma experiência do *e-lab* o seu *config file*: {"msg_id": "1","config_file":"*(JSON...)*"} (mais detalhes ver código apresentado na secção 4 na lista 3).

`ConfigureStartExperiment(user_json)`

Parameters:

- user_json → Configuração da actividade experimental que o utilizador definiu.

Envia a configuração que utilizador pretende executar para o *RPi_Server*, verifica se o número de parâmetros estão correctos de acordo com a *config file* e se os valores destes estão dentro dos limites (secção 4 na lista 2).

JSON enviado para o RPi_Server: {"msg_id": "2","config_params":"user_config"} ou {"msg": "2","error":-1","config_params":"user_config_limitado"} quando os parâmetros passaram o limite destes.

`StopCurrentExperiment(conn)`

Parameters:

- conn → Conexão do socket.

Para a experiência que está a decorrer(secção 4 na lista 4).
JSON enviado para o RPi_Server: {"msg_id":"3"}

`Reset(conn)`

Parameters:

- conn → Conexão do socket.

Para a experiência e volta a lançar a mesma configuração que o utilizador inicialmente tinha enviado(secção 4 na lista 5).
JSON enviado para o RPi_Server: {"msg_id":"4"}

`GetCurrentExperimentStatus(conn)`

Parameters:

- conn → Conexão do socket.

Pede ao *RPi_Server* o estado da experiência. Esta função por si não retorna nenhum valor, mas quando lançada esperar-se receber uma *reply_msg* com o estado da experiência no seu JSON = {"reply_id": "5","status":"Success. Depois aparece aqui o status"}(secção 4 na lista 6).
JSON enviado para o *RPi_Server*: {"msg_id":"5"}

3.2 *RPi_Server*

`IStarted()`

Envia o IP e password do *RPi_Server* para o *Exp_Serve* para ser verificada. Esta função não retorna qualquer informação mas está à espera de um *reply_msg* :

- {"reply_id": "6", "status":"0","info": "Segredo correcto. Bem vindo ao e-lab!"}
- {"reply_id": "6", "status":-1","error": "Segredo incorreto.", "nota":"Verifique se os ficheiros de autenticação estao atualizados."}
- {"reply_id": "6", "status":-2","error": "Experiencia não existe na data base.", "nota":"Contacte o E-lab a experiencia ainda não deve estar registada."}

e do envio do ficheiro de configuração da experiência (secção 4 na lista 7).
JSON enviado para o *Exp_Server*: {"msg_id": "6","id_RP":"JSON_MY_IP","segredo":"JSON_Segredo"}

`Send_Config_to_Pid(myjson)`

Parameters:

- myjson → JSON com os nomes dos parâmetros e os valores dados pelo utilizador.

Envia o JSON com os parâmetros do utilizador para o controlador da experiência dando a reposta ao *Exp_Server*:

- {"reply_id": "2","status":"Experiment Running"}
- {"reply_id": "2", "error":-1", "status":"Experiment could not start"}
- {"reply_id": "2", "error":-2", "status":"Experiment could not be configured"}

depois gera uma *thread* onde é lançada a função *send_exp_data*(secção 4 na lista 8).

`send_exp_data()`

Envia os dados obtidos durante o decorrer da experiência para *Exp_Server* JSON = {"msg_id":"11", "timestamp":"time","status":"running","Data":"Data_point"} e no final da experiência volta a enviar todos os pontos obtidos ao longo da experiencia JSON = {"msg_id":"7", "results":"Full_Data"}(secção 4 na lista 9).

`SendFile(bin_data)`

Parameters:

- bin_data → Array de dados binário (e.g. imagem).

Envia para o *Exp_Server* dados experimentais em formato binário. O *Exp_Server* responde com o endereço onde ficaram guardados os dados (secção 4 na lista 10), sendo que por fim o *RPi_Server* transmite uma mensagem de dados com a localização dos dados binários para o *Exp_Server*.

`SendStatus(type_data,timestamp, experiment_status,current_config)`

Parameters:

- type_data → Tipo de mensagem a ser comunicada para o *Exp_Server*.
- timestamp → Tempo do envio da mensagem.
- experiment_status → Estado da experiência (type_data = 9) ou local onde foram guardados os dados enviados pela função *SendFile*.
- current_config → Configuração da experiência.

Envia o estado da experiência(*type_data* = 9), e o local onde se encontra guardado na base de dados(a implementar no futuro) o ficheiro binário que a experiência gerou(*type_data* = 10)(secção 4 na lista 11).

3.2.1 *interface.py* (*RPi_Server*)

`do_init(config_json)`

Parameters:

- config_json → JSON com a configuração da experiência, onde se encontra explicita a forma de comunicação entre o controlador e o *RPi_Server*.

Verifica se o controlador da experiência está ligado à porta definida na configuração e enviada ao *RPi_Server*, assim iniciando a comunicação, dando como retorno *True*, em caso de erro retorna *False*.

`do_config(config_json)`

Parameters:

- config_json → Ficheiro com os parâmetros que o utilizador escolheu para a realização da experiência.

Returns:

- (pic_message, True)
- (-1, False)

Configura o controlador da experiência com a configuração do utilizador e retorna a configuração enviada pelo controlador da experiência e *True*, o envio da configuração retornada pelo controlador é só uma medida para a verificação de erros. Em caso de erro esta função envia (-1, False).

`receive_data_from_exp()`

Detecta o envio de dados por parte do controlador retornando a *string* "DATA_START", à medida que os dados vão chegando ao port este formata o JSON com o *time_samp* e os pontos experimentais obtidos, no caso do Pêndulo {"Sample_number":"...","Val1":"...","Val2":"...","Val3":"...","Val4":"..."}. No final volta a envia uma *string* que representa o fim da experiência "DATA_END".

`do_start()`

Retorna *True* se o controlador iniciar a experiência programada, *False* em caso de falha.

`do_stop()`

Retorna *True* se o controlador parar a experiência que esta em execução, *False* em caso de falha.

`do_reset()`

Retorna *True* se o controlador fizer restart, *False* em caso de falha.

`get_status()`

Ainda não desenvolvida mas esta função retornará o estado da experiência que o controlador da mesma tiver a enviar.

```
(...)
segredos = {"10.7.0.35":{"segredo":"sou eu","nome":"WP_LIS_IST"},%
            "192.168.1.7":{"segredo":"estou bem","nome":"Cavidade"}}
(...)

def check_Experiment(id_Exp, segredo,conn):
    #print(segredos.get(id_Exp)!=None)

    # Testa se o ID existe no safe das experiencias que existem.
    if segredos.get(id_Exp)!=None:
        # Testa se o segredo esta correcto
        if (segredo == segredos[id_Exp]['segredo']):
            print('A experiencia '+ segredos[id_Exp]['nome'] + ' ('+id_Exp +') foi conectada')
            global EXP_CONN_LIST
            EXP_CONN_LIST[segredos[id_Exp]['nome']] = conn
            return 0
        return -1
    else:
        return -2

def check_msg(myjson,conn):
    msg_id = myjson['msg_id']
    (...)
    elif(msg_id == '6'):
            print("Recebi mensagem com pedido de configuracao\n")
            msg_erro = check_Experiment(myjson['id_RP'], myjson['segredo'],conn)
            if (msg_erro == 0):
                print ('reply_id = 6, Comunicação: Conecção estabelecida.')
                send_mensage ='{"reply_id": "6", "status":"0", \
                               "info": "Segredo correcto. Bem vindo ao e-lab!"}'
                send(send_mensage,conn)
                ConfigureRP(conn,myjson['id_RP'])
            elif (msg_erro == -1):
                print ('msg_id = 6, ERROR: Segredo incorreto.')
                send_mensage ='{"reply_id": "6", "status":"-1",\
                               "error": "Segredo incorreto.",\
                               "nota":"Verifique se os ficheiros de autenticação estao atualizados."}'
                send(send_mensage,conn)
            elif (msg_erro == -2):
                print ('msg_id = 6, ERROR: Experiencia não existe na data base.')
                send_mensage ='{"reply_id": "6", "status":"-2",\
                               "error": "Experiencia não existe na data base.",\
                               "nota":"Contacte o E-lab a experiencia ainda não deve estar registada."}'
                send(send_mensage,conn)
            return True
    (...)
```

Lista 1: Código da função check_Experiment.

```
def ConfigureStartExperiment(user_json):
    #VALIDAR CONFIG
    verificar = []
    conn = EXP_CONN_LIST[user_json['experiment_name']]
    exp_config_json = user_json['config_experiment']
    if 'protocol' in user_json:
        protocol = int(user_json['protocol'])
    else:
        protocol = 0
    tester = json.loads(EXP_PROCOL[user_json['experiment_name']])
    if len(tester['protocols'][protocol]['exp_paremeters']) == len(exp_config_json.keys()):
        print('\nO numero de parametros enviado pelo user esta de acordo com a config! \n')
        print('\nA iniciar verificação de limites! \n')
        for i in range(0,len(exp_config_json.keys())):
            min = int(tester['protocols'][protocol]['exp_paremeters'][i]['min_val'])
            max = int(tester['protocols'][protocol]['exp_paremeters'][i]['max_val'])
            if min <= int(exp_config_json[list(exp_config_json.keys())[i]]) and
               max >= int(exp_config_json[list(exp_config_json.keys())[i]]):
                print ('Esta dentro dos limites a variavel '+ str(list(exp_config_json.keys())[i]))
                print ('\n')
                verificar.append(True)
            elif min > int(exp_config_json[list(exp_config_json.keys())[i]]):
                print('erro')
                exp_config_json[list(exp_config_json.keys())[i]] = str(min)
                verificar.append(False)
            else:
                print('erro')
                exp_config_json[list(exp_config_json.keys())[i]] = str(max)
                exp_config = json.dumps(exp_config_json)
                verificar.append(False)
        if all(verificar) == False:
            exp_config = json.dumps(exp_config_json)
            send_mensage = '{"msg_id": "2","error":"-1","config_params":'+str(exp_config)+'}'
        else:
            exp_config = json.dumps(exp_config_json)
            send_mensage = '{"msg_id": "2","config_params":'+str(exp_config)+'}'

        print(send_mensage)

        #Send to target experiment server
        send(send_mensage,conn)
    else:
        print('Numero de parametos errado por favor verifique a config que mandou!!\n')
```

Lista 2: Código da função ConfigureStartExperiment.

```
def get_Config(Exp):
    #Apanhar erro se ficheiro no existir
    config = 'Configs/'+str(Exp)+'.json'
    with open(config) as json_file:
        data = json.load(json_file)
    return data

def ConfigureRP(conn, id_exp):
    data = get_Config(segredos[id_exp]['nome'])
    data_1 = json.dumps(data)
    send_mensage ='{"msg_id": "1","config_file":'+str(data_1)+'}'
    print(send_mensage)
    send(send_mensage,conn)
    global EXP_PROCOL
    EXP_PROCOL[segredos[id_exp]['nome']] = '{"protocols":'+str(data['protocols']).replace('\'','"')+'}'
```

Lista 3: Código da função ConfigureRP.

```
def StopCurrentExperiment(conn):
    print("A enviar mensagem com pedido para para experiencia\n")
    send_message = '{"msg_id":"3"}'
    send(send_message,conn)
    #CHECK_REPLY
```

Lista 4: Código da função StopCurrentExperiment.

```
def Reset(conn):
    print("A enviar mensagem com pedido para dar reset na experiencia")
    send_message = '{"msg_id":"4"}'
    send(send_message,conn)
    #CHECK_REPLY
```

Lista 5: Código da função Reset.

```
def GetCurrentExperimentStatus(conn):
    print("A enviar mensagem com pedido sobre estado da experiencia")
    send_message = '{"msg_id":"5"}'
    send(send_message,conn)
    #CHECK_REPLY
```

Lista 6: Código da função GetCurrentExperimentStatus.

```
def IStarted():
    print("Arranquei. A mandar pedido de ligacao ao servidor de experiencias\n")
    send_msg = '{"msg_id": "6","id_RP":"'+str(MY_IP)+'","segredo":"'+str(SEGREDO)+'"}'
    send(send_msg)
```

Lista 7: Código da função IStarted.

```
def Send_Config_to_Pid(myjson):
    print("Recebi mensagem de configurestart. A tentar configurar pic")
    actual_config, config_feita_correcta = interface.do_config(myjson)
    if config_feita_correcta :   #se config feita igual a pedida? (opcional?)
        data_thread = threading.Thread(target=send_exp_data,daemon=True)
        print("PIC configurado.\n")
        if interface.do_start():                          #tentar começar experiencia
            data_thread.start()
           send_mensage = '{"reply_id": "2","status":"Experiment Running"}'
        else :
            send_mensage = '{"reply_id": "2", "error":"-1", "status":"Experiment could not start"}'

    else:
        send_mensage = '{"reply_id": "2", "error":"-2", "status":"Experiment could not be configured"}'
    return send_mensage
```

Lista 8: Código da função Send_Config_to_Pid.

```
def send_exp_data():
    global SAVE_DATA

    while interface.receive_data_from_exp() != "DATA_START":
        pass
    send_message = '{"msg_id":"11","timestamp":"'+str(time.time_ns())+
                    '","status":"Experiment Starting","Data":""}'
    send(send_message)
    while True:
        exp_data = interface.receive_data_from_exp()
        if exp_data != "DATA_END":
            SAVE_DATA.append('{"timestamp":"'+str(time.time_ns())+'","Data":'+str(exp_data)+'}')
            send_message = '{"msg_id":"11","timestamp":"'+str(time.time_ns())+
                           '","status":"running","Data":'+str(exp_data)+'}'
            send(send_message)
        else:
            send_message = '{"msg_id":"11","timestamp":"'+str(time.time_ns())+
                           '","status":"Experiment Ended","Data":""}'
            send(send_message)
            send_message = '{"msg_id":"7", "results":'+str(SAVE_DATA).replace('\'', '')+'}'
            send(send_message)
            return #EXPERIMENT ENDED; END THREAD
```

Lista 9: Código da função send_exp_data.

```
def SendFile(bin_data) :
    bin_data_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    bin_data_socket.connect((SERVER, BINARY_DATA_PORT)) #APANHAR ERROS

    bin_data_socket.send(len(bin_data))
    bin_data_socket.send(bin_data)

    reply_size = bin_data_socket.recv(HEADER)
    msg = bin_data_socket.recv(reply_size)

    #Reply message received, comunication is done, close socket
    bin_data_socket.shutdown(socket.SHUT_RDWR)
    bin_data_socket.close()

    #CHECK MESSAGE
    #SEND STATUS TO SERVER
```

Lista 10: Código da função SendFile.

```
def SendStatus(type_data,timestamp, experiment_status,current_config):
    if type_data == 9:
        send_msg = '{"msg_id":"9", " timestamp":"'+timestamp+'",\
                    "experiment_status":"'+ experiment_status +'",\
                    "current_config":"'+current_config+'"}'
    if type_data == 10:
        send_msg = '{"msg_id":"10", " timestamp":"'+timestamp+'", "id_dados_bin":'+ experiment_status+'"}'
    send(send_msg)
```

Lista 11: Código da função SendStatus.