Exemplos

Utilização do sockets em Python

Troca de mensagens

```
{Cliente: É enviada string para o servidor e recebe string do cliente}

def eco(self, msg: str) -> Union[str, None]:
    self.s.send(msg.encode(constante.CODIFICACAO_STR))

    if msg != constante.FIM:
        dados_recebidos: bytes = self.s.recv(constante.TAMANHO_MENSAGEM)
        return dados_recebidos.decode(constante.CODIFICACAO_STR)
    else:
        self.s.close()
dados_recebidos: bytes = socket_client.recv(constante.TAMANHO_MENSAGEM)
dados = dados_recebidos.decode(constante.CODIFICACAO_STR)
```

Uma string é enviada codificada (*encode*) com o respetivo código de codificação. É recebida como bytes e novamente convertida (*decode*) usando o mesmo código.

Troca de mensagens

{Cliente: envio de dois inteiros e receção de resultado}

```
...
self.s.send(msg.encode(constante.CODIFICACAO_STR))
self.s.send(value1.to_bytes(constante.N_BYTES, byteorder="big", signed=True))
self.s.send(value2.to_bytes(constante.N_BYTES, byteorder="big", signed=True))
dados_recebidos: bytes = self.s.recv(constante.N_BYTES)
return int.from_bytes(dados_recebidos, byteorder='big', signed=True)
```

Quando se trata de um inteiro, este é convertido em bytes pela função *to_bytes.* O número de bytes indica a dimensão máxima da mensagem.
Ao ser recebido, tem de ser devidamente descodificado através da função *from_bytes*.

Troca de mensagens

{Servidor: “Serialização” de dados e envio de dados codificados como string }

```
msg = json.dumps(value)
# Get the size of serialized data
size = len(msg)
sc.send(size.to_bytes(constant.BYTE_NR, byteorder="big", signed=True))
# Test
print("Obstacles, message sent to client:",msg)
sc.send(msg.encode(constant.STR_CODE))
```

Quando o conteúdo da mensagem é complexo como, por exemplo, um dicionário, ele tem de ser transformado numa sequência de caracteres ordenados de determinada forma. É o que acontece com o comando *json.dumps - diz-se que ocorre a serialização dos dados*.
Depois enviado como uma *string*. .
Note-se que primeiro é enviada dimensão da mensagem.

Troca de mensagens

{Cliente: receção de dados complexos e “des-serialização”}

```
rec: bytes = self.s.recv(constant.BYTE_NR)
obst_size = int.from_bytes(rec, byteorder='big', signed=True)
rec: bytes = self.s.recv(obst_size)
obst = json.loads(rec)
```

Ao receber uma estrutura de dados complexa, a dimensão é importante.
A primeira mensagem corresponde a dimensão da mensagens recebida.
Depois ela é convertida usando o *json.loads(rec)*

Protocolo

- Quando existe um cliente e um servidor há que definir o protocolo que gere a comunicação entre ambos, sabendo que:
  - Usa-se o TCP/IP, ou seja, existe um canal de comunicação aberto.
  - Quem inicia a comunicação é sempre o cliente.
  - Podem ser enviadas várias mensagens quer do cliente quer do servidor de acordo com o protocolo estabelecido.

Protocolo

- Exemplos
  - O jogo inicia-se.
  - É necessário receber as dimensões do jogo do servidor para construir o painel do Pygame [MAX_X_OP, MAX_Y_OP].
  - É necessário saber onde estão os obstáculos para colocar no painel do Pygame.
  - É preciso dar o nome do jogador e receber a sua posição inicial.

Dimensões do Jogo

O protocolo obedece às seguintes etapas:
- O cliente enviar msg pedindo dimensão x do jogo.
- O servidor retorna essa dimensão.
- O cliente envia msg pedindo dimensão y do jogo.
- O servidor retorna essa mensagem

{cliente: Pedido de dimensões do jogo}

```
def get_max_values(self) -> tuple:
    msg = constant.MAX_X_OP
    self.s.send(msg.encode(constant.STR_CODE))
    rec: bytes = self.s.recv(constant.BYTE_NR)
    max_x = int.from_bytes(rec, byteorder='big', signed=True)
    msg = constant.MAX_Y_OP
    self.s.send(msg.encode(constant.STR_CODE))
    rec: bytes = self.s.recv(constant.BYTE_NR)
    max_y = int.from_bytes(rec, byteorder='big', signed=True)
    # Test
    print("Receive values max_x=",max_x," max_y=",max_y)
    return (max_x, max_y)
```

Obstáculos

O protocolo é o seguinte:

- O cliente pede o número de obstáculos.
- O servidor devolve esse número.
- O cliente pede a localização dos obstáculos.
- O servidor envia primeiro a dimensão da mensagem e depois envia os obstáculos (“serialized”).

```
def get_obstacles(self) -> Union [int, dict]:
    msg = constant.NR_OBST_OP
    self.s.send(msg.encode(constant.STR_CODE))
    rec: bytes = self.s.recv(constant.BYTE_NR)
    nr_obst = int.from_bytes(rec, byteorder = 'big', signed = True)
    print("Nr of obstacles:", nr_obst)
    msg = constant.OBST_OP
    self.s.send(msg.encode(constant.STR_CODE))
    rec: bytes = self.s.recv(constant.BYTE_NR)
    obst_size = int.from_bytes(rec, byteorder='big', signed=True)
    rec: bytes = self.s.recv(obst_size)
    obst = json.loads(rec)
    # Test
    print("Object received:",obst)
    return nr_obst, obst
```

Múltiplos clientes

- Quando um servidor tem mais do que um cliente:
  - Interage sequencialmente com cada cliente.
  - Criar uma *thread* específica para esse cliente. É a partir dessa thread que o servidor responde às solicitações do cliente.

Servidor com várias *thread*

- Um servidor com uma thread por cliente irá aceder aos dados do jogo, atualizando-os sempre que um jogador executa uma jogada, por exemplo.
- Essa atualização é realizada em concorrência. Por essa razão é necessário controlar o acesso aos dados.
- O acesso aos dados partilhados tem de ser controlado por ferrolhos ou fechaduras (*i.e. locks*) que impedem mais do que uma thread aceder a parte do código considerado crítico.
- A parte do código que é crítica (*i.e.* secção crítica) é aquela parte em que se existem mais do que uma thread a usá-la pode gerar inconsistência. Tem de garantir-se que apenas uma thread a executa (*i.e.* exclusão mútua)

Inconsistência

Inconsistência na partilha de dados

```c
#include <stdio.h>
#include <pthread.h>
#include "mythreads.h"

static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

//
// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

O contador é uma variável partilhada!
Porquê “volatile”?

Vai contar até 10 000 000

Dois **thread** executam sobre a mesma variável. Que valor final para o contador?

Inconsistência na partilha de dados

Variável **counter** encontra-se no endereço **0x8049a1c**

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Adiciona 1 (0x1) aos conteúdos do registo

Guarda o o valor do registo no endereço da variável.

OS	Thread 1	Thread 2	(after instruction) PC	%eax	counter
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	**50**	50
	add $0x1, %eax		108	**51**	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	**50**	50
		add $0x1, %eax	108	**51**	50
		mov %eax, 0x8049a1c	113	51	**51**
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

Thread 1: O registo fica com o valor 51 enquanto o contador ainda tem o valor 50.

Thread 2: Completa-se o ciclo de incremento!

Thread 1: Vai recuperar o valor do registo eax (51) e coloca-o na memória (**novamente 51!**)