

Cloud computing Homework 1

Hao Yu

October 15 2021

Abstract

This assignment is an RPC experiment, divided into the following parts, RPC principle, experimental environment, experimental details, source code and conclusion.

1 RPC principle

- (a) Remote procedure call (RPC) is a technology used to build distributed, client-server based applications. It is based on extending the conventional local procedure calling so that the called procedure need not exist in the same address space as the calling procedure.
- (b) Figure 1 is the model of remote procedure call.

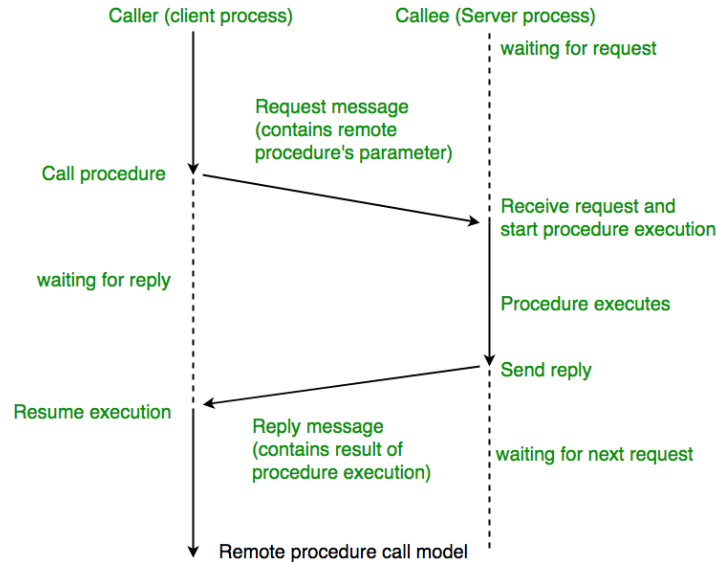


Figure 1: Remote procedure call model

2 Experimental environment

- (a) The programming language used in the experiment program is Python, and the version is 3.7.

- (b) The RPC package used in the experimental program is RPyC. RPyC (pronounced like are-pie-see), or Remote Python Call, is a transparent library for symmetrical remote procedure calls, clustering, and distributed-computing. RPyC makes use of object-proxying, a technique that employs python's dynamic nature, to overcome the physical boundaries between processes and computers, so that remote objects can be manipulated as if they were local.
- (c) The client is deployed on macOS. Figure 2 shows the system information.

```

      'c.      ethan@Ethans-MacBook-Pro.local
      ,xMMM.   -----
      .OMMMMo  OS: macOS 11.5.2 20G95 arm64
      OMMM0,   Host: MacBookPro17,1
      .;lodd0:' loolloddol;.  Kernel: 20.6.0
      cKMMMMMMMMMMNWMMMMMMMMMM0:  Uptime: 1 day, 9 hours, 26 mins
      .KMMMMMMMMMMMMMMMMMMMMMMMMWd.  Packages: 1 (brew)
      XMMMMMMMMMMMMMMMMMMMMMMMMMX.  Shell: zsh 5.8
      ;MMMMMMMMMMMMMMMMMMMMMMMMM:  Resolution: 1440x900, 1920x1080
      :MMMMMMMMMMMMMMMMMMMMMMMMM:  DE: Aqua
      .MMMMMMMMMMMMMMMMMMMMMMMMMX.  WM: Quartz Compositor
      kMMMMMMMMMMMMMMMMMMMMMMMMWd.  WM Theme: Blue (Light)
      .XMMMMMMMMMMMMMMMMMMMMMMMMMk  Terminal: tmux
      .XMMMMMMMMMMMMMMMMMMMMMMMMMk.  CPU: Apple M1
      kMMMMMMMMMMMMMMMMMMMMMMMMMd   GPU: Apple M1
      ;KMMMMMMMMWWXXMMMMMMMMMk.     Memory: 2978MiB / 16384MiB
      .COOC,.      .,COO:.

```



Figure 2: Client system information

- (d) The server is deployed in the docker container of an Azure server in the US. Figure 3 shows the system information.

```

      _met$$$$$gg.      root@1ae09ca31e1a
      ,g$$$$$$$$$$$$P.  -----
      ,g$$P"      ""Y$$.".  OS: Debian GNU/Linux 11 (bullseye) x86_64
      ,$$P'      `$$$.      Host: Virtual Machine 7.0
      ',$$P      ,ggs.      `$$b:  Kernel: 5.8.0-1042-azure
      `d$$'      ,$$P"      .   $$$  Uptime: 2 days, 23 hours, 24 mins
      $$$      d$'      ,   $$$P  Packages: 462 (dpkg)
      $$$      $$.      -   d$$'   Shell: zsh 5.8
      $;$      Y$b._   _,$$P'     CPU: Intel Xeon Platinum 8272CL (1) @ 2.593GHz
      Y$$      `."Y$$$$$P"      Memory: 442MiB / 916MiB
      `$$b      "-._
      `Y$$
      `Y$$
      `$$b.
      `Y$$b.
      `Y$b._

```




Figure 3: Server system information

3 Experimental details

(a) Experiment 1

Request 100 read operations, the data size is 100 bytes, and the average response time is 0.40775113833000004 seconds.

Request 100 write operations, the data size is 100 bytes, and the average response time is 0.4097092793800001 seconds.

Request 100 modification operations, the data size is 100 bytes, and the average response time is 0.418472302340725 seconds.

(b) Experiment 2

Request 100 write operations, the data size is 10 bytes, and the average response time is 0.4070674958099999 seconds.

Request 100 write operations, the data size is 1000 bytes, and the average response time is 0.41493206457999965 seconds.

Request 100 write operations, the data size is 100000 bytes, and the average response time is 0.6903830932800006 seconds.

(c) Experiment 3

Request 100 modification operations, the data size is 100 bytes, and the average response time is 0.418472302340725 seconds.

Request 100 modification operations for complex calculations, the data size is 100 bytes, and the average response time is 0.4574660183399997 seconds.

4 Source code

(a) The GitHub repository link is <https://github.com/0x00000024/rpc-experiment>

(b) Please refer to Figure 4 for the client code.

(c) Please refer to Figure 5 for the server code.

5 Conclusion

After testing, I found that the average response time of the client calling the function in the server depends on the following two factors:

(a) How much data is transmitted in the communication process

(b) The running time of the function called remotely

```

1  import random
1  from timeit import default_timer as timer
2  import string
3  from typing import Any
4  import rpyc
5
6
7  def get_n_random_chars(n: int) -> str:
8      return ''.join(random.choices(string.ascii_uppercase + string.digits, k=n))
9
10
11 class RemoteProcedureCall:
12     def __init__(self, connection: Any) -> None:
13         self.data: str = ''
14         self.conn: Any = connection
15
16     def read_operation(self) -> None:
17         print(f'Read data:\n{self.conn.root.read()}')
18
19     def write_operation(self, n: int) -> None:
20         self.data = get_n_random_chars(n)
21         print(f'Write data:\n{self.conn.root.write(self.data)}')
22
23     def modify_operation(self, start_index: int, n: int) -> None:
24         # Make the size of the sent data consistent
25         self.data = get_n_random_chars(n * 2)
26         print(
27             f'Modify data:\n{self.conn.root.modify(self.data, start_index, n)}'
28         )
29
30
31  if __name__ == "__main__":
32      total_time = 0
33      cycles = 100
34      string_len = 1000
35      print('RPC client started!')
36      conn = rpyc.connect("13.87.134.248", 18861)
37      rpc = RemoteProcedureCall(connection=conn)
38      # print(rpc.read_operation())
39      for i in range(cycles):
40          start_time = timer()
41          # rpc.read_operation()
42          rpc.write_operation(string_len)
43          # rpc.modify_operation(start_index=random.randint(0, string_len),
44          #                       n=int(string_len/2))
45          end_time = timer()
46          elapsed_time = end_time - start_time
47          print(f'{elapsed_time} seconds')
48          total_time += elapsed_time
49
50      print(f'Average time: {total_time / cycles}')

```

Figure 4: The source code of the client

```

1  import rpyc
1  from rpyc.utils.server import ThreadedServer
2
3
4  class MyService(rpyc.Service):
5      def __init__(self):
6          self.data = '00000I am the server, this is a 100-byte string. I am the server,' \
7                      ' this is a 100-byte string. I000000'
8          print(f'Server initialized data:\n{self.data}')
9
10     def exposed_read(self) -> str:
11         print(f'Read data:\n{self.data}')
12         return self.data
13
14     def exposed_write(self, client_data: str) -> str:
15         self.data = client_data
16         print(f'Write data:\n{self.data}')
17         return self.data
18
19     def exposed_modify(self, client_data: str, start_index: int,
20                       n: int) -> str:
21         if n > 100:
22             return ''
23         tmp = self.data[0:start_index] + client_data[0:n] + self.data[
24             start_index + n:len(self.data)]
25         self.data = tmp
26         print(f'Modify data:\n{self.data}')
27         return self.data
28
29
30 ► if __name__ == "__main__":
31     print('RPC server started!')
32     t = ThreadedServer(MyService, port=18861)
33     t.start()

```

Figure 5: The source code of the server