Sample Application with *NETWORK CONNECTION*

1. Introduction

1.1. Objectives

<to be developed>

1.2. Our contributions

We provide a sample framework of designing and implementing an application called NetApp. This application is a reference design to illustrate the theoretical materials in the course Computer Network (CO3093 and CO3094) at HCMUT. Our work includes :

- Provide background concepts of NetApp
- Present the design of the two versions of applications by employing the mentioned concepts.
- Provide a sample implementation framework.

2. Background

2.1. Application in academic theory

An application or a program is implemented by employing a multi-**process** multi-**thread** concept. NetApp is an application including process communication, i.e. a connection from process to process.

In process communication, there are fundamentally two types of process: process server and process client.

1. Process server: need permanent address and must be alway online
2. Process client: need to know the server address (know real and exact address), and is the guy who initialize the connection to connect to server process

There is no such term to call an application as client-server or peer-to-peer. The term is only the two **paradigms** where we place the two kinds of processes: (i) Client-server paradigm: on each host there are only one kind of processes; and (ii) Peer-to-peer paradigm: on each host there are both types of processes

2.2. HTTP and 6 alternatives of client server paradigm

1. Protocol has 2 info (i) formats of data and (ii) the rule of communication (step 1, step 2, etc.).
2. HTTP (original form) is a prominent representative protocol for client-server paradigm.
    a. HTTP protocol rule has 2 ways: (a) persistent connect to setup once and each connection retrieve each of the set of file (in total N+1 RTT for N file) RTT is a round of connect from one side to another side and return back to the original side; (b) non-persistent each of the set file connect one and get only 1 file (in total 2N RTT for N file).
    b. HTTP format: data in ASCII encode (normal text) and follow the format of request and response. Do the experiment in slide 31 to see the simple of HTTP request and response
3. HTTP limitation and can be fixed (technically) throughput is decided by bottle neck point where it is the connection to remote server. Fix by saving a copy of content at the local server, the **cache** technique, and we have to think of a good design to get a good hit rate. *(alternative 1)*
4. HTTP limitation and can be fixed (technically) duplicate content ⇒ **conditional** GET only gets the content component which has been modified. Non-modified components then return 304 and reuse the old content. *(alternative 2)*
5. HTTP limitation NOT fixable (technically) stateless servers return the same result for all connections, all users and all the time. Fixing by adding client info saved in **cookies** and each request sends both cookies content + request to get personalized response from server. Long time to remove the use of cookies but cannot, then limit the use by law. *(alternative 3)*
6. HTTP limitation and can be fixed (technically) server can be offline. Workaround by indirectly sending to the **agency**, then the agency is on behalf of the client to connect to the server to do the task when the agency detects the server online. *(Email is an illustration - alternative 4)*

7. HTTP limitation and can be fixed (technically) HTTP only connect one server, how to get respónse by distributed (and **multiple**) **servers**, how these servers can give response together, two mode: *(DNS is an illustration - alternative 5)*
    a. Iterative query: ask one, get the next guy (server) response delegated to closer answer then contact that delegated guy
    b. Recursive query: as a server, the server is on behalf of client to ask the next guy and wait to obtain the answer, if the next guy is not closed enough, he will recursively ask the next of the next guy,

3. System design of NetApp

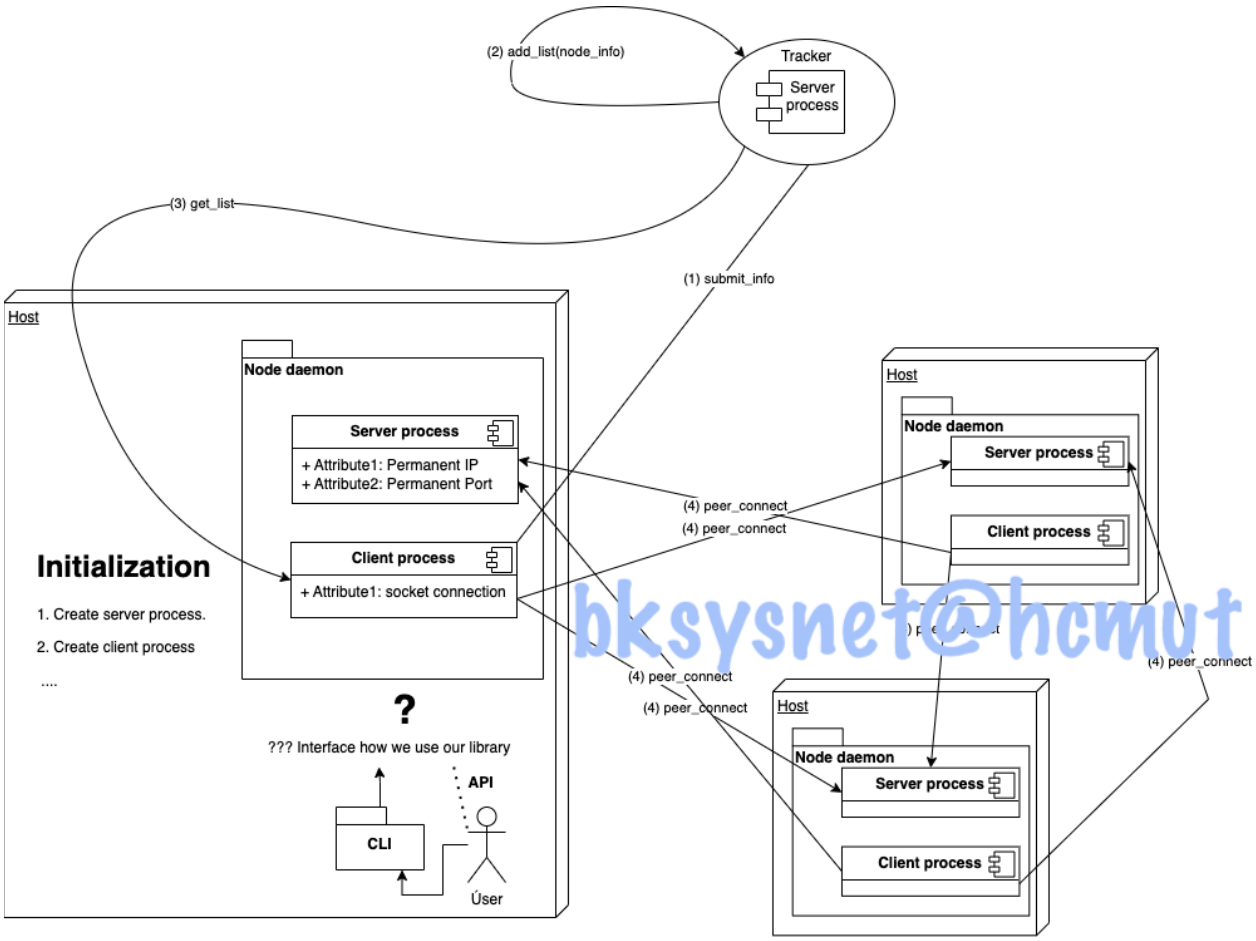
In this section, we introduce the steps of designing a sample NetApp application. This application employs the two paradigm: client-server paradigm and peer-to-peer paradigm of Chapter 2 Application layer.

3.1. Version 1 - A simple hybrid NetApp - Purely server process and client process combination

The design schematic is illustrated in Figure 1

The hybrid NetApp system involves two entities: the tracker (acted as server role) and the node (acted as peer). The hybrid NetApp operation has purely two, and separable, phase paradigms:

Phase1: client server paradigm
Step 1: send_info (peer_ip, peer_port)
Step 2: add_list (peer_ip, peer_port)
Step 3: get_list ()
Phase 2: peer-to-peer paradigm
Step 4: peer_connect(IP_x, Port_x)
Step 5: peer_transfer(data)



…

Figure 1. System design of simple hybrid NetApp

Design question: The remaining issue is how we (the user) interact with the created library/system.

3.2. Version 2 - Agency and multi tracker updating

To tackle the question in the previous section, the interface/interaction mechanism is provided as agency design in this section. We also update the design with multi (servers/) trackers. The updated design schematic is illustrated in Figure 2.

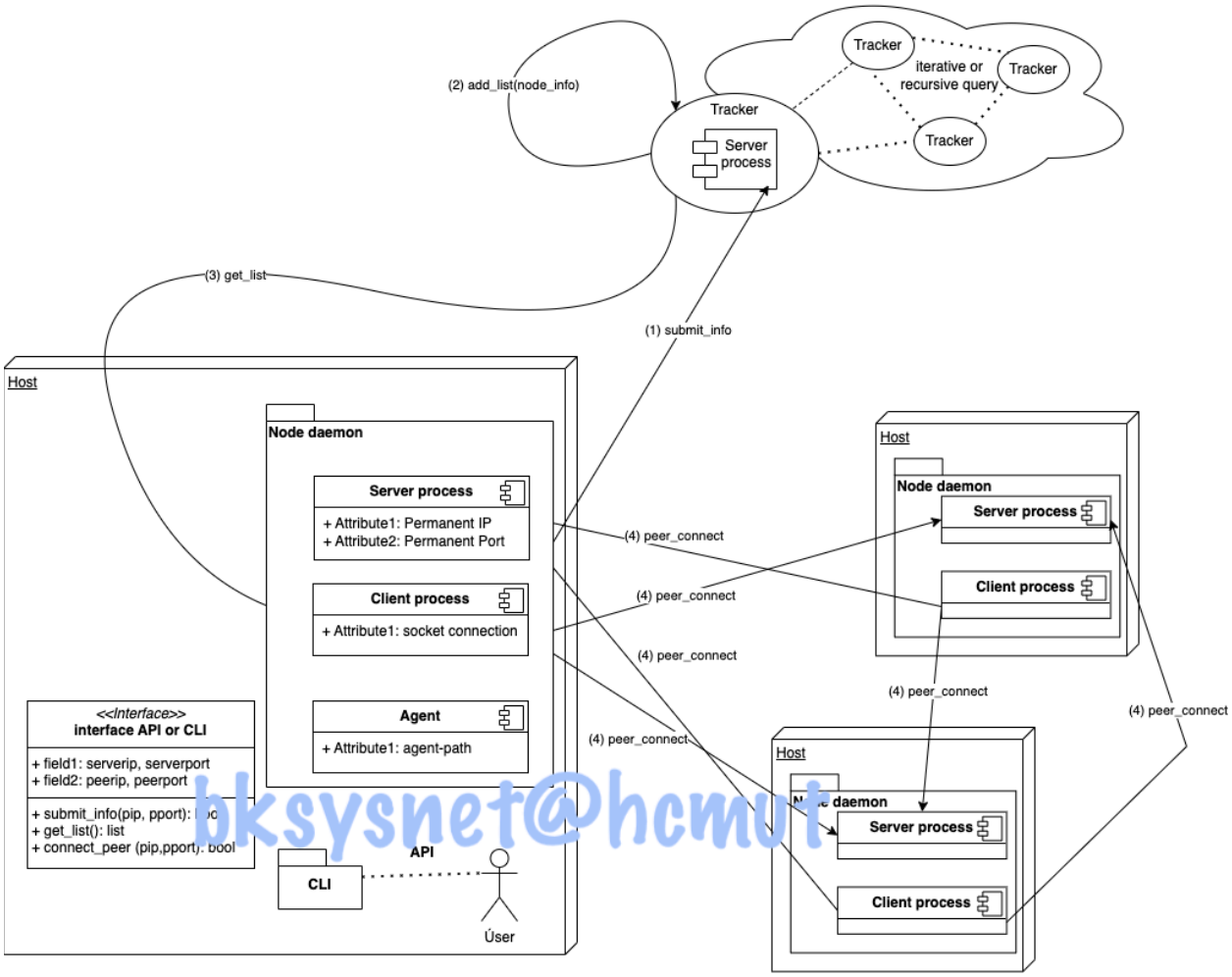


Figure 2 Updated design schematic with agency and multi (server) tracker

In this updated version:

Agent: on behalf of the user to issue the request and handle the response in a node. The multi-trackers enhance the scalability of the designed system.

The agency design

*(**Reminder** Email is a reference of agency design)*

The agent, who implements agency service, can provide API which is a machine friendly interface and CLI, command-line interface, which is user (or human) friendly interface. In our concept framework, GUI is for the end-user who is the users (aka. our developer user) needs to serve. GUI operations, all, in the

end can be served by calling an associated API or CLI command while the reverse direction is not guaranteed.

In a design point of view, agent and cli are processes (an instance of a program in execution) then the communication between them is IPC programming and there are various implementation strategies.

IPC - **I**nter**p**rocess **c**ommunication:

- using shared memory: how to encode/decode data, byte/stream access, struct wrapper, or open standard (Json, Google protocol buffer. etc.)
- using message passing: socket, Google RPC, REST API. Inside technical implementation, messages can be organized as in various patterns/architectures…
    - Asynchronous request-response pattern:
        - Request: send_request(ID, operation)
        - Acknowledgement: ECHO with ID
        - Asynchronous processing
        - Response: send_response(ID, answer)
        - Client handling
    - Claim check pattern (replace datastore by shorten form hash/identifier)
    - Competing consumer
    - Pub-sub by registering an interest of the specific pattern
    - Queue based (reader writer) (priority or load level)
    - Pipe and filter pattern
    - Scheduler agent
    - …

4. System implementation

4.1. Fundamental *multi-process multi-thread program implementation* with process communication

```python
import socket
from threading import Thread


def new_connection(addr, conn):
    print(conn)

def get_host_default_interface_ip():
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    try:
        s.connect(('8.8.8.8',1))
        ip = s.getsockname()[0]
    except Exception:
        ip = '127.0.0.1'
    finally:
        s.close()
    return ip


def server_program(host, port):
    serversocket = socket.socket()
    serversocket.bind((host, port))

    serversocket.listen(10)
    while True:
        conn, addr = serversocket.accept()
        nconn = Thread(target=new_connection, args=(addr, conn))
        nconn.start()


if __name__ == "__main__":
    #hostname = socket.gethostname()
    hostip = get_host_default_interface_ip()
    port = 22236
    print("Listening on: {}:{}".format(hostip,port))
    server_program(hostip, port)
```


server.py

```python
import socket
import time
import argparse
from threading import Thread

def new_connection(tid, host, port):
    print('Thread ID {:d} connecting to {}:{:d}'.format(tid, host, port))

    client_socket = socket.socket()
    client_socket.connect((host, port))

    # Demo sleep time for fun (dummy command)
    for i in range(0,3):
       print('Let me, ID={:d} sleep in {:d}s'.format(tid,3-i))
       time.sleep(1)

    print('OK! I am ID={:d} done here'.format(tid))


def connect_server(threadnum, host, port):

    # Create "threadnum" of Thread to parallelly connnect
    threads = [Thread(target=new_connection, args=(i, host, port)) for i in range(0,threadnum)]
    [t.start() for t in threads]

    # TODO: wait for all threads to finish
    [t.join() for t in threads]

if __name__ == "__main__":
    parser = argparse.ArgumentParser(
                        prog='Client',
                        description='Connect to pre-declard server',
                        epilog='!!!It requires the server is running and listening!!!')
    parser.add_argument('--server-ip')
    parser.add_argument('--server-port', type=int)
    parser.add_argument('--client-num', type=int)
    args = parser.parse_args()
    host = args.server_ip
    port = args.server_port
    cnum = args.client_num
    connect_server(cnum, host, port)
```


client.py

4.2. NetApp - Node server process

```
def new_server_incoming(addr, conn):
   print(addr)


def thread_server(host, port):
   print("Thread server listening on: {}:{}".format(host,port))

   serversocket = socket.socket()
   serversocket.bind((host, port))

   serversocket.listen(10)
   while True:
      addr, conn = serversocket.accept()
      nconn = Thread(target=new_server_incoming, args=(addr, conn))
      nconn.start
```

thread_server

4.3. NetApp - Node client process

```
def thread_client(id, serverip, serverport, peerip, peerport):
   print('Thread ID {:d} connnecting to {}:{:d}'.format(id, serverip, serverport))
```

thread_client

4.4. NetApp - Node agent

Various strategies can be applied to implement this module. We start with a small implementation using a shared memory sample in order to enrich our illustrations since we have already used message passing in socket connection between node and tracker (server).

```python
def thread_agent(time_fetching, filepath):
    print(filepath)

    with open(filepath, mode="w+", encoding="utf-8") as f:
        f.truncate(100)
        f.close()

    while True:
        with open(filepath, mode="r+", encoding="utf-8") as file_obj:
            with mmap.mmap(file_obj.fileno(), length=0, access=mmap.ACCESS_READ) as mmap_obj:
                text = mmap_obj.read()
                print(text.decode('utf-8'))
            file_obj.close()

        if [some action]:
            #TODO: the target action

        with open(filepath, mode="w+", encoding="utf-8") as wfile_obj:
            wfile_obj.truncate(0)
            wfile_obj.truncate(100)
            with mmap.mmap(wfile_obj.fileno(), length=0, access=mmap.ACCESS_WRITE) as mmap_wobj:
                text = f'done'
                mmap_wobj.write(text.encode('utf-8'))

            wfile_obj.close()

        time.sleep(time_fetching)

    exit()
```


thread_agent

The communication between Agent and CLI has been established, then the design of performing the processing task associated with each establishing communication is the work of agent task scheduling. Due to the scope of this computer network course CO3093 and CO3094, this remaining design is reserved for the readers. There are many reference designs including polling, watchdog, event-driven, interruptible handler, signal propagation etc. To keep it short and simple, we perform a baselined periodical fetching in our sample.

```python
import argparse
import mmap

def submit_info(args):
    #TODO: issue the submit command procedure
    with open(filepath, mode="r+", encoding="utf-8") as file_obj:
        file_obj.truncate(100)
        with mmap.mmap(file_obj.fileno(), length=0, access=mmap.ACCESS_WRITE, offset=0) as map_obj:
            text = f'submit_info'
            print(text)
            map_obj.write(text.encode("utf-8"))


def get_list(args):
    #TODO: issue the get_list command procedure
    with open(filepath, mode="r+", encoding="utf-8") as file_obj:
        file_obj.truncate(100)
        with mmap.mmap(file_obj.fileno(), length=0, access=mmap.ACCESS_WRITE, offset=0) as map_obj:
            text = f'get_list'
            map_obj.write(text.encode("utf-8"))

def ...(..):
    #TODO: implement all the command procedure

class NodeCLI:
    def __init__(self):
        self.parser = argparse.ArgumentParser(prog='NodeCLI', description='CLI for interacting with Node')
        self.add_subparsers()

    def add_subparsers(self):
        self.parser.add_argument('--func')
        self.parser.add_argument('--server-ip')
        self.parser.add_argument('--server-port', type=int)
        self.parser.add_argument('--agent-path')

    def run(self):
        args = self.parser.parse_args()
        f = globals()[args.func]
        f(args)

def main():
    cli = NodeCLI()
    cli.run()


if __name__ == "__main__":
    main()
```


node_cli

Discussion: With this design, it embeds a potential capability of auto-generated CLI through any API extension in the future but the authors leave the room for your creativity and try to explore it in your programmer career later. But that topic is still up-to-date even in 2023, 2024 !!!

4.5. Node initialization

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser(
                        prog='node',
                        description='Node connect to predeclared server',
                        epilog='<-- !!! It requires the server is running and listening !!!
    parser.add_argument('--server-ip')
    parser.add_argument('--server-port', type=int)
    parser.add_argument('--agent-path')

    args = parser.parse_args()
    serverip = args.server_ip
    serverport = args.server_port
    agentpath = args.agent_path

    peerip = get_host_default_interface_ip()
    peerport = 33357
    tserver = Thread(target=thread_server, args=(peerip, 33357))
    tclient = Thread(target=thread_client,
                     args=(1, serverip, serverport, peerip, peerport ))
    tagent = Thread(target=thread_agent, args=(2, agentpath))


    tserver.start()

    tclient.start()
    tclient.join()

    tagent.start()

    # Never completed
    tserver.join()
```


node.py

4.6. NetApp - Tracker server process

```python
import socket
from threading import Thread


def new_connection(addr, conn):
    print(addr)

    while True:
       try:
          data = conn.recv(1024)

          # TODO: process at tracker side
       except Exception:
          print('Error occured!')
          break


def get_host_default_interface_ip():
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    try:
       s.connect(('8.8.8.8',1))
       ip = s.getsockname()[0]
    except Exception:
       ip = '127.0.0.1'
    finally:
       s.close()
    return ip


def server_program(host, port):
    serversocket = socket.socket()
    serversocket.bind((host, port))

    serversocket.listen(10)
    while True:
        conn, addr = serversocket.accept()
        nconn = Thread(target=new_connection, args=(addr, conn))
        nconn.start()


if __name__ == "__main__":
    #hostname = socket.gethostname()
    hostip = get_host_default_interface_ip()
    port = 22236
    print("Listening on: {}:{}".format(hostip,port))
    server_program(hostip, port)
```


tracker

For multi-tracer with traversing mechanisms of iterative or recursive strategies, the work is a modified copycat of the previous designs and is reserved for the readers.