文本复制检测报告单(全文标明引文)

№:ADBD2020R_20200506153726451403811519　　检测时间:2020-05-06 15:37:26

检测文献:　用Python构建一个现代通信系统

作者:　胡英杰

检测范围:　中国学术期刊网络出版总库
中国博士学位论文全文数据库/中国优秀硕士学位论文全文数据库
中国重要会议论文全文数据库
中国重要报纸全文数据库
中国专利全文数据库
图书资源
优先出版文献库
大学生论文联合比对库
互联网资源(包含贴吧等论坛资源)
英文数据库(涵盖期刊、博硕、会议的英文数据以及德国Springer、英国Taylor&Francis 期刊数据库等)
港澳台学术文献库
互联网文档资源
源代码库
CNKI大成编客-原创作品库
个人比对库

时间范围:　1900-01-01至2020-05-06

检测结果

去除本人已发表文献复制比：0%　　跨语言检测结果：0%

去除引用文献复制比：0%　　总文字复制比：0%

单篇最大文字复制比：0%（ ）

重复字数：[0]	总字数：[11277]	单篇最大重复字数：[0]
总段落数：[1]	前部重合字数：[0]	疑似段落最大重合字数：[0]
疑似段落数：[0]	后部重合字数：[0]	疑似段落最小重合字数：[0]

指　标：疑似剽窃观点　疑似剽窃文字表述　疑似自我剽窃　疑似整体剽窃　过度引用

表　格：0　　公　式：没有公式　　疑似文字的图片：0　　脚注与尾注：0

（注释：无问题部分　文字复制部分　引用部分）

指导教师审查结果

指导教师:　刘苏扬

审阅结果:

审阅意见:　指导老师未填写审阅意见

1. 用Python构建一个现代通信系统

总字数：11277

相似文献列表

去除本人已发表文献复制比：0%(0)　　文字复制比：0%(0)　　疑似剽窃观点：(0)

原文内容

南京铁道职业技术学院
毕业论文
题目： 用Python构建一个
现代通信系统
作者： 胡英杰学号： 1756670231
二级学院： 通信信号学院
系 ： 通信系

专业： 铁道通信与信息化技术
班级： 铁道通信1702
指导者： 刘苏杨讲师
评阅者：
2020年 5月

毕业设计（论文）中文摘要

用Python构建一个现代通信系统

摘要现今通信系统正发生非常大的变化，物理底层的通信设计趋于保守。通信系统的设计今后将主要围绕软件、上层网络展开。微信、抖音对人们生活的变革便是很好的例子。本论文立足实践，坚持用代码说话，并附带一些讲解，坚决不弄虚作假，空谈理论。前1到2章介绍了主要的网络通信协议，后3到6章讲了一些编程的必要知识，最后7到10章开始动手搭建起整个系统。本系统使用了一种称为RESTFul API的技术。因此它几乎可以用于任何一种编程语言或设备，例如Android中的Java，iPhone中的Swift，以及网页上的Javascript。由于研究目的，最终作品将无法在工业上使用。但是它仍然很有价值，因为它是使用python建立通信系统的通用教程。

关键词 Python 服务器客户端 RESTFulAPI 多用户管理

目次

1 什么是通信
我们将在此讨论通信的定义。
1.1 在古代
这是人们相互交流的一种行为。一个人和另一个人说话。
自己和自己说话显然不算，因为你心里总是知道你自己在想什么，你不需要通信。
1.2 在当代
我会说它指“信息交换”这一个概念。
任何事物，如果你能用数据把它表示出来，传给别人，我们都算它是“通信“。
1.3 通信的原则
如果我不得不用一个词来形容它，那将是“稳定的”。
在传输过程中，您不应丢失任何原始发送的信息。
而且，它应该尽可能快。比如，如果您有紧急消息要发送给其他人，您肯定不想让其他人在十天后收到该消息。
而且，安全性也很重要。您不想将消息发送给错误的人，这会给您带来很多麻烦。
1.3.1 通信三原则
完整性：永不丢失信息。
速度：及时提供信息。
安全性：确保没有其他人能看到！
1.3.2 例外
但是仍然有一些通信类型仅遵循 1 或 2 种通信原则。
例如，广播。只要您发出消息，听众是否能完整持续地收到消息，这不关您的事 (因为听众可以自由决定什么时候听，什么时候不听)。
再比如，用于娱乐的广播不需要安全性，但是用于军事的广播确实需要保护其消息不被外人所知道。
1.4 通讯的分类
1.1.1 按终端设备数量区分
有些人喜欢用发送者或接收者的数量来分类通信。
如“一对一”与“一对多”。
“一对一”也称为“point to point”，“一对多”也称为“broadcast”。
1.4.2 按信息类型区分
另一些人喜欢根据所传输的信息类型对通信进行分类。
如“语音聊天”与“视频聊天”
1.5 机器语言
在上面的说明中，我们仅介绍了沟通的基础知识。或者人类如何理解这一点。
在本节中，我将讨论机器如何看待该问题。因为最终，在现代社会中，通信是基于计算机而不是人类。
对于计算机，他本身就是计算机，与之通信的计算机也是计算机。因此通信速度可能很高。至少比人类快。
然后剩下的问题是计算机如何相互理解，最重要的是，以正确的方式进行协作，以便信息可以自由正确地流动。
例如：如果我们有智能手机和计算机，我们如何相互通信？（智能手机实际上也是计算机。）
我们需要创建一种用于通信的方法，该方法称为协议。
再例如：如何确保我们发送的所有信息均被其他人正确接收？
我们需要创建某种方法来确保传输过程中信息的完整性，该方法是协议的一部分。
最终问题：世界上是否有最佳的通信协议？
没有！对于不同的场景，有不同的传输协议可供选择。
2 用于通信编程中的主要协议
我们将讨论在编程中可以使用多少种协议。
2.1 Ethernet
有人说这是物理协议。我不信。即使它们在铜线中运行，您仍然必须使用数字设备来处理它，并且该设备需要被编程。
2.2 IP
IP 代表 Internet 协议。
这是通过网络发送数据的一组规则。
但是对我来说，我认为它是某个特定计算机的身份标识，有了它，您就可以通过 Internet 访问世界上的任何计算机。
2.3 UDP
UDP 代表用户数据报协议。
使用 UDP，计算机应用程序可以将消息（在这种情况下称为数据报）发送到网络上的其他主机。
但是 UDP 不可靠。UDP 应用程序必须能够接受某些数据包丢失、乱序、错误或者重复。
UDP 通常会在流媒体、实时多人游戏和 IP 语音（VoIP）上使用。因为对于那些应用程序，丢失数据包通常不是致命的问题。
2.4 TCP
TCP 是指“传输控制协议”。

TCP 通过 IP 网络，在不同主机上运行的不同应用程序之间，提供可靠、有序且经过错误检查的字节流传输。诸如 World Wide Web，Email 之类的互联网应用程序都依赖于 TCP。

这是一个很流行的协议。许多其他的协议都基于 TCP 开发。

2.5 HTTP

HTTP 指超文本传输协议。

它用于在线表示信息。

我不能再强调它的重要性了。因为它是现代网络的基础。

网站是使用 HTTP 协议的最终产品。

2.6 HTTPS

它是 HTTP 的升级版本。它使用某种技术来确保 HTTP 传输的安全性。

因此，在接下来的十年中，如果您看到有人仍在使用纯 HTTP 协议，那将是不安全的。通过该协议传输的数据可能会泄露给坏人。

2.7 socket

从技术上讲，它不是协议，只是端点的名称。

实际上，套接字通常是指 IP 网络中的两个端点，它是一种用于一对一连接的术语。

2.8 WebSocket

有时人们不希望直接将套接字与系统编程语言（例如 Python）一起使用，而是希望用浏览器（网站）建立点对点连接。

这就是 WebSocket 的来源，它是用于在网站或浏览器上创建套接字连接的协议。

3 使用Python进行Socket通讯

让我们开始使用 Python 创建第一个通信应用程序。

伟大的征程起源于最基础的第一步。

3.1 一个简单的程序

3.1.1 服务器.py

```
# first of all import the socket library
import socket

# next create a socket object
s = socket.socket()
print("Socket successfully created")

# reserve a port on your computer in our
# case it is 12345 but it can be anything in a range of 0 to 65535
port = 12345

# bind the listener to Local Area Network
s.bind(('0.0.0.0', port))
print(f"socket binded to {port}")

# put the socket into listening mode
s.listen(5)
print("socket is listening")

# we run this server forever until we interrupt it or any error occurs
try:
    while True:
        # Establish connection with client.
        c, addr = s.accept()
        print('Got connection from', addr)

        # send a thank you message to the client.
        c.send(b'Thank you for connecting')
except Exception as e:
    print(e)
    # Close the connection with the client
    c.close()
```

在这里，我们先是 import 了一个 socket library，然后创建了一个 socket instance。接着把当前host 的 ip address 给了那个socket 实例。这个实例将会一直监听到来的请求。

当一个新的request到来时，这个实例会打印出对方的ip address，并且回复“Thank you for connecting”。

3.1.2 客户端.py

```
# Import socket module
import socket

# Create a socket object
s = socket.socket()

# Define the port on which you want to connect, it should be the same as server's
bound port
port = 12345

try:
    # connect to the server
    s.connect(('127.0.0.1', port))

    # receive data from the server
    print(s.recv(1024))
except Exception as e:
    print(e)
    # close the connection
    s.close()
```

在这里，我们同样创建了一个socket instance，只不过这一次我们没有监听，而是主动地向一个特定的ip address做了一次连接。并且最终我们打印了对方返回的message。

3.1.3 总结

从上面的两段代码中，我们可以知道，如果要创建通信，则需要先创建一个服务器，

该服务器首先侦听带有端口的 IP 地址，然后创建一个连接到确切 IP 地址和端口的客户端。

值得注意的一点是：如果server 比 client 后启动，那client就会连接不上server。

3.2 有关进程阻塞的问题

我们在这里遇到的问题是服务器将尝试永远运行，而不会自行停止。它将保持接受连接，并向所有想要连接的客户端发送“Thank you for connection”字符串。

在这种情况下，您只能预定义该通信处理。假设客户发送给您“How are you?”，您回答“I’m fine.”。

这并不酷，尤其是当您想与某人实时聊天时。因为在聊天时，必须有双向的信息流，一个人不仅需要发送消息，还必须从其他人那里接收消息。

3.2.1 线程

定义：在计算机科学中，线程是可以由调度程序（通常是操作系统的一部分）独立管理的最小化编程指令序列。

我只把它理解为可以让两段程序同时运行的一种技术罢了。

3.2.1.1 简单的例子

```
import logging
import threading
import time

def thread_function(whatever):
    logging.info(f"Thread: starting {whatever}")
    time.sleep(2)
    logging.info(f"Thread: finishing {whatever}")

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    logging.info("Main    : before creating thread")
    x = threading.Thread(target=thread_function, args=("yeah!",))
    logging.info("Main    : before running thread")
    x.start()
    logging.info("Main    : wait for the thread to finish")
    # x.join()
    logging.info("Main    : all done")
```

输出是：

```
22:16:00: Main      : before creating thread
22:16:00: Main      : before running thread
22:16:00: Thread: starting yeah!
22:16:00: Main      : wait for the thread to finish
22:16:00: Main      : all done
22:16:02: Thread: finishing yeah!
```

线程程序可以与主进程分开运行。我们可以使用此功能来创建能够进行双向通信的节点。

这里我使用了logging，是因为只有它才能跨线程打印字符串。

3.2.1.2 把它加到服务器.py

```
import socket
import threading

# thread function
def receive_and_print(c):
    while True:
        # data received from client
        data = c.recv(1024)
        if not data:
            print('Bye')
            break
        print('\nmessage received: ', str(data.decode('utf-8')))

    # connection closed
    c.close()


def Main():
    host = "0.0.0.0"
    port = 12345

    s = socket.socket()
    s.bind((host, port))
    print("socket binded to port", port)

    # put the socket into listening mode
    s.listen(5)
    print("socket is listening")

    # a forever loop until client wants to exit
    try:
        while True:
            # establish connection with client
            c, addr = s.accept()

            # Start a new thread
            t = threading.Thread(target=receive_and_print, args=(c,))
            t.start()

            while True:
                text = input("\n")
                c.send(text.encode("utf-8"))
    except Exception as e:
        print(e)
    # close the connection
    s.close()


if __name__ == '__main__':
    Main()
```

在这里，我们创建了一个名叫receive_and_print的function，它将会在接收到client的信息之后把它打印出来。

另外，我们在主loop循环里做了对一个ip地址的监听。

3.2.1.3 把它加到客户端.py

```
import socket
import threading


def receive_and_print(c):
    while True:
        # data received from server
        data = c.recv(1024)
        if not data:
            print('Bye')
            break
        print('\nmessage received: ', str(data.decode('utf-8')))
        print("\n")

    # connection closed
    c.close()


def Main():
    host = '127.0.0.1'
    port = 12345

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

    t = threading.Thread(target=receive_and_print, args=(s,))
    t.start()

    # greeting
    message = "yingshaoxo is awesome"
    s.send(message.encode('utf-8'))

    try:
        while True:
            # message sent to server
            text = input("\n")
            s.send(text.encode('utf-8'))
    except Exception as e:
        print(e)

    # close the connection
    s.close()


if __name__ == '__main__':
    Main()
```

在这里我们先是单独设置了一个function，用于打印接收到的message。然后在主loop线程里循环地发送信息。这里的信息取自terminal用户的文字输入。

3.2.1.4 总结

于是我们就这么愉快地实现了双向通信。发信者可以随时随地发送消息，接收者也可以随时随地发送消息，并且，在任何时候任何一方接收到了新消息，程序都会在屏幕上打印出来。

3.2.2 多进程线程不安全。

任何资深程序员都会同意这一点。这就是为什么我们需要一种更安全的编程技术来实现相同的目的：多进程。多进程是在单个计算机系统中使用两个或多个中央处理单元。该术语还指系统支持多个处理器或能分配任务给多个处理器。

有了多进程技术，我们同样可以在一个程序里运行两段独立的运算。

3.2.2.1 一个简单的例子

```
from multiprocessing import Process


def my_function(x):
    print(x**2)


if __name__ == '__main__':
    p = Process(target=my_function, args=(3,))
    p.start()
    print("3 to the power of 2 is: ")
```

The result is:

```
3 to the power of 2 is:

9
```

讲真的，我不觉得这个东西看起来有多么高大上或者神秘，我只是觉得它和线程操作类似，都需要你事先提供一个function。而这个function，会在另一个process里面运行。

3.2.2.2 把它加到服务器.py

```
from multiprocessing import Process
import socket


def receive_and_print(c):
    while True:
        data = c.recv(1024)
        if not data:
            print('Bye')
            break
        print('\nmessage received: ', str(data.decode('utf-8')))

    c.close()


host = "0.0.0.0"
port = 12345

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

s.listen(5)

try:
    while True:
        c, addr = s.accept()

        p = Process(target=receive_and_print, args=(c,))
        p.start()

        while True:
            text = input("\n")
            c.send(text.encode("utf-8"))
except Exception as e:
    print(e)

s.close()
```

同样的道理，我们在主进程里监听connection，并且接收terminal里用户发来的消息，并把消息发送给连接上来的client。接着如果client有发消息过来，我们就在子进程里把接收到的东西打印出来。

3.2.2.3 把它加到客户端.py

```
from multiprocessing import Process
import socket


def receive_and_print(c):
    while True:
        data = c.recv(1024)
        if not data:
            print('Bye')
            break
        print('\nmessage received: ', str(data.decode('utf-8')))

    c.close()


host = '127.0.0.1'
port = 12345

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

p = Process(target=receive_and_print, args=(s,))
p.start()

message = "yingshaoxo is awesome"
s.send(message.encode('utf-8'))

try:
    while True:
        text = input("\n")
        s.send(text.encode('utf-8'))
except Exception as e:
    print(e)

s.close()
```

客户端与服务器做的事情没什么区别，只不过变被动为主动，积极地连接上server、积极地先发送信息给server，最后打印出server返回来的信息。

3.2.2.4 总结

使用多进程的好处就是它更加稳定，因为不同的进程可以由不同的处理器处理。

现代计算机与古代计算机之间的一个很大不同就是运算性能得到了极大提升。这种提升很大程度上得益于多CPU构架。

除了以上的优点，多进程比多线程好的地方是：它能支持不同电脑之间互相交换计算资源。我的意思是如果你有一个庞大的数据库需要处理，当一台运算器无法处理它的时候，你可以把大数据变成小数据，运用多进程的手法，把数据分布在多台不同的运算器上。这样，就能同步的在多台机器上对某个数据做处理。节省了时间，同时也提高了运算性能。据我所知，大型电影的3D特效渲染就是这么做的。几千台机器同时工作，共同创造出优美的好莱坞大片。

4 异步通信

人类拥有的大多数交流都是同步的。一个说，一个听。如果他们无秩序地谈话，那将是一场争吵。

但是实际上，大多数基于计算机的通信系统都是异步的。即使它们看起来是同步的，也并非如此。通信传输过程需要时间，因此这个世界上没有 100%的实时性。

在本章中，我们将讨论可在 Python 中使用的异步操作，以便我们稍后可以使用它来构建通信系统。

4.1 asyncio

asyncio[1]是用于制作异步程序的 Python 库。

4.1.1 一个简单的例子

```
import asyncio

async def func1():
    await asyncio.sleep(1)
    print('Hello ...')

async def func2():
    await asyncio.sleep(2)
    print('... World!')

async def main():
    await asyncio.gather(
        func2(),
        func1()
    )

asyncio.run(main())
```

输出是:

```
Hello _
_ World!
```

从以上结果可以看出，即使将 function2 放在 function1 之前，程序仍会先打印出 function1 字符串“Hello”，然后打印出 function2 字符串“World”。得出这个结果，是因为两个函数同时启动，function1 的延迟时间比 function2 的延迟短，1 小于 2。

4.2 一个简单的服务器和客户端程序

4.2.1 服务器

```
import asyncio

async def handle_echo(reader, writer):
    data = await reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')

    print(f"Received {message!r} from {addr!r}")

    print(f"Send: {message!r}")
    writer.write(data)
    await writer.drain()

    print("Close the connection")
    writer.close()

async def main():
    server = await asyncio.start_server(
        handle_echo, '127.0.0.1', 8888)

    addr = server.sockets[0].getsockname()
    print(f'Serving on {addr}')

    async with server:
        await server.serve_forever()

asyncio.run(main())
```

当我们写这样一个程序的时候，我们很容易感觉到，它更加简洁了，基本上，我们只是定义了一个function，所有事情就都解决了。

这不得不让人倾佩Python带给人的便捷。

4.2.2 客户端

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()

asyncio.run(tcp_echo_client('Hello World!'))
```

另外，通过这几个server和client 的例子，我们能清晰地感受到，在数据线路上传输的东西绝对不是纯文本，而是bytes

stream。

这也是为什么我们需要不断地做decode和encode。Decode是把bytes转为string，encode是把string转为bytes。

4.2.3 总结

所有这些代码背后的基本思想与我们之前看到的相同。

首先，我们必须创建一个服务器，该服务器定义了我们希望客户端连接的 IP 地址和端口。

然后，我们创建一个不断访问 IP 地址以建立连接的客户端，通过该连接，我们可以发送消息并从双方接收消息。

4.3 是否保持连接

大多数人以为他们的通讯连接一直都在，但事实并非如此。

没有任何连接是永远存在的。因为我们正在使用 Internet。Internet 本身是基于数据包的，这意味着基于 Internet 传输的所有内容都应进行拆分，并按部分交付。

这就是数字通信或现代通信的工作方式，他们不会一直保持连接状态，但是他们将尝试尽可能频繁地连接服务器。

就像，如果客户端每 1 秒连接一次服务器以获取信息。对于一个人去获得不重要的消息就足够了。

我说不重要是因为没有人可以在 1 秒钟内采取大动作。因此，延迟 1 秒对人类而言并不重要。

5 基于 HTTP 的通信

5.1 什么是 API

API 代表“应用程序编程接口”，即使用编程方法来创建接口，以便另一种编程语言或另一种计算设备可以使用它来进行一些计算或从中获取一些信息。 API 在云计算中很常见，世界上几乎所有云服务都向其客户提供其 API。而且，客户端需要使用 API 与服务器进行通信。在这种情况下，API 的作用就像连接多个用户和中央服务器的桥梁。

5.2 如何使用 Python 创建 API 服务器

Flask 是执行此操作的常用工具。Django 也很好。但是为了简化和支持异步编程，我将使用类似于 Flask 的工具“sanic[2]”作为我们使用的工具。

5.2.1 API 服务器示例

```
from sanic import Sanic, response

app = Sanic(name="contacts_app")

# Create some test data for our catalog in the form of a list of contact.
contacts = [
    {
        'username': "yingshaoxo",
        'friends': ["god", "angles"]
    },
    {
        'username': "A",
        'friends': ["A's friend 1", "A's friend 2"]
    },
    {
        'username': "B",
        'friends': ["B's friend 1", "B's friend 2"]
    },
]


@app.route('/', methods=['GET'])
async def home(request):
    return response.html('''<h1>awesome people do awesome things.</h1>''')


@app.route('/api/v1/contacts/all', methods=['GET'])
async def api_all(request):
    return response.json(contacts)


@app.route('/api/v1/contacts', methods=['GET'])
async def api_username(request):
    # Check if an username was provided as part of the URL.
    # If username is provided, assign it to a variable.
    # If no username is provided, display an error in the browser.
    if 'username' in request.args:
        username = str(request.args['username'][0])
    else:
        return response.text("Error: No username field provided. Please specify an
username.")

    # Create an empty list for our results
    results = []

    # Loop through the data and match results that fit the requested username.
    # Username are unique, but other fields might return many results.
    for contact in contacts:
        if contact['username'] == username:
            results.append(contact)

    # Use the jsonify function from Flask to convert our list of
    # Python dictionaries to the JSON format.
    return response.json(results)


if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000, debug=True)
```

5.2.2 测试该服务器

现在，如果你打开浏览器，访问 http://0.0.0.0:8000，你将会看到：

```
awesome people do awesome things.
```

如果你访问 http://0.0.0.0:8000/api/v1/contacts?username=A，则会看到：

```
[
  {
    "username": "A",
    "friends": [
      "A's friend 1",
      "A's friend 2"
    ]
  }
]
```

如果你访问 http://0.0.0.0:8000/api/v1/contacts?username=B，则会看到：

```
[
  {
    "username": "B",
    "friends": [
      "B's friend 1",
      "B's friend 2"
    ]
  }
]
```

为什么会出现这种情况？很明显是因为我们读取了URL中的参数。在这里，这个参数是“username”。对于不同的user，我们返回了不同的数据。

这种通过url的变化改变返回值的方式在API设计中非常常见。因为它一目了然。用户可以对自己进入到了什么功能的页面非常清楚。

6 基于数据库的通信

6.1 什么是 MongoDB

这是一个数据库框架，我们可以使用它来保存用户的数据，包括用户帐户和聊天记录。

为什么选它呢？因为它比大多数传统数据库（例如 SQL）更简单。而且它与JSON更兼容。

另外，该数据库的好处是您不必预先定义数据结构。数据库一个集合中的每个元素都可以具有不同的数据结构。这使您可以灵活快速地尽快制作软件产品原型。

6.2 如何在 Python 中使用 MongoDB

6.2.1 PyMongo

“PyMongo[4]”是操作 MongoDB[3] 的 python 库。通过使用它，我们可以轻松地处理数据库。

6.2.2 简单的例子

```
from pymongo import MongoClient
import datetime
import pprint

client = MongoClient('mongodb://localhost:27017/')
db = client.test_database
posts = db.posts
posts.delete_many({})

post = {
    "author": "Mike",
    "text": "My first blog post!",
    "tags": ["mongodb", "python", "pymongo"],
    "date": datetime.datetime.utcnow()
}
post2 = {
    "author": "yingshaoxo",
    "text": "My first latex book!",
    "tags": ["mongodb", "python", "pymongo", "latex"],
    # "date": datetime.datetime.utcnow()
}

posts.insert_one(post)
posts.insert_one(post2)

post = posts.find_one({'author': 'Mike'})
print("Find one item from the database:")
pprint.pprint(post)
print('\n'*2)

posts.find_one_and_update({"_id": post.get("_id")}, {"$set": {"new_field":
"88888888888888888888888888888888888888888888888888"}})

print("Find multiple items from the database:")
for post in posts.find():
    pprint.pprint(post)
    print()
print('\n'*2)

print(client.list_database_names())
```

在此示例中，首先，我们定义了一个“posts”。 “posts”是一个集合，或者，如果您愿意，也可以将其视为列表。

然后我们定义了两个post，它们是dict对象。 我们将它们插入了“posts”集合。

然后，我们尝试通过使用其部分数据“author==Mike”来查找其中一个post。

然后，我们为该post添加了一个新属性。 该属性的值是一系列数字8。

最后，我们迭代“posts”集合以打印出该列表中的每个post。

6.3 为什么我们要使用数据库

为什么我们必须使用数据库？这是因为我们要永久保存用户的数据，包括用户帐户本身。您不希望您的用户在服务器计算机断电后发现其帐户已消失。

我们可以将所有数据保存在计算机内存中，但是由于以下原因，这是不实际的：

(1) 一台计算机的内存是有限的。我从未听说过计算机的内存大于计算机的磁盘空间。

(2) 当计算机关闭时，保存在计算机内存中的所有数据都将消失。那不是好事。

(3) 你可以自己写一个脚本来处理数据，但很快你会发现你自己写的东西效率太低。

(4) 方便我们共享以及备份数据。

7 用户管理系统

该系统是为人类打造的，地球上有很多人。因此，系统应具有管理大量用户的机制。假设您有一个服务器，它就像可以创建用户、授权用户的授权机构。您怎么知道一个连接来自您创建的用户而不是其他用户？最简单的方法是请求该用户提供密码，如果该密码与您的数据集中存储的密码匹配，那么它就是您之前创建的用户。

7.1 用户管理的基本思想

(1) 注册: sign up

(2) 登陆: log in

(3) 注销: log out

(4) 账号删除: account deletion

7.2 代码实现

```
from sanic import Sanic, response
from pymongo import MongoClient


client = MongoClient('mongodb://localhost:27017/')
db = client.my_app
users = db.users


app = Sanic(name="user_manager")


@app.route('/api/v1/usermanagement', methods=['POST'])
async def user_management(request):
    data = request.json
    result = {"status": "ok"}
    if data:
        action = data.get("action")
        body = data.get("body")
        if action and body:
            if ("username" in body) and ("password" in body):
                username = body.get("username")
                password = body.get("password")
                the_user_we_found = users.find_one({"username": username, "password":
password})
                if action == "signup":
                    if the_user_we_found == None:
                        new_user = {
                            "username": username,
                            "password": password,
                        }
                        users.insert_one(new_user)
                        result["status"] = "ok: account has been created"
                    else:
                        result["status"] = "wrong: account has already been created"
                elif action == "login":
                    if the_user_we_found == None:
                        result["status"] = "wrong: no account or wrong password"
                    else:
                        result["status"] = "ok: logged in"
                elif action == "delete":
                    if the_user_we_found == None:
                        result["status"] = "wrong: no account or wrong password"
                    else:
                        users.find_one_and_delete({"_id":
the_user_we_found.get("_id")})
                        result["status"] = "ok: account has been deleted"
            else:
                result["status"] = "wrong: no username or password received"
        else:
            result["status"] = "wrong: no action or body received"
    else:
        result["status"] = "wrong: no json data received"
    return response.json(result)


if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000, debug=True)
```

7.3 注册

当我们收到“用户管理”请求时，我们首先检查该请求是否针对“注册”提出。如果是，我们将检查该请求是否包含用户名和密码。如果是这样，我们将检查数据库以查看数据库中是否没有该用户。如果是这样，我们将创建一个新帐户。

7.4 登陆

当我们获得“用户管理”请求时，我们首先检查该请求是否用于“登录”。如果是，我们将检查该请求是否包含用户名和密码。如果是这样，我们将检查数据库以查看数据库中是否有该用户。如果是这样，我们可以说它是我们的用户。

7.5 注销

这对于我们的应用来说并不重要。所以我不会实现它。

但有一点是非常清楚的，每次在用户注销之后，用户必须再输入一次账号密码，重新登陆，才能继续使用我们的服务。

7.6 账号删除

收到“用户管理”请求时，我们首先检查该请求是否用于“删除”。如果是，我们将检查该请求是否包含用户名和密码。如果是这样，我们将检查数据库以查看数据库中是否有该用户。如果是这样，我们将从数据库中删除该用户。

8 通信加密

数据需要加密。否则，任何入侵通信系统或监视互联网大门的人都可以得到您对同事所说的话。

更糟的是，如果您是一家公司的老板。您绝对不想泄漏您经营企业的秘密。

因为这将使您损失数十亿美元甚至更多。

8.1 级别一: HTTPS

仅通过网络传输密码而没有任何保护是不安全的。拥有网络物理访问权限的任何人都有机会窃取用户密码。

这就是为什么我们需要使用 HTTPS[5]。

只要我们可以确保手中的通信设备安全，就可以确保 HTTPS 级加密是安全的。

HTTPS 用于保护数据包传输级别的安全。

8.2 级别二: Sha256

但是，如果我们手中的通讯设备不安全怎么办？如果存储在我们设备中的数据可以被我们安装的任何应用程序读取怎么办？那可能是个大问题。

甚至还有，如果黑客入侵服务器怎么办？

在服务器或本地设备上以纯文本格式保存我们的用户密码可能是一场灾难。

在这种情况下，我想介绍一下我自己的加密用户密码的方法：使用 sha256[6]。

8.2.1 服务器需要做的事

(1) 定义用户数据结构

```
{
    username:,
    sha256_password:,
    temp_token:,
}
```

(2) 每当用户登录时，我们都会给用户一个临时令牌，该令牌是随机字符串。

(3) 每当用户想要发送或接收消息时，他必须提供服务器之前给他的临时令牌。

(4) 任何传输都应在 https 下进行，并且密码应为真实密码的哈希字符串。

8.2.2 客户端需要做的事

(1) 获取其密码的 sha256 代码。加些盐会更好。(盐指随机字符串，可以增大破解的难度)

```
import hashlib
m = hashlib.sha256()
m.update(b"password")
print(m.hexdigest())
```

(2) 将其用户名和哈希密码发送到服务器以获取临时令牌。

(3) 使用该临时令牌与服务器通信，因此服务器将知道是他。

8.2.3 Server.py

```
from sanic import Sanic, response
from pymongo import MongoClient
import hashlib
import random
import string


client = MongoClient('mongodb://localhost:27017/')
db = client.my_app
users = db.users


app = Sanic(name="user_manager")


@app.route('/api/v1/usermanagement', methods=['POST'])
async def user_management(request):
    data = request.json
    result = {"status": "ok"}
    if data:
        action = data.get("action")
        body = data.get("body")
        if action and body:
            if ("username" in body) and ("sha256_password" in body):
                username = body.get("username")
                sha256_password = body.get("sha256_password")
                the_user_we_found = users.find_one({"username": username,
"sha256_password": sha256_password})

                if action == "signup":
                    if the_user_we_found == None:
                        new_user = {
                            "username": username,
                            "sha256_password": sha256_password,
                        }
                        users.insert_one(new_user)
                        result["status"] = "ok: account has been created"
                    else:
                        result["status"] = "wrong: account has already been created"
                elif action == "login":
                    if the_user_we_found == None:
                        result["status"] = "wrong: no account or wrong password"
                    else:
                        random_string = ''.join(random.choice(string.ascii_lowercase)
for i in range(10))
                        random_string += username
                        temp_token =
hashlib.sha256(random_string.encode("utf-8")).hexdigest()
                        users.find_one_and_update({"_id":
the_user_we_found.get("_id")}, {"$set": {"temp_token": temp_token}})
                        result["status"] = "ok: logged in"
                        result["result"] = {"temp_token": temp_token}
                elif action == "delete":
                    if the_user_we_found == None:
                        result["status"] = "wrong: no account or wrong password"
                    else:
                        users.find_one_and_delete({"_id":
the_user_we_found.get("_id")})
```

8.2.4 Client.py

```
from user_management import app
import json
from logzero import logger
import hashlib


# clean database
from pymongo import MongoClient
client = MongoClient('mongodb://localhost:27017/')
db = client.my_app
users = db.users
users.delete_many({})


username = "yingshaoxo"
password = "hello"

salt = username[(len(username)//2):]
sha256_password = hashlib.sha256((password+salt).encode("utf-8")).hexdigest()

temp_token = ""


def test_user_sign_up():
    body = {
        "username": username,
        "sha256_password": sha256_password,
    }
    data = {
        "action": "signup",
        "body": body,
    }
    request, response = app.test_client.post("/api/v1/usermanagement",
data=json.dumps(data))

    logger.debug(type(response.json))
    logger.debug(response.json)
    assert "ok" in response.json.get("status"), f"response.json['status']"


def test_user_log_in():
    global temp_token
    body = {
        "username": username,
        "sha256_password": sha256_password,
    }
    data = {
        "action": "login",
        "body": body,
    }
    request, response = app.test_client.post("/api/v1/usermanagement",
data=json.dumps(data))

    logger.debug(type(response.json))
    logger.debug(response.json)
    assert "ok" in response.json.get("status"), f"response.json['status']"
    temp_token = response.json.get("result").get("temp_token")
    logger.debug(temp_token)
```

8.2.5 总结

完成上述操作后，由于本地端和服务器端均未存储任何明文密码，因此我们可以成功验证用户身份而无需担心数据泄漏。即使黑客获得了 sha256 密码，他们仍然无法计算真实密码，也无法将 sha256 密码视为真实密码的替代，因为我们添加了 salt。

而且，如果黑客获取了所有存储在服务器上的数据，我们要做的就是更换服务器，更换盐，并通过电子邮件地址验证我们的用户，然后一切都会恢复正常。

为什么使用电子邮件进行帐户恢复？由于电子邮件是由第三方管理的，因此黑客无法访问它。(或者说黑客同时黑掉第三方邮件系统的可能性不大)

数据泄漏后，一些公司希望要求其用户更改密码。这不是必需的。

9 API设计

在开始构建 UI 之前，我们必须先设计 API。

9.1 验证

这是一个简单的 API，供潜在用户了解它是否是服务器中的真实用户。

```
@app.route('/api/v1/verify', methods=['POST'])
async def verify(request):
    data = request.json
    result = {"status": "wrong, you have to sign up or log in"}
    if data:
        username = data.get("username")
        temp_token = data.get("temp_token")
        if username and temp_token:
            the_user_we_found = users.find_one({"username": username, "temp_token":
temp_token})
            if the_user_we_found != None:
                result["status"] = "ok"
    return response.json(result)
```

如果我们的客户的状态为“错误”，则他可能需要重新登录。

9.2 获取联系人

我们使用此 API 来获取可以与之交谈的人员的列表。

这里的逻辑很简单，我们在服务器中获取所有用户的姓名，但要求该列表的用户除外。

```
@app.route('/api/v1/contactlist', methods=['POST'])
async def get_contact_list(request):
    data = request.json
    result = {"status": "wrong"}
    if data:
        username = data.get("username")
        temp_token = data.get("temp_token")
        if username and temp_token:
            the_user_we_found = users.find_one({"username": username, "temp_token"
temp_token})
            if the_user_we_found == None:
                result["status"] = "wrong: user doesn't exist"
            else:
                userlist = users.find({"_id": {"$nin":
[the_user_we_found.get("_id")]}})
                #userlist = users.find({"username": {"$nin": [username]}})
                name_list = []
                for user in userlist:
                    name_list.append(user.get("username"))
                result["result"] = {"name_list": name_list}
                result["status"] = "ok"
    return response.json(result)
```

9.3 发送消息

我们需要创建一个 API 来处理消息发送。

9.3.1 步骤 1

我想让这个 API 具有 3 个参数：

⑴from:此消息来自哪里。

⑵to: 我们需要将此消息发送给哪个目标用户.

⑶text: 我们要发送的消息。

处理“/api/v1/send”请求的函数:

```
username = data.get("username")
 temp_token = data.get("temp_token")
 body = data.get("body")
 if username and temp_token and body:
     the_user_we_found = users.find_one({"username": username, "temp_token":
temp_token})
     if the_user_we_found != None:
         from_ = username
         to = body.get("to")
         text = body.get("text")
         if to and text:
             result["status"] = "ok"
             update_an_conversation(from_, to, text)
```

9.3.2 步骤 2

另外，我们需要创建一个新的数据结构来保存聊天记录。

如果 from=a-user，to=b-user，文本 =“hi”，则数据结构为：

```
[
    {
        "conversation_name": "a-user_b-user",
        "history": [
            {
                "date": "2020.3.29",
                "from": "a-user",
                "to": "b-user",
                "text": "hi",
            }
        ]
    }
]
```

这样，我们可以基于两个用户的姓名、日期，获得一系列聊天记录。有了这些信息或者数据，建立聊天视图就不会困难了

。

这是处理此过程的函数：

```
def update_an_conversation(from_, to_, text):
    date = datetime.datetime.utcnow()
    conversation_name = "_".join(list(sorted([from_, to_])))
    conversation = conversations.find_one({"conversation_name":
conversation_name})
    if conversation == None:
        create_an_conversation(from_, to_, text, date)
    else:
        conversations.find_one_and_update({"_id": conversation.get("_id")}, {
            "$addToSet": {
                "history": {
                    "date": date,
                    "from": from_,
                    "to": to_,
                    "text": text,
                }
            }
        })

def create_an_conversation(from_, to_, text, date):
    conversation_name = "_".join(list(sorted([from_, to_])))
    conversation = {
        "conversation_name": conversation_name,
        "history": [
            {
                "date": date,
                "from": from_,
                "to": to_,
                "text": text,
            }
        ]
    }
    conversations.insert_one(conversation)
```

9.4 接收信息

我们需要定义一个 API，以允许用户获取其聊天记录。

通过分析聊天记录，用户还可以知道他们是否收到其他人的新消息。

对于客户端而言，这非常简单，因为如果新的聊天记录与旧的聊天记录不同，则意味着它收到了新消息。

9.4.1 步骤 1

我想让这个 API 有 1 个参数：target

当目标 ==“空字符串”时，我们返回与此用户相关的所有聊天记录。

当目标 ==“另一个用户”时，我们仅返回另一个用户与该用户之间发生的聊天记录。

我用到的代码是：

```
target = body.get("target")
if target == "" or target:
    got = get_conversation(username, target)
    if got == None:
        result["status"] = f"wrong: no record with {username}_{target}"
    else:
        result["status"] = "ok"
        result["result"] = got
```

9.4.2 步骤 2

此过程全都涉及查找与此用户相关的历史聊天记录。

如果用户提供了目标用户，我们可以得到一个唯一的字符串，表示存储在数据库中的会话之一。

或者，如果用户未向我们提供特定的目标用户，我们仍然可以使用正则表达式来获取包括该用户的所有对话。

我用到的代码是：

```
def get_conversation(username, target):
    data = None
    if target == "" or target:
        if target != "":
            conversation_name = "_".join(list(sorted([username, target])))
            conversation = conversations.find_one({"conversation_name":
conversation_name})
            if conversation != None:
                data = [conversation]

        else:
            regex = "(" + username+"_.*)|(.*_"+username+")"
            many_conversations = conversations.find({"conversation_name":
{"$regex": regex}})
            if many_conversations != None:
                result = []
                for conversation in many_conversations:
                    result.append(conversation)
                data = result
    return data
```

10 UI设计

“UI”表示“用户界面”。

没有用户界面，我们的用户将无法使用我们的应用程序。

这对普通人来说非常重要。（虽然很多精英程序员对此不屑一顾，他们更喜欢无尽的黑框框: terminal 或者 shell）

10.1 选择一个框架

有很多 UI 框架可供我们使用。特别是在网站方面。

我们可以使用 WPF 在 Windows 上进行设计，也可以使用 Flutter 在 Android 上进行设计，更可以使用Swift在IOS上设计。

但这对我来说实在太繁杂了。我只想有一个简单的 UI，可以使我的论文更加可信。

因此，我将仅使用 vue.js[7] 来完成此工作。

另外，由于界面不是本论文的重点，Python 才是，所以我不会把界面的代码写上来。

10.2 主要页面

- 首页: 在此页面中，我们可以注册或登录
- 联系人界面: 在此页面上，我们可以看到可以联系的人
- 聊天界面: 在此页面中，我们可以向他人发送消息，同时也能看到别人发过来的消息

10.2.1 首页

图10-1 首页

首页为了突出简洁性，并没有增设过多的按钮，首先一个问好，突出对使用者的诚意，然后腾出两个文本输入框，方便用户输入自己的名字和密码。最后，右下角再加上两个天蓝色的小按钮，鲜明地显示出用户最终需要点击的两个按钮：注册与登录。

我们当然可以把注册与登录两个功能放在一起。比如如果数据库中没有该用户，就帮助它注册。如果数据库中有该用户，就帮助他登录。虽然这看起来很智能，但实际上会产生很多不必要的麻烦，比如：已注册用户偶然间输错了他的密码，可是系统帮他注册了一个新账号，这很明显是对服务器资源地极大浪费。

10.2.2 联系人界面

图10-2 联系人界面

在这个界面，我能想到的最多的东西就是如何优雅而又完美地显示出一个个人的名字。

首先，人都是感性动物，通常我们更擅长于记住人的脸比起记住人名。所以在每一行的开始，我都加了一个头像，方便人们通过视觉识别快速地找到联系人。

另外，找到联系人后的功效性也必须凸显出来，我到底是想给这个人打电话？还是给这个人发短信？一个良好的功能性图标的作用这里就显示出来了。由于我们这里只有发信息的功能，所以每行的右侧就只有一个图标。

10.2.3 聊天界面

聊天界面我是仿造腾讯QQ做出来的，观察QQ，不难发现，对于每一个聊天视图，顶部永远显示的是聊天对象的名字，并且这个名字是居中对齐的。

然后则是聊天记录，上面显示有历史上你和别人聊天的信息。右边是对方说的话，左边是你说的话。

按道理来讲，两边的聊天记录应该用不同的颜色标识出来。但是由于我们是在写论文、做实验，这一繁琐的过程就可以掠过不做。

10.2.3.1 User 1

图10-3 聊天界面1

10.2.3.2 User 2

图10-4 聊天界面2

结论

利用Python 制作一个现代通信系统是可行的。不仅实时性可以得到保证，安全性也能得到充分保障。至于更多的通信特性，如语音、视频，只需用一些特定的编码技术把二进制文件转为文本，然后再进行传输即可。

但我们也需要看到用Python制作通信系统的局限性：我们不能直接用Python制作一个优良的用户操作界面。我们必须得把Python与其它一些编程语言如Javascript结合起来，才能很好地完成通信系统构建这个任务。

最后，有人可能会质疑Python的效率问题，我觉得这是一个不必要的问题。世界上还有很多解释型编程语言，如PHP，它们也有效率问题，但它们仍然在现代网络建设工程中大放异彩。并且随着计算机运算能力的不断提升，阻碍通信软件技术发展的只能是开发效率，而不是运行效率。

致谢

感谢谷歌以及开源社区所作出的贡献，使得每个人在这个世界上都能平等地获取知识。
感谢导师刘苏杨对本人的指导，一篇好的论文需要精心地修改。
同时感谢我的父母，他们在我成长阶段给了我很多关心与鼓励，并让我能在一个相对自由的环境学习、成长。
在论文撰写期间，多亏有父母的帮助，让我在家里不至于忍受饥饿，特此致谢。
最后，感谢党和国家对我的栽培，不经历9年义务教育，我是绝对写不出这样一篇中文论文的，特此致谢。
参考文献
[1] Python社区，Python Documentation.2019: docs.python.org/3/library/asyncio.html
[2] Sanic 社区，Sanic Documentation.2019: sanic.readthedocs.io/en/latest
[3] MongoDB 社区，MongoDB Documentation.2019: docs.mongodb.com
[4] PyMongo 社区，PyMongo Documentation.2019: pymongo.readthedocs.io/en/stable
[5] Wiki 社区，Wikipedia HTTPS Page.2019: en.wikipedia.org/wiki/HTTPS
[6] Python 社区，Hashlib Documentation.2019: docs.python.org/3/library/hashlib.html
[7] Vuejs 社区，Vuejs Documentation.2019: vuejs.org/v2/guide
[8] Vue Material 社区，Vue Material Documentation.2019: vuematerial.io/getting-started