基于Bento的Tor网络功能扩展

姚宣骋 肖真然 冯璇

Dec. 2022

1 项目简介

Tor网络为全世界用户提供强大的匿名访问工具，但直接在Tor中部署新服务却非常困难。Reininger、Arora等人在其文章（2021）[1]中介绍了一种可在中继节点上运行复杂功能的体系结构Bento，并对其设计、实现和应用进行了详细展示。Bento在实时的Tor网络上运行，并不直接修改Tor，而是对Tor网络的出口节点作一定扩展，以运行新的函数。本项目希望利用Bento，复现文章中的部分函数，并实现对server服务器的流量监控功能。

本项目的环境需求为Ubuntu18.04和Python3.6.x。

2 原理简述

2.1 Tor（The Onion Router）：匿名访问

Tor（The Onion Router）是第二代洋葱路由（onion routing）的一种实现，用户通过Tor可以在因特网上进行匿名交流。

2.1.1 代理通信（Proxied Communication）

Tor网络可帮助用户在一定程度上实现匿名访问，其实质是一种代理节点快速动态变化的三重加密代理。

一条Tor链路中通常有三个中继：一个入口节点（与源节点进行通信），一个中间节点，和一个出口节点（与目标节点进行通信）；它们均由源节点选择，并不断动态变化。在传输过程中，真实数据像洋葱一样被一层层被加密，这也是Tor被命名为洋葱路由的原因。

一段会被定时拆除的路径如图1所示。A、B、C为三个中继节点，key_A、key_B、key_C分别为三个节点与信息发送者Bob共享的加密密钥。当Bob向Alice发送数据时，先将数据Data用key_C加密，再用key_B加密，再用key_A加密，然后发往节点A。节点A解开一层加密，发往节点B。节点B解开一层加密，发往节点C。节点C解开一层加密，得到Bob发往Alice的消息，最终发送给Alice。如此，一段较短时间内，Alice只能知道数据来自节点C（即出口节点，exit node），节点C只能知道有数据从节点B发送到了Alice，节点B只能知道有数据从节点A（即入口节点，introduction node）发送到了节点C，节点A只能知道Bob访问了节点B。快速动态变化的三重代理使得攻击者难以通过单个或少数节点来获取用户的完整访问记录。

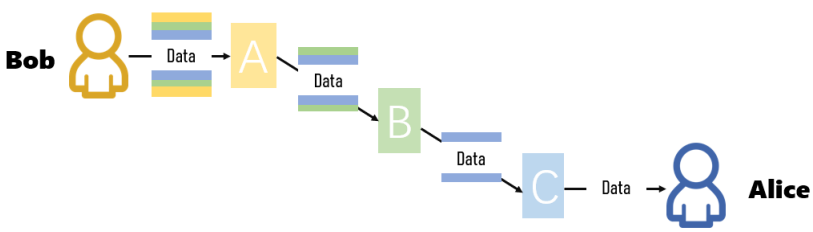

图1：Tor网络基本原理示意图

2.1.2 隐藏服务（Hidden Services）

除了连接外部服务器外，Tor还支持隐藏服务，即Onion Service。隐藏服务为用户提供HTTPS的所有安全性，并增加Tor浏览器的隐私优势。在本项目中，我们没有用到隐藏服务进行调试，因此相关原理不再详述。

2.2 Bento Box

Bento是一种新型体系结构，它允许网络中继充当可编程的“中间盒”。

利用Bento Box，client客户端可以用高级语言（在我们的实现中为Python）编写复杂的中间盒函数，在可用的Tor中继上存储和运行。例如，我们可以在网络上添加一定的覆盖流量（cover traffic，作为噪声与正常通信流量混合）实现短时间内部分用户匿名性大大增强（代价是增加带宽消耗）。

2.2.1 Bento Server

Bento Server与其所搭档的Tor中继在同一台机器上运行，但由于它的目标是不修改Tor，它将作为一个单独的进程在单独的端口监听，相当于Tor网络的扩展。如图2，原Tor中继通过修改出口节点的连接策略连接到Bento Server，然后由后者代替前者，实现客户端所发送的函数想要实现的功能。

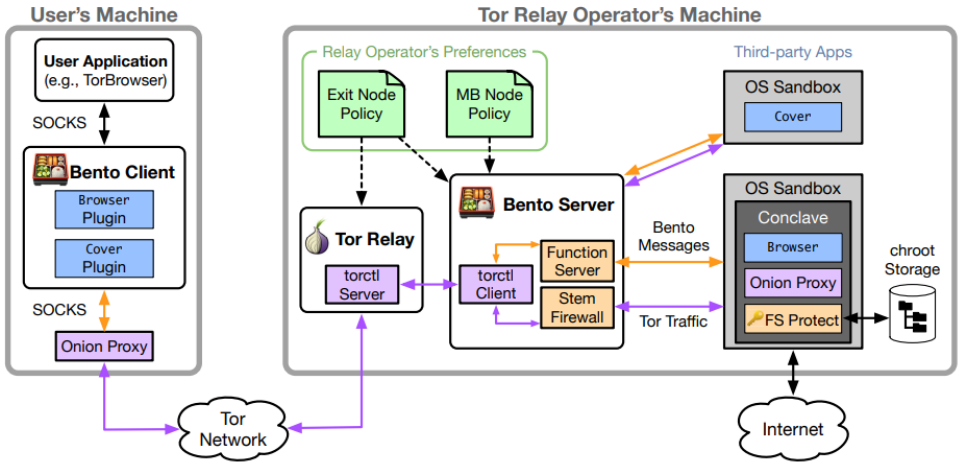

图2：Bento Server扩展Tor中继示意图（紫色部分表示Tor连接，黄色部分表示Bento连接）

要使Tor中继连接到Bento服务器，可以通过localhost连接，或直接让Bento作为一个隐藏服务（hidden service）运行。本项目测试采用第一种方法，即我们将在client端指定server的IP地址。

2.2.2 Bento Client

Bento Client将编写好的函数（如browser）经Tor代理匿名地发送到server端储存并运行，并接收server返回的结果（如图2所示）。

在本项目中，我们希望从hello_world开始复现原文中的几个函数——包括允许client在命令行向server发送函数并简单测试的interactive_client函数和让server端代理运行网页客户端的browser函数；然后尝试实现server端的流量监控函数monitor。

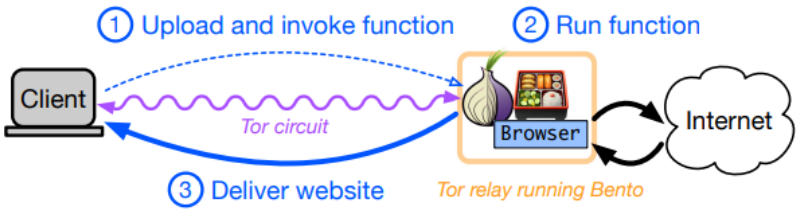

图3：利用bento扩展运行browser原理示意图

2.2.3 代码框架

Bento Box的代码框架如下：

```
bento
├─server
|   |  runserver.py
|   |  driver.py
|   |
|   ├─core
|   |      __init__.py
|   |      bentoapi.py
|   |      config.py
|   |      function.py
|   |      handler.py
|   |      session_mngr.py
|   |
|   └─graphene-sgx
|           ...
|
├─common
|      __init__.py
|      protocol.py
|      util.py
|
└─client
       __init__.py
       api.py
```

- server：包含bento server的核心模块和bento server的启动代码。
  - runserver.py：用于启动bento server。
  - driver.py：server在子进程中调用此程序来执行client上传的函数代码。
  - core：包含用于处理请求(handler.py)、会话管理(session_mngr.py)和函数存储(function.py)等的核心server模块。
  - graphene-sgx：包含与graphene-sgx飞地相关的代码和文件，本项目中尚未用到。
- common: 包含server和bento都要使用的通用模块。
  - protocol.py：定义了用于client请求、server响应以及会话管理等的应用协议。
  - util.py：定义了一些工具函数。
- client：包含client端用于与server端进行交互的API。
  - api.py：定义了ClientConnection类及其成员函数send_store_request等，后续主要使用该类即其成员函数来实现各个功能。

3 功能复现

3.1 hello_world

首先以hello_world为例，运行Bento程序的基本框架如下（完整代码文件见/testings/test_hello_world.py）：

- 首先将client端连接到指定的server端：

```
'''
test_hello_world.py
'''
conn = ClientConnection(args.host, args.port)
```

- 向server发送存储（store）请求，将function存储到server，返回函数的token：

```
'''
/testing/test_hello_world.py
'''
function_name ='hello'
function_code = util.read_file(f"functions/{function_name}")
token, errmsg= conn.send_store_request(function_name, function_code)
if errmsg is not None:
    util.fatal(f"Error message from server {errmsg}")

    logging.debug(f"Got token: {token}")
```

其中的function_code存储在外部文件中：

```
'''
/testing/functions/hello
'''
def hello():
    api.send("hello world")
```

- 发送执行（exec）请求，返回一个用以打开结果的session_id：

```
'''
/testing/test_hello_world.py
'''
call= f"{function_name}()"
session_id, errmsg= conn.send_execute_request(call, token)
if errmsg is not None:
    util.fatal(f"Error message from server {errmsg}")

    logging.debug(f"Got session_id: {session_id}")
```

- 发送打开（open）请求，打印结果：

```
'''
/testing/test_hello_world.py
'''
conn.send_open_request(session_id)
data, session_id, err= conn.get_sessionmsg()
print(data.decode())
```

执行结果如图4。

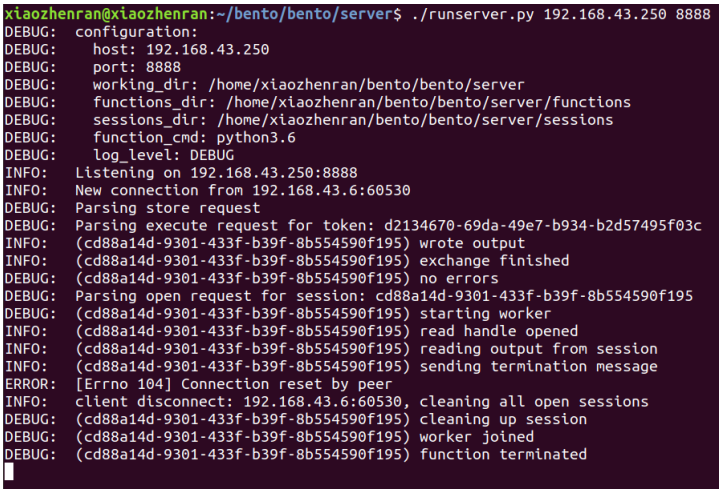

图4-1：test_hello_world（server端）

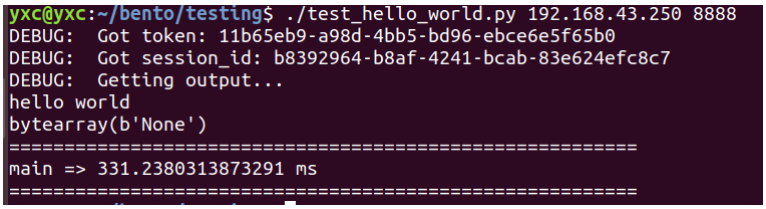

图4-2：test_hello_world（client端）

3.2 interactive_client

interactive_client 利用ClientConnection类及其成员函数实现对了store、exec、open、recv、send、exit六个指令的响应（完整代码见 /experiments/ interactive_client.py），让用户能够在命令行使用这些指令向Bento server发送请求，编写函数并对其进行简单测试。在这六个函数中，前四个起主要作用。下面分别从client和server两边的反应分析：

1. 建立连接

- server

首先利用socket包指定IP地址和端口号：

```
'''
/bento/server/runserver.py
'''
sock= socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.bind((opts.host, opts.port))
```

然后server开始对该端口进行监听，等待client与之相连并发送信息：

```
'''
/bento/server/runserver.py
'''
try:
    sock.listen()
    logging.info(f"Listening on {opts.host}:{opts.port}")
    while True:
        conn, addr= sock.accept()
        logging.info("New connection from %s:%d" % addr)
        new_thread= ClientThread(addr[0], addr[1], conn)
        new_thread.start()
```

连接成功建立后，server会根据传输过来的指令创建一个新的线程类对象ClientThread，然后start()开始运行线程。ClientThread的定义在/bento/server/core/handler.py中，可以在其handle_client函数的定义中看到其对不同指令做出分辨，并调用相应的函数执行。

- client

首先所有cilent都需要跟server建立TCP连接。在ClientConnection类中可直接调用socket包的函数。

```
'''
/experiments/interactive_client.py
'''
conn= ClientConnection(args.host, args.port)
```

```
'''
/bento/client/api.py
'''
class ClientConnection:
    """
    represents a connection with a Bento server in order to allow
sending/recving
    requests/responses and session messages
    """

    def __init__(self, address: str, port: int):
        self.conn= socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.conn.connect((address, port))
```

2. store指令

接下来首先分析store指令，其功能是将我们需要执行的函数传输到server端，并接受server的反馈信息（包括token），信息传输过程如下：

```
'''
/experiments/interactive_client.py
'''
token, errmsg= conn.send_store_request(function_name, "\n".join(function_code))
```

server收到store指令后，调用handle_store_request函数，将代码储存在当前的session中，并返回一个后续调用所需的token：

```
'''
/bento/server/core/handler.py
'''
def handle_store_request(self, request: StoreRequest):
    """
        generate a token, write the function info to disk, and then send the
        token back to the client
    """
    token= str(uuid.uuid4())
    function.create_function(token, request.name, request.code)
    self.__send_pkt(StoreResponse(token))
```

3. exec指令

- client

client 将token和具体要执行的函数发送过去，并接收返回的信息（包括结果储存的session_id）。若token错误，则server不会接受。代码如下：

```
'''
/experiments/interactive_client.py
'''
session_id, errmsg= conn.send_execute_request(call, token)
```

- server

收到exec指令后，server调用handle_execute_request函数，根据token找到需要执行的函数，创建线程并执行：

```
'''
/bento/server/core/handler.py
'''
function_data= function.get_function(request.token)
```

最后的执行在/bento/server/driver.py中，使用的是python自带的compile、exec、eval三种函数。

```
'''
/bento/server/driver.py
'''
def __execute(code, call):
    """
    Load the function's context and then execute it
    """
    context = dict(locals(), **globals())
    context['api']= bentoapi
    byte_code= compile(code, '<inline>', 'exec')
    exec(byte_code, context, context)
    return eval(call, context)
```

server根据返回的函数执行结果，将结果储存到一个新的session中，并将对应的session_id传回cilent，代码如下：

```
'''
/bento/server/core/handler.py
'''
if function_data is not None:
    exec_data= {"call": request.call, "code": function_data['code']}
    exec_data= base64.urlsafe_b64encode(json.dumps(exec_data).encode()).decode()
    new_session= session_mngr.create()
    new_session.execute(exec_data)
    self.__send_pkt(ExecuteResponse(new_session.session_id))
else:
    self.__send_pkt(ErrorResponse("invalid token", Types.Execute))
```

4. open指令

- client

  在client端，我们只需要将我们想要打开的session_id和open命令发送给server即可。

```
'''
/experiments/interactive_client.py
'''
conn.send_open_request(session_id)
```

- server

收到open指令后，server调用handle_open_request函数，根据传输过来的session_id读取数据。

```
'''
/bento/server/core/handler.py
'''
session= session_mngr.get(request.session_id)
```

然后将执行结果发送回client。

```
'''
/bento/server/core/handler.py
'''
if session is None:
    self.__send_pkt(ErrorResponse(f"no session exists with id:
{request.session_id}", Types.Open))
    return

if session in self.open_sessions:
    self.__send_pkt(ErrorResponse(f"session: {session.session_id} already open",
Types.Open))
    return
```

执行open指令后，server已经将结果发送出去，只是client此时尚未接收此结果。

5. recv指令

- client

client在recv时，不需要参数，也不需要对server传输任何指令。需要的结果在上一步open中已经发送了，只需要直接接收：

```
'''
/experiments/interactive_client.py
'''
data, session_id, err= conn.get_sessionmsg()
```

接收到结果后，根据情况打印data：

```
'''
/experiments/interactive_client.py
'''
if data is not None:
    if err:
        print(f"Error: {data}")
    elif conn.conn in select.select([conn.conn], [], [], 2)[0]:
        print(f"==recved message from session: {session_id}")
        data=data.decode()#这里需要将数据从 bytes 类转化回 string 类
        print(data)
    else:
        print("no data")
```

通过以上分析也可以看出，hello_world其实只是在代码文件中直接调用了对应的函数来执行function code，而不需要用户在命令行通过指令进行输入。后续的browser和monitor也是使用这些函数来实现功能。

在复现interactive_client的过程中，我们发现bento的源代码中存在一些错误，这些错误会导致程序无法正确执行。***这里我们对源码中的两个错误进行说明：***

- 问题1：只能录入整数行代码

  interactive_client.py的store的部分存在错误，修改如下：

  ```
  '''
  /experiments/interactive_client.py
  '''
  if selection == 'store':
      line= input()
      line_next= input()
      # 此处将源码中的 and 修改为 or，否则只能录入偶数行的代码
      while line or line_next:
          function_code.append(line)
          function_code.append(line_next)
          line= input()
          line_next= input()

          token, errmsg= conn.send_store_request(function_name,
  "\n".join(function_code))
  ```

- 问题2：数据类型转换错误

在interactive_client中发送简单的两个实数加法函数并执行，recv后报错信息如下：

```
>> recv
Error: bytearray(b'session terminated')
```

原因为 interactive_client.py 源码中 exec 部分调用 Bento API 的send函数时，其中的 len(data) 会自动将int数据转换为 str/bytes 类型。因此我们将 /bento/server/driver.py 中所有函数执行结果全部转换为str类型，如下：

```
'''
/bento/server/driver.py
'''
retval= __execute(code, call)
# 直接将所有输出都转为str类型
data=str(retval)
bentoapi.send(data)
```

发现此时可以store和exec可以正常执行，但由于 str 类型数据被写入 stdout 中时会自动转换为 bytes类型，因此 client 接收到的数据类型为 bytes 而不是str。因此需要在 interactive_client.py 的recv部分将 bytes 转换成str，如下：

```
'''
/experiments/interactive_client.py
'''
if data is not None:
    if err:
        print(f"Error: {data}")
    elif conn.conn in select.select([conn.conn], [], [], 2)[0]:
        print(f"==recved message from session: {session_id}")
        # 将 data 从 bytes 类转回str类
        data=data.decode()
        print(data)
    else:
        print("no data")
```

至此interactive_client可以正常运行，如图5：

```
xiaozhenran@xiaozhenran:~/bento/bento/server$ ./runserver.py 192.168.43.250 8888
DEBUG:  configuration:
DEBUG:    host: 192.168.43.250
DEBUG:    port: 8888
DEBUG:    working_dir: /home/xiaozhenran/bento/bento/server
DEBUG:    functions_dir: /home/xiaozhenran/bento/bento/server/functions
DEBUG:    sessions_dir: /home/xiaozhenran/bento/bento/server/sessions
DEBUG:    function_cmd: python3.6
DEBUG:    log_level: DEBUG
INFO:   Listening on 192.168.43.250:8888
INFO:   New connection from 192.168.43.6:52692
DEBUG:  Parsing store request
DEBUG:  Parsing execute request for token: 11b65eb9-a98d-4bb5-bd96-ebce6e5f65b0
INFO:   (b8392964-b8af-4241-bcab-83e624efc8c7) wrote output
INFO:   (b8392964-b8af-4241-bcab-83e624efc8c7) wrote output
INFO:   (b8392964-b8af-4241-bcab-83e624efc8c7) exchange finished
DEBUG:  (b8392964-b8af-4241-bcab-83e624efc8c7) no errors
DEBUG:  Parsing open request for session: b8392964-b8af-4241-bcab-83e624efc8c7
DEBUG:  (b8392964-b8af-4241-bcab-83e624efc8c7) starting worker
INFO:   (b8392964-b8af-4241-bcab-83e624efc8c7) read handle opened
INFO:   (b8392964-b8af-4241-bcab-83e624efc8c7) reading output from session
INFO:   (b8392964-b8af-4241-bcab-83e624efc8c7) reading output from session
INFO:   (b8392964-b8af-4241-bcab-83e624efc8c7) sending termination message
ERROR:  [Errno 104] Connection reset by peer
INFO:   client disconnect: 192.168.43.6:52692, cleaning all open sessions
DEBUG:  (b8392964-b8af-4241-bcab-83e624efc8c7) cleaning up session
DEBUG:  (b8392964-b8af-4241-bcab-83e624efc8c7) worker joined
DEBUG:  (b8392964-b8af-4241-bcab-83e624efc8c7) function terminated
```

图5-1：interactive_client（server端）

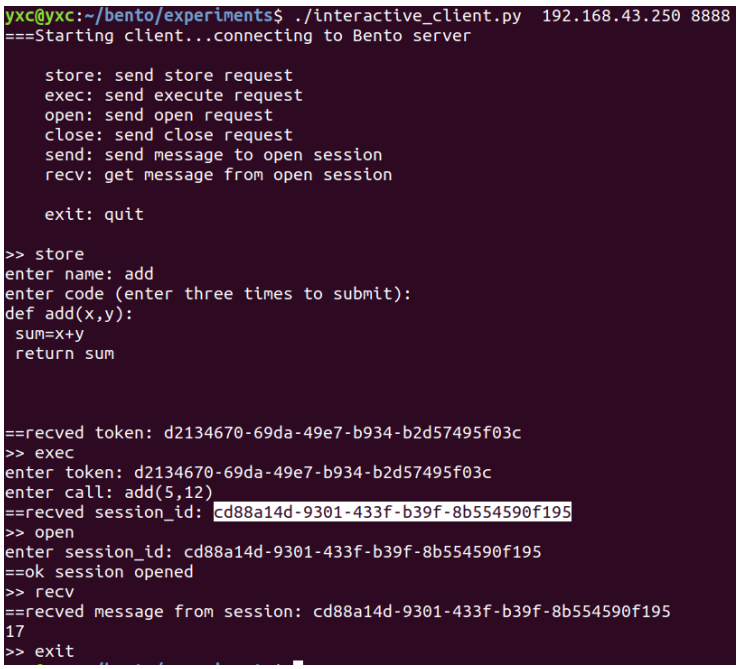

图5-2：interactive_client（client端）

3.3 browser浏览器功能

我们希望用Bento实现浏览器功能以达到更强的安全性。如果用户通过在一个由Tor连接到的单独的Bento Box上浏览网页，而其本身不运行网页客户端，攻击者就无法观察到可识别的行为。 Browser函数允许client向server发送网站URL，然后由server运行网页客户端。由于网页本身的大小也会暴露一定信息，client还可以指定需要填充多少字节送回，以更好地实现安全性。

此处浏览器的设计不能用于对延迟敏感的网页互动，例如在线游戏或视频聊天。

我们发送URL为http://www.baidu.com/，并要求一字节填充，测试结果如图6所示：

```
yxc@yxc:~/bento/bento/server$ ./runserver.py 192.168.43.6 8888
DEBUG:  configuration:
DEBUG:    host: 192.168.43.6
DEBUG:    port: 8888
DEBUG:    working_dir: /home/yxc/bento/bento/server
DEBUG:    functions_dir: /home/yxc/bento/bento/server/functions
DEBUG:    sessions_dir: /home/yxc/bento/bento/server/sessions
DEBUG:    function_cmd: python3.6
DEBUG:    log_level: DEBUG
INFO:   Listening on 192.168.43.6:8888
INFO:   New connection from 192.168.43.250:40730
DEBUG:  Parsing store request
DEBUG:  Parsing execute request for token: 160eec80-e5fe-4381-bd49-7296781e5c22
DEBUG:  Parsing open request for session: 3a7be6ac-43b0-43b1-b8b9-a1faa585df88
DEBUG:  (3a7be6ac-43b0-43b1-b8b9-a1faa585df88) starting worker
INFO:   (3a7be6ac-43b0-43b1-b8b9-a1faa585df88) read handle opened
INFO:   (3a7be6ac-43b0-43b1-b8b9-a1faa585df88) wrote output
INFO:   (3a7be6ac-43b0-43b1-b8b9-a1faa585df88) wrote output
INFO:   (3a7be6ac-43b0-43b1-b8b9-a1faa585df88) reading output from session
INFO:   (3a7be6ac-43b0-43b1-b8b9-a1faa585df88) reading output from session
INFO:   (3a7be6ac-43b0-43b1-b8b9-a1faa585df88) exchange finished
DEBUG:  (3a7be6ac-43b0-43b1-b8b9-a1faa585df88) no errors
INFO:   (3a7be6ac-43b0-43b1-b8b9-a1faa585df88) sending termination message
ERROR:  failed to recv header
INFO:   client disconnect: 192.168.43.250:40730, cleaning all open sessions
DEBUG:  (3a7be6ac-43b0-43b1-b8b9-a1faa585df88) cleaning up session
DEBUG:  (3a7be6ac-43b0-43b1-b8b9-a1faa585df88) worker joined
DEBUG:  (3a7be6ac-43b0-43b1-b8b9-a1faa585df88) function terminated
```

图6-1：browser（server端）

图6-2：browser（client端）

4 功能拓展: 监控（monitor）

在非匿名网络中，通常会随需求动态地扩大或缩小服务器的“备份”，将请求引导到到负载最低的服务器。这在Tor中却较难实现。对于Tor，PeerFlow[2]提出通过带宽加权投票为Tor中继站进行负载平衡，限制对手伪造带宽的能力。对于隐藏服务，OnionBalance[3]引入了预先创建副本的方法，并为每个副本发布不同的描述符到隐藏服务目录。默认的Tor客户端将从这些描述符中随机选择一个介绍点，从而将客户端均匀地分配到各个副本。然而这会使得它过分依赖可用引入点的最大数目，适应性和故障切换能力被严重限制。为了提高隐藏服务的可扩展性，Sucu引入了额外的负载平衡技术，需要对Tor的源代码进行修改。

这里我们希望利用Bento实现monitor功能server服务器的当前运行数据进行监控，并据此进行负载平衡调整。由于时间紧迫，且官方文章中给出的Load Balancer尚未在GitHub公开源码，我们仅完成了第一部分，即监控和抓取server的CPU核数、使用率、内存使用情况等信息。我们用psutil包实现这些功能，如下：

```
'''
/testing/test_monitor.py
'''
import psutil
import datetime
import time

def monitor():
    final="Bento服务器当前数据如下\n"
    # CPU的逻辑核数
    cpu_count = psutil.cpu_count()
    # cpu的使用率
    cup_per = psutil.cpu_percent(interval=0.5) # 0.5刷新频率
    final=final+f"cpu的逻辑核数为{cpu_count},cpu的平均使用率为{cup_per}\n"
    # 内存信息
    memory_info = psutil.virtual_memory()
    # 总内存
    memory_total = memory_info.total / 1024 / 1024
    # 内存使用率
    #memory_per = (memory_total - memory_info.available) / memory_total * 100
    memory_per = memory_info.percent
    final=final+f"总内存大小为{memory_total}M,内存的使用率为{memory_per}\n"
    # 硬盘信息
    disk_info = psutil.disk_usage("/") # 根目录磁盘信息
    # 根目录大小
    disk_total = disk_info.total
```

```
    # 根目录使用情况
    disk_per = float(disk_info.used / disk_total * 100 )
    final=final+f"根目录大小为{disk_total / 1024 / 1024}M,根目录使用率为
{round(disk_per,2)}\n"
    # 网络使用情况
    net = psutil.net_io_counters()
    # print(net)
    # 网卡配置信息
    net_ipy = psutil.net_if_addrs()
    #print(f"net_ipy {net_ipy}")
    net_ip = net_ipy['wlo1'][0][1]
    final=final+f"本机的IP地址为{net_ip}\n"
    # 收取数据
    net_recv = float( net.bytes_recv / 1024 /1024)
    # 发送数据
    net_sent = float(net.bytes_sent /1024 /1024)
    final=final+f"网络收取{round(net_recv,2)}M的数据,发送{round(net_sent,2)}M的数据
\n"
    # 获取当前系统时间
    current_time = datetime.datetime.now().strftime("%F %T") # %F年月日 %T时分秒
    final=final+f"当前时间是：{current_time}\n"
    time.sleep(1)
    api.send(final)
```

直接运行会发现抓取信息不完全，原因是python对中文字符的长度len读取为1，而转换为二进制字符后长度为2或3；解决方法为用encode函数直接读取字符的二进制长度。

最终测试结果如图7所示。

```
xiaozhenran@xiaozhenran:~/bento/bento/server$ ./runserver.py 192.168.43.250 8888
DEBUG:  configuration:
DEBUG:    host: 192.168.43.250
DEBUG:    port: 8888
DEBUG:    working_dir: /home/xiaozhenran/bento/bento/server
DEBUG:    functions_dir: /home/xiaozhenran/bento/bento/server/functions
DEBUG:    sessions_dir: /home/xiaozhenran/bento/bento/server/sessions
DEBUG:    function_cmd: python3.6
DEBUG:    log_level: DEBUG
INFO:   Listening on 192.168.43.250:8888
INFO:   New connection from 192.168.43.6:59108
DEBUG:  Parsing store request
DEBUG:  Parsing execute request for token: 977587ea-87dd-4ce6-9e2e-121112814729
DEBUG:  Parsing open request for session: 5b20ee6a-f98a-4137-9b53-1ac6644ac609
DEBUG:  (5b20ee6a-f98a-4137-9b53-1ac6644ac609) starting worker
INFO:   (5b20ee6a-f98a-4137-9b53-1ac6644ac609) read handle opened
INFO:   (5b20ee6a-f98a-4137-9b53-1ac6644ac609) wrote output
INFO:   (5b20ee6a-f98a-4137-9b53-1ac6644ac609) wrote output
INFO:   (5b20ee6a-f98a-4137-9b53-1ac6644ac609) exchange finished
DEBUG:  (5b20ee6a-f98a-4137-9b53-1ac6644ac609) no errors
INFO:   (5b20ee6a-f98a-4137-9b53-1ac6644ac609) reading output from session
INFO:   (5b20ee6a-f98a-4137-9b53-1ac6644ac609) reading output from session
INFO:   (5b20ee6a-f98a-4137-9b53-1ac6644ac609) sending termination message
ERROR:  [Errno 104] Connection reset by peer
INFO:   client disconnect: 192.168.43.6:59108, cleaning all open sessions
DEBUG:  (5b20ee6a-f98a-4137-9b53-1ac6644ac609) cleaning up session
DEBUG:  (5b20ee6a-f98a-4137-9b53-1ac6644ac609) worker joined
DEBUG:  (5b20ee6a-f98a-4137-9b53-1ac6644ac609) function terminated
```

图7-1：test_monitor（server端）

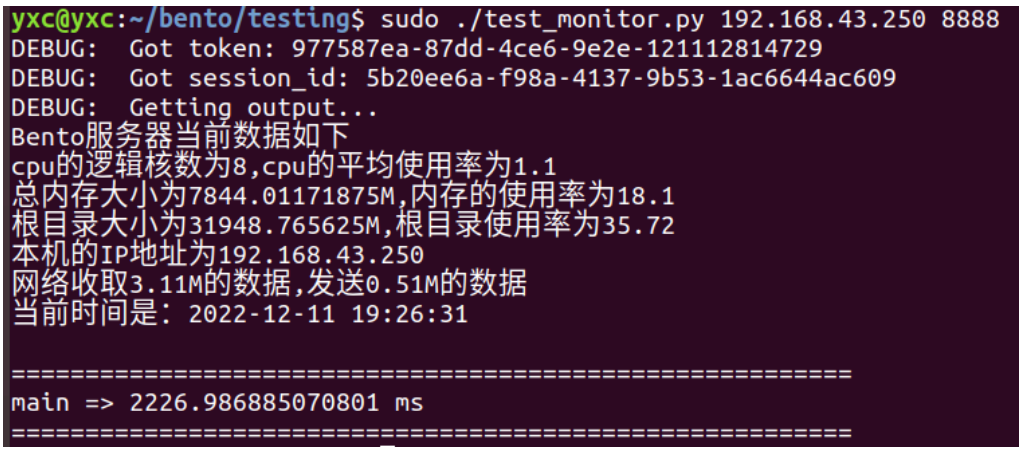

图7-2：test_monitor（client端）

5 总结与展望

本文基于Bento对Tor网络的功能拓展作了一些探索。Tor网络上难以部署额外的功能，因此我们利用Bento Box在Tor网络的最后一个中继上作扩展，首先复现了原文章中提到的几个函数，包括interactive_client和browser，并修正了源码中的几处错误；然后实现了简单的对server服务器端的监控函数。由于时间紧迫，且原项目的Load Balancer源码也尚未公开，我们仅实现了监控部分，如果未来有机会时间充足，可以继续尝试完成负载平衡的部分。

本次项目前期花费较多时间修正源码中的几处错误，另外在Tor的安装和连接上也花费了较多时间。因此除项目内容本身之外，我们对Tor网络连接的原理也有了更多了解，如为Tor配置网桥等问题；也在配置Tor的障碍中深刻地感受到了互联网安全的重要性。

Bento证明了可编程匿名网络的实现是可能的，对改善匿名性（如在server端代理运行浏览器以更好抵御指纹攻击）、改善隐藏服务的性能、实现匿名网络上的下载管理（如原文中提出的DropBox、Shard）等可能发挥重要作用。同时Bento结构本身也可以进一步改进，如对可信执行环境（trusted execution environment，TEE，在Bento的实现中为Intel SGX）的依赖使得其存在一定安全漏洞。我们希望未来能够有机会对可编程匿名网络作进一步探索。

参考文献

1. Reininger, Michael, et al. "Bento: Safely bringing network function virtualization to Tor." *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 2021.
2. Johnson, Aaron, et al. "PeerFlow: Secure Load Balancing in Tor." *Proc. Priv. Enhancing Technol.* 2017.2 (2017): 74-94.
3. onionbalance [n. d.]. OnionBalance. https://onionbalance.readthedocs.io/en/latest/. ([n. d.]).