


A TCP based Real Framework

- Three roles and one main function
 - File Scanner
 - TCP Listener
 - File Downloader

3 processes
- A main function to start each process and finally say hello to peers

A TCP based Real Framework

- File Scanner: (an infinity loop)
 1. Traverse the “share” folder
 2. Compare with the existing file dictionary using file size and mtime
 3. If anything changes, send “news” to peers

A TCP based Real Framework

- TCP Listener: (an infinity loop)

1. Listen a port
2. If any new connection comes, create a new thread and pass this connection to the new thread (call it sub connection)

Sub connection:

3 types of messages will be checked: “news”, “news” and “get file” requests:

1. “**news**”: checkout new files from the new file list of peer. If any news, make **tickets** for downloaders
2. “**hello**”: checkout new files from the entire file list of peer. Send our entire file list back. If any news, make **tickets** for downloader.
3. “**get_file**”: send the requested file blocks using a long alive connection or short connection.

A TCP based Real Framework

- File Downloader: : (an infinity loop)
 1. Checkout **tickets** of new files
 2. Download the earliest **ticket**
 - I. Check the size and mtime using file dictionary, if existed, skip
 - II. For the first block, create a new file (named xxx.yyy.lefting) with the same size (filling 0)
 - III. Get blocks one by one, if any error, stop, delete and download the next file. If the block is received well, record on the **ticket**
 - IV. When the file is received completely, delete the ticket, change the filename.
 3. Update the file dictionary

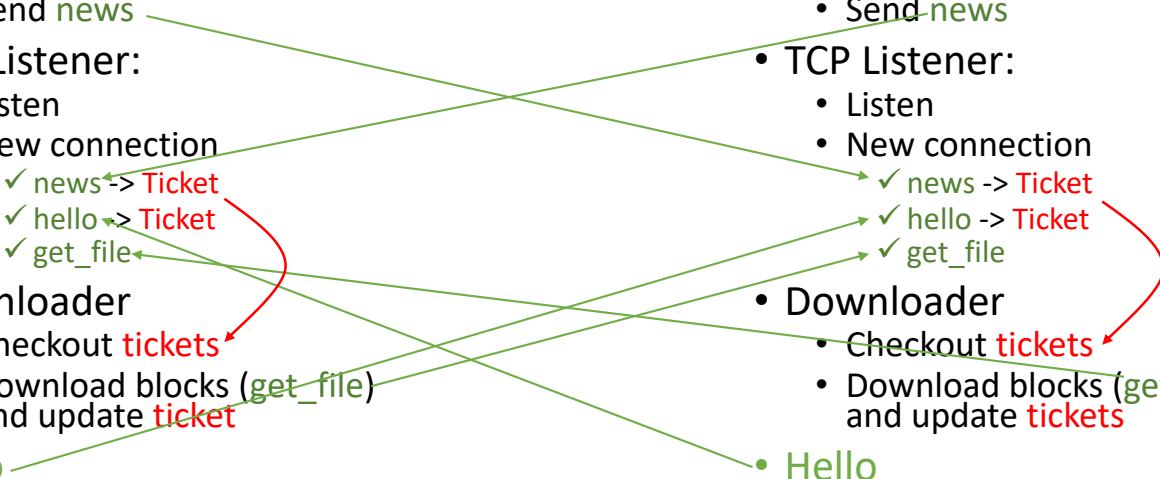
Overall view

APP 1

- File Scanner
 - Traverse
 - Send **news**
- TCP Listener:
 - Listen
 - New connection
 - ✓ **news** ↔ **Ticket**
 - ✓ **hello** ↔ **Ticket**
 - ✓ **get_file** ←
- Downloader
 - Checkout **tickets**
 - Download blocks (**get_file**) and update **ticket**
- **Hello**

APP 2

- File Scanner
 - Traverse
 - Send **news**
- TCP Listener:
 - Listen
 - New connection
 - ✓ **news** → **Ticket**
 - ✓ **hello** → **Ticket**
 - ✓ **get_file**
- Downloader
 - Checkout **tickets**
 - Download blocks (**get_file**) and update **tickets**
- **Hello**



Difficulties

- Share variables between processes and threads

```
import multiprocessing as mp

from multiprocessing import Process
from threading import Thread

g_peers = mp.Manager().list([])

g_file_dict = mp.Manager().dict({})


# A process for scanning all the files

file_d_process = Process/Thread(target=file_scanner, args=(g_file_dict, g_peers,))
file_d_process.daemon = True
file_d_process.start()
```

Difficulties

- My protocol:

- 4 bytes: json header size
- 4 bytes: binary data size
- json header (binary)
- binary data (if need)



```
{  
  "news":...  
  "hello":...  
  "get_file":...  
}
```

Difficulties

- TCP File transfer – so called “sticky package problem”
 - Implement TCP correctly! STREAM!! Not Package!

File Sender:

`conn_socket.send(packaged_data)`

File Receiver:

`conn_socket.send(packaged_request)`

`conn_socket.recv(8) # => json_header_size,
binary data size`

`buf = b''`

`while received_size < json_header_size:
 buf += conn_socket.recv(json_header_size)`

`json_bin = buf[:json_header_size]
buf = buf[json_header_size:]`

`while received_size < json_header_size:
 buf += conn_socket.recv(json_header_size)
data_bin = buf`

- My protocol:

- 4 bytes: json header size
- 4 bytes: binary data size
- json header (binary)
- binary data (if need)

Difficulties

- Folder!
 - Try to use a long path for each file to avoid to record the information of folder
 - Eg. share/123/456.txt is just a FILE. When the receiver know the file name, the structure of the folder is known.