Introduction to Networks

lyakhovs@oregonstate.edu

CS 372/ECE 372, April 27, 2023

Lab 1: Feedback

- How do we feel about Python?

Lab 1: Feedback

- How do we feel about Python?
- Was the lab at an appropriate difficulty?

Lab 1: Feedback

- How do we feel about Python?
- Was the lab at an appropriate difficulty?
- Was the lab fun? Tedious? Boring?

Lab 1: Feedback

- How do we feel about Python?
- Was the lab at an appropriate difficulty?
- Was the lab fun? Tedious? Boring?
- Help me create future labs! What are you interested in?

Welcome to Lab 2: Threads and Async

- Learn to use python threads and *asyncio* module
- Implement a simple version of FTP
- Learn a little more wireshark

Threads and *asyncio*

Why do we care?

- Server from lab 1 serves one client at a time: the rest have to wait
- How do we serve clients simultaneously?
- Concurrent programming!

Concurrency is not Parallelism

- A talk by Rob Pike: `https://www.youtube.com/watch?v=oV9rvDllKEg`
- Doing things concurrently doesn't mean doing them at the same time
- Example: while waiting for Task A to finish do Task B

Concurrency is not Parallelism: TA Office Hours

- How do I help people in office hours?

Concurrency is not Parallelism: TA Office Hours

- How do I help people in office hours?
  1. Go from student to student

Concurrency is not Parallelism: TA Office Hours

- How do I help people in office hours?
  1. Go from student to student
  2. Give advice to a student

Concurrency is not Parallelism: TA Office Hours

- How do I help people in office hours?
  1. Go from student to student
  2. Give advice to a student
  3. Move on to the next student while they're trying it out

Concurrency is not Parallelism: TA Office Hours

- How do I help people in office hours?
  1. Go from student to student
  2. Give advice to a student
  3. Move on to the next student while they're trying it out
  4. ...answer the next question when I come back around!

Concurrency is not Parallelism: TA Office Hours

- How do I help people in office hours?
  1. Go from student to student
  2. Give advice to a student
  3. Move on to the next student while they're trying it out
  4. ...answer the next question when I come back around!
- I'm not helping students in *parallel* but we are doing office hours *concurrently*!

Concurrency is not Parallelism: Sending Data

- Let's say we want to send 2 files from Host A to Host B
  - No concurrency:

Concurrency is not Parallelism: Sending Data

- Let's say we want to send 2 files from Host A to Host B
  - No concurrency: $2(L/R + d_{prop})$

Concurrency is not Parallelism: Sending Data

- Let's say we want to send 2 files from Host A to Host B
  - No concurrency: $2(L/R + d_{prop})$
  - Parallelism (e.g. 2 links):

Concurrency is not Parallelism: Sending Data

- Let's say we want to send 2 files from Host A to Host B
  - No concurrency: $2(L/R + d_{prop})$
  - Parallelism (e.g. 2 links): $L/R + d_{prop}$

Concurrency is not Parallelism: Sending Data

- Let's say we want to send 2 files from Host A to Host B
  - No concurrency: $2(L/R + d_{prop})$
  - Parallelism (e.g. 2 links): $L/R + d_{prop}$
  - Concurrency:

Concurrency is not Parallelism: Sending Data

- Let's say we want to send 2 files from Host A to Host B
  - No concurrency: $2(L/R + d_{prop})$
  - Parallelism (e.g. 2 links): $L/R + d_{prop}$
  - Concurrency: $2(L/R) + d_{prop}$

Concurrency is not Parallelism: Sending Data

- Let's say we want to send 2 files from Host A to Host B
  - No concurrency: $2(L/R + d_{prop})$
  - Parallelism (e.g. 2 links): $L/R + d_{prop}$
  - Concurrency: $2(L/R) + d_{prop}$
- Even with a single link, we can achieve faster speeds with concurrency

Concurrency is not Parallelism: Sending Data

- Let's say we want to send 2 files from Host A to Host B
  - No concurrency: $2(L/R + d_{prop})$
  - Parallelism (e.g. 2 links): $L/R + d_{prop}$
  - Concurrency: $2(L/R) + d_{prop}$
- Even with a single link, we can achieve faster speeds with concurrency

- Rule of thumb: If we're doing computation we need parallelism.
- If we're doing IO (e.g. communicating over the network) we're looking for concurrency

Python Threads

- Threads In python **do not** execute in parallel!
- The operating system decides which thread should be running at what time
- The OS is fairly smart and can make good decisions
- **Note:** if you want parallel computation see *multiprocessing* module

Python Threads Library

- Use the builtin *threading* module (link in references)

```
1   from threading import Thread
2
3   def print_two(a, b):
4     print(a, ”and”, b)
5
6   # Initialize thread
7   thread = Thread(target=print_two, args=[2, 3])
8
9   # Start thread
10  thread.start()
11
12  # Wait for thread to finish
13  thread.join()
14
15  # Output: 2 and 3
```

Python Threads: Demonstration

Python's *asyncio*

- Instead of the operating system deciding when to switch context, the program does
- Creates a lighter version of threads: doesn't rely on complex schedulers
- A little more to it than threads (have to learn more!)

Python's *asyncio*: Concept

- "async/await" paradigm: awaiting *yields* execution until it is done waiting
- Can make regular functions *async* functions (called *coroutines*)
- To get output of coroutines we have to *await* them
- Only coroutines can await other coroutines
- **Note:** coroutines can still call regular (synchronous) functions

Python *asyncio* Library

- Use the builtin *asyncio* module (link in references)

```
1   import asyncio
2
3   async def print_me(i):
4       await asyncio.sleep(1)
5       print("Hello there", i)
6
7
8   async def main():
9       tasks = []
10      for i in range(100):
11          tasks.append(print_me(i))
12
13      await asyncio.gather(*tasks)
14
15  if __name__ == "__main__":
16      asyncio.run(main())
```

Python *asyncio*: Server

```
1   import asyncio
2
3   # Function that will run each time a client connects
4   async def handle_client(reader, writer):
5
6     # Send message to client
7     writer.write(b"Hello World!")
8     await writer.drain()
9
10    # Each side closes their own writer
11    writer.close()
12    await writer.wait_closed()
13
14  async def main():
15    # Initialize server
16    server = await asyncio.start_server(
17          handle_client,
18          INTERFACE, SPORT
19    )
20    # Start the server
21    async with server:
22      await server.serve_forever()
```

Python *asyncio*: Client

```
1   import asyncio
2
3   async def main():
4     # Connect to the server
5     reader, writer = await asyncio.open_connection(IP, DPORT)
6
7     # Read the message
8     message = await reader.readexactly(12)
9     print(message)
10
11    # Each side closes their own writer
12    writer.close()
13    await writer.wait_closed()
```

asyncio: Reader/Writer

Question: How to use the reader/writer? Answer: See official documentation!

- StreamReader: `https://docs.python.org/3/library/asyncio-stream.html#asyncio.StreamReader`
- StreamWriter `https://docs.python.org/3/library/asyncio-stream.html#asyncio.StreamWriter`

Python *asyncio*: Demonstration

Python's *asyncio*: Results

- We can make the network requests happen *concurrently*
- In effect we minimized the initial queueing delay
- Is it possible to overdo it?

Monitoring congestion with Wireshark

Lab 2: Concurrent Networking

Lab 2 Tasks

1. Implement server from assignment 1 with threading and test it with threaded client!
2. Use wireshark to plot graph showing network activity and congestion
3. Implement a simple file transfer client/server using asyncio

References

- Python Threads
  `https://docs.python.org/3/library/threading.html#threading.Thread`
- General Asyncio
  `https://docs.python.org/3/library/asyncio.html`
- Asyncio Networking
  `https://docs.python.org/3/library/asyncio-stream.html#asyncio-streams`