Socket Debugging, Server Design and Special Functions

Unit 14 - Hands-On Networking - 2018

Prof. Dr.-Ing. Thorsten Herfet, Andreas Schmidt, Pablo Gil Pereira

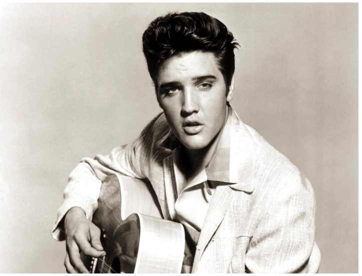
Telecommunications Lab, Saarland Informatics Campus, 23rd Feb. 2018

Socket Debugging Tools

- **Validate binding socket works**: `netstat -tulpn`
  - `-t` to show TCP sockets.
  - `-u` to show UDP sockets.
  - `-n` to show numeric IP addresses.
  - `-l` to show listening sockets.
  - `-p` to also show the process.
- **Validate TCP server working**: `netcat <server> <port>`
  - Go ahead typing your data. Enter to send.
- **Validate UDP server working**: `netcat -u <server> <port>`
  - Go ahead typing your data. Enter to send.

Return To Sender

Sometimes you might contact a server that is not listening.



Reactions

TCP: (RST=1).

UDP: ICMP

.

Servers

(The Good, The Bad and The Ugly)

UDP Time Service

```
# Server

from socket import *
import datetime

s = socket(AF_INET, SOCK_DGRAM)
s.bind(("0.0.0.0", 9000))
req, sender = s.recvfrom(1000)
print("Sender", sender, "asked for:",
      req.decode("utf-8"))
now = datetime.datetime.now()
print("Replying with:", now)
s.sendto(now.isoformat().encode("utf-8"), sender)
s.close()
```

```
# Client

from socket import *

s = socket(AF_INET, SOCK_DGRAM)
cmd = "time"
c = s.sendto(cmd.encode("utf-8"),
             ('127.0.0.1', 9000))
print("Requested",cmd,"which is",c,"bytes long.")
time, sender = s.recvfrom(100)
print("Sender",sender,"replied with time:",
      time.decode("utf-8"))
s.close()
```

What is the problem with this server implementation?

Single-use, disposable. Restart server after each served response.

UDP Time Service - Iterative 🔁

```python
# Server

from socket import *
import datetime

s = socket(AF_INET, SOCK_DGRAM)
s.bind(("0.0.0.0", 9000))

try:
    while True:
        req, sender = s.recvfrom(1000)
        print("Sender", sender, "asked for:",
              req.decode("utf-8"))
        now = datetime.datetime.now()
        print("Replying with:", now)
        s.sendto(now.isoformat().encode("utf-8"),
                 sender)
except KeyboardInterrupt:
    pass
finally:
    s.close()
```

```python
# Client

from socket import *

s = socket(AF_INET, SOCK_DGRAM)
cmd = "time"
c = s.sendto(cmd.encode("utf-8"),
             ('127.0.0.1', 9000))
print("Requested",cmd,"which is",c,"bytes long.")
time, sender = s.recvfrom(100)
print("Sender",sender,"replied with time:",
      time.decode("utf-8"))
s.close()
```

TCP Fibonacci Service - Iterative 🔁

```
# Server

from socket import *

def fib(n):
    if n == 0: return 0
    elif n == 1: return 1
    else: return fib(n-1)+fib(n-2)

def handle_request(client):
    buf = ""
    while True:
        d = client.recv(1).decode("utf-8")
        if d == "\n" or d == "\0":
            break
        buf += d
    number = int(buf)
    fibNumber = fib(number)
    client.send(str(fibNumber).encode("utf-8"))
    client.close()

if __name__ == "__main__":
    s = socket(AF_INET, SOCK_STREAM)
    s.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    s.bind(("0.0.0.0", 9000))
    s.listen(1)
    try:
        while True:
            client_socket, peer = s.accept()
            handle_request(client_socket)
    except KeyboardInterrupt:
        pass
    finally:
        s.close()
```

```
# Client

from socket import *
import datetime

start = datetime.datetime.now()
n = 37
s = socket(AF_INET, SOCK_STREAM)
s.connect(("localhost", 9000))
s.send(f"{n}\n".encode("utf-8"))
r = int(s.recv(10))
stop = datetime.datetime.now()
d = (stop - start).total_seconds()
print(f"fib({n})={r} ({d}s calc time)")
s.close()
```

TCP Fibonacci Service - Forked

```python
# Server

from socket import *
import os

s = socket(AF_INET, SOCK_STREAM)
s.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
s.bind(("0.0.0.0", 9000))
s.listen(5)

# Avoid zombies
signal.signal(signal.SIGCHLD, signal.SIG_IGN)

try:
    while True:
        client_socket, peer = s.accept()
        child_pid = os.fork()

        # Handle fork
        if child_pid == 0:
            handle_request(client_socket)
            break
        else:
            client_socket.close()
except KeyboardInterrupt:
    pass
finally:
    s.close()
```

Changes

- Fork after accept.
- `fork`: Creates a child process as copy of current process.
- Now: Two processes at the same line of code.
- Returns: 0 (inside child process), non-zero (parent process).

Details

- Handle `SIGCHLD`, otherwise we get zombie processes.
- Close `child_socket` in parent, otherwise it will never be released.

TCP Fibonacci Service - Threaded ⇄

```python
from socket import *
from threading import Thread

s = socket(AF_INET, SOCK_STREAM)
s.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
s.bind(("0.0.0.0", 9000))
s.listen(5)
try:
    while True:
        client_socket, peer = s.accept()
        t = Thread(target=handle_request,
                   args=(client_socket,))
        t.start()
except KeyboardInterrupt:
    pass
finally:
    s.close()
```

Changes

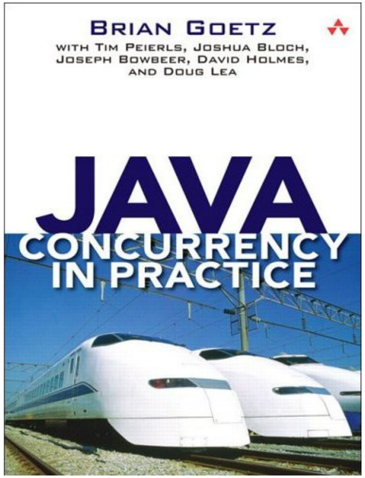
- Instead of `fork`, create a `thread` and give it the request method to run and `client_socket` to process.
- More lightweight and faster than `fork`, because only a second execution thread is generated. Memory is shared (leading to the usual problems)!

Concurrency and Parallelism - Advice

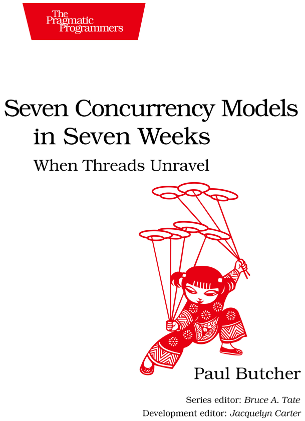
⚠ Not a central topic of this course, but **very** important for network engineers.

- Consider "Nebenläufige Programmierung" (basic Computer Science lecture, offered in summer semesters by Prof. Dr.-Ing. Holger Hermanns).
- Read a good book on concurrency.

Java Concurrency in Practice


Brian Goetz
with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea
JAVA
CONCURRENCY IN PRACTICE

Seven Concurrency Models in Seven Weeks: When Threads Unravel


The Pragmatic Programmers
Seven Concurrency Models in Seven Weeks
When Threads Unravel
Paul Butcher
Series editor: Bruce A. Tate
Development editor: Jacquelyn Carter

Block or Not

(Non-)Blocking I/O

Problem

- Some networking operations are blocking:
  - Examples: `send`, `recv`, `accept`, `connect`.
  - Program waits until operation succeeded.
  - Might wait forever if sender/receiver is not responsive.
- Example:
  - Client connects to server.
  - Server starts receiving, but client never sends.
  - Server waits forever.

Solutions

A) Use blocking I/O. Use Multithreading.
Spawn threads or child processes (see before).

B) Use blocking I/O. Specify Timeout.
Try to send/recv for X seconds, abort afterwards.

C) Use non-blocking I/O.
Call to blocking operation immediately returns an error if operation would have blocked; retry later.

D) I/O multiplexing.
Check which socket "is ready" to do certain actions.

Not to block (for too long)

Timeouts

- Timeouts specified on per-socket basis.
- `s.settimeout(timeout)`.
- `timeout`: timeout in seconds (float).

```
import socket

s = socket.socket(socket.AF_INET,
                  socket.SOCK_STREAM, 0)

s.settimeout(0.5)

try:
    s.connect(("127.0.0.1", 4567)) # try connect
    s.recv(1024)                   # try receive
except socket.timeout:
    print("Failed to connect/receive: TIMEOUT!")
s.close()
```

Non-blocking operation

- Tell the OS to immediately return, even with no data (and hence error).
  - `s.setblocking(flag)`
  - `flag`: enable (flag=1) or disable (flag=0) blocking
- If successful, return as usual.
- But if socket is "not ready", throw exception:
  - Program has to retry later (`EAGAIN`, `EWOULDBLOCK`).
  - Unclean design: keep polling for non-blocking operation.

Non-Blocking: Example

```python
import socket
import errno
import time

s = socket.socket(socket.AF_INET,
                  socket.SOCK_STREAM,
                  0)

print("connecting!")
s.connect(("smtp.uni-saarland.de", 25))
print("connected!")
s.setblocking(0)

while True:
    try:
        print("received: {}".format(
              s.recv(1024).decode('utf-8')))
        break
    except socket.error as e:
        err = e.args[0]
        if err == errno.EAGAIN:
            print("EAGAIN")
        if err == errno.EWOULDBLOCK:
            print("EWOULDBLOCK")
        time.sleep(0.01)    # avoid busy loop
```

```
$ python nonblocking.py
connecting!
connected!
EAGAIN
EWOULDBLOCK
EAGAIN
EWOULDBLOCK
EAGAIN
EWOULDBLOCK
EAGAIN
EWOULDBLOCK
EAGAIN
EWOULDBLOCK
received: 220 triton.rz.uni-saarland.de ESMTP
Sendmail; Fri, 2 Jun 2017 17:17:35 +0200
```

I/O Multiplexing I

- Imagine you run a server:
    - Server handles hundreds of client connections.
    - Any client could send data at any time.
    - But you do not know which one, so you have to poll.
- Inefficient example:

```
for s in sockets:  # iterate over clients
    try:
        # try to receive from this client
        data = s.recv()
    except socket.timeout:
        # nothing to receive: next client
        continue
```

- Query OS which sockets are "ready". OS returns sockets that...
    - ... can be read from.
    - ... can be sent to.
    - ... have an exception.
- Command: `select(rdrs, wrtrs, errs, tmout)`
    - `rdrs`: list of sockets to check if they are readable.
    - `wrtrs`: list of sockets to check if they are writeable.
    - `errs`: list of sockets to check if they have an I/O error.
    - `tmout`: timeout in seconds for `select()` to return.
    - Note: Select also accepts other I/O objects than sockets.

I/O Multiplexing II

Sockets are readable if:

- There is at least one byte to read.
- The other end has terminated the connection (EOF).
- The socket is listening and `accept()` would not block.
- A socket error is pending.

Sockets are writeable if:

- Kernel socket buffers still fit at least one byte.
- The write half of the connection is closed(!).
- Non-blocking connect has completed.
- A socket error is pending.

```
while True:
    rds = wrs = errs = None

    try:
        (rds, wrs, errs) = select.select(sockets,
                                         [],
                                         sockets,
                                         0.2)
    except socket.timeout:
        continue   # no socket ready or disrupted

    for s in rds:        # read available sockets
        data = s.recv()  # recv will *not* block

    for s in errs: # close sockets with errors
        s.close()
```

More on Sockets

Options

Manipulate socket behaviour (via `socket.setsockopt()`).

Socket Option	Description
SO_KEEPALIVE	Probe "silent" TCP connections if peer still reactive
SO_LINGER	Change socket close behaviour (flush/wait)
SO_RCVBUF	Size of receive buffer (default 4kB or more)
SO_SNDBUF	Size of send buffer (default 4kB or more)
SO_RCVLOWAT	Minimum number of bytes to recv (default = 1)
SO_SNDLOWAT	Minimum number of bytes to send (default = 1)
SO_REUSEADDR	Bind to port even if established connections exist to that port (e.g. quick restart of a server)
IP_TTL	Specify initial TTL of an IP packet
IP_RECVIF	Return on which interface a segment was received

Retrieve Hostnames and Service Ports

Hostnames:

- Use `socket.gethostbyname(hostname)` to translate the hostname to an IPv4 address.
  Example: `gethostbyname('www.nt.uni-saarland.de')` -> 134.96.7.186

- DNS and contents of `/etc/hosts` are consumed by this:

```
# /etc/hosts
# The following lines are desirable for IPv4 capable hosts
127.0.0.1       localhost

# 127.0.1.1 is often used for the FQDN of the machine
127.0.1.1       thishost.mydomain.org  thishost
192.168.1.10    foo.mydomain.org       foo
192.168.1.13    bar.mydomain.org       bar
```

Services:

- Use `socket.getservbyname(servicename[, protocolname])` to retrieve a protocol to a well-known port (for tcp or udp as `protocolname`).
  Example: `getservbyname('ssh')` -> 22

- Mapping can be found (and edited) in `/etc/services`.

Retrieve Address Information

⚠ **Problem:** Previously mentioned functions only work with IPv4 and require multiple calls to solve one task.

- New call combines all in one command:
  `socket.getaddrinfo(host, port, family=0, type=0, proto=0, flags=0)`
  - `host`: Domain Name, String representation of IPv4/v6 or `None`.
  - `port`: String service name (e.g. `http`).
  - `family`, `proto`, `flags` can be used to narrow results.
- Returns: `(family, type, proto, canonname, sockaddr)`
- `sockaddr` can be directly passed to `socket.connect()`.

IP Protocols

- The file `/etc/protocols` lists the protocols (and their numbers) that are payloads of the IP layer.
    - e.g. TCP(6), UDP(17), ICMP(1), …
- The number specified there is put into the field in the IPv4 header and the field in IPv6.
- Python's `socket` namespace includes `IPPROTO_*` constants that map to the same values.

Wrap-Up

Take-Home Messages

- Server design impacts performance. Chose your primitives well.
- Blocking calls have to be handeled to avoid stalls.
- Functions `gethostbyname()` etc. allow you to utilize other protocols (e.g. DNS).

Action Points

- **Download** and **solve** the task sheet.