Networking: Multiple Clients

Instructor: Linyi Li
Slides adapted from Dr. B. Fraser

Topics

- How can one program handle (very?) many requests?
  - Specifically a server handle many TCP clients?

TCP Server Recap

- Recall that on a TCP server:
  - We open the first socket and call accept()
  - accept() will return
  - .. a new socket file descriptor for the new client connection.
- How can we make our server work with multiple client sockets?

Idea 1: Thread per Connection

- Idea 1:
  - .. Server creates a new thread (or child process) for each accepted connection.
  - This thread handles the new client's socket.
- Pros:
  - Handle multiple clients cleanly.
- Cons:
  - .. Higher overhead of creating new processes or threads.

Idea 2: Non-Blocking Sockets

- Non-blocking accept() will either:

a) accept a new connection immediately or

b) or return immediately if no incoming connection.

– Also use non-blocking read() and write()

- Idea 2:

– ..Create array of open sockets and poll with non-blocking calls

– General Idea:

Server will infinitely loop through calling:

- non-blocking-accept to add any new socket to array
- non-blocking-read or non-bloccking-write (or both) on each socket in array as needed

– Pros: Avoids creating new processes/threads

– Cons: .. Busy-wait loop checking sockets.

Idea 3: Kernel Notify on Socket Event

- Idea 3:
  - .. Kernel notifies program on socket event.
  - –Use non-blocking sockets and kernel notifies program on socket events.
- .. I/O Multiplexing:
  - –Use syscalls to monitor multiple file descriptors.
  - –Program is notified when
    - .. a monitored file descriptor is ready for read or write (or on error).
  - –Use: select(), poll(), and epoll()

Idea 3: (cont)

- Generally speaking, this is how I/O multiplexing works:
  - We add file descriptors to the monitored list.
  - We indicate what events we want to monitor the file descriptors for, e.g., read and write.
  - We call the blocking function to wait for an event, e.g., select() or epoll()
  - When it returns, check which file descriptors can perform I/O.
  - We perform the I/O.
- Pros:
  - No thread overhead, no polling.
- Cons:
  - .. More complex to maintain list of file descriptors to monitor.

Idea 3: Implementing Sketch with epoll

- 3 Calls to implement I/O Multiplexing with epoll():

  epoll_create()

  – Returns an epoll instance.

  – We can think of this as a

  .. monitor object that maintains the monitoring list.

  epoll_ctl()

  – Allows us to

  .. add, remove, or modify a file descriptor to the epoll instance.

  - Start by monitoring socket for accept()
  - Each new FD from accept() is added to set to monitor

  epoll_wait()

  – Waits for a file descriptor to be available for I/O

ABCD: Server choices

- Match the server implementation idea with the problem it suffers:

1) Non-blocking IO in a loop

2) epoll() to watch sockets

3) Thread per client

(a) More complex code

(b) Only handle one socket at a time.

(c) More likely to use too much system resources (such as RAM), or too high kernel overhead.

(d) Wastes CPU Time

Summary

- accept() returns a new socket for each TCP client.
- Server must likely handle many sockets at once:
  - Can create a new thread per socket.
  - Can use non-blocking IO to busy-wait checking for ready sockets
  - Can use epoll() or select() to have kernel monitor sockets