**Concurrent Connection-Oriented Server Conceptual Algorithm**

# 1. Server Initialization (common to all variants)

1. CREATE a TCP socket `listen\_fd`
2. SET socket options (e.g. SO\_REUSEADDR)
3. BIND `listen\_fd` to (ANY\_ADDRESS, PORT)
4. LISTEN on `listen\_fd` with a suitable BACKLOG

# 2. Main Accept Loop

while (true) do

client\_fd, client\_addr ← ACCEPT(listen\_fd)

if (client\_fd < 0) then

log("accept failed"), CONTINUE

end if

// Dispatch to a concurrent handler:

spawn\_handler(client\_fd, client\_addr)

end while

Here, **spawn\_handler** differs by variant:

# 3. Variant A: Processes

spawn\_handler(fd, addr):

pid ← FORK()

if (pid == 0) then // Child

CLOSE(listen\_fd) // Child doesn’t accept new connections

handle\_client(fd)

EXIT(0)

else // Parent

CLOSE(fd) // Parent hands client fd off to child

end if

# 4. Variant B: Threads

spawn\_handler(fd, addr):

// Allocate storage for the fd so each thread has its own copy

allocate fd\_ptr ← new int

\*fd\_ptr ← fd

CREATE\_THREAD(thread\_fn=handle\_thread, arg=fd\_ptr)

DETACH\_THREAD // Let it clean up itself when done

handle\_thread(fd\_ptr):

client\_fd ← \*fd\_ptr

free(fd\_ptr)

handle\_client(client\_fd)

return

# 5. Variant C: Asynchronous I/O (select/poll/epoll)

// Instead of per-connection handlers, you maintain a set of active fds.

initialize fd\_set MASTER\_SET

ADD listen\_fd to MASTER\_SET

max\_fd ← listen\_fd

while (true) do

read\_set ← MASTER\_SET

ready\_set ← SELECT(read\_set) // block until one or more fds ready

if (listen\_fd in ready\_set) then // New connection

client\_fd, \_ ← ACCEPT(listen\_fd)

ADD client\_fd to MASTER\_SET

max\_fd ← max(max\_fd, client\_fd)

end if

for each fd in MASTER\_SET excluding listen\_fd do

if (fd in ready\_set) then

nbytes ← READ(fd, buffer)

if (nbytes ≤ 0) then // connection closed or error

CLOSE(fd)

REMOVE fd from MASTER\_SET

else

handle\_request(fd, buffer)

end if

end if

end for

end while

# 6. Request Handling (shared by all variants)

handle\_client(client\_fd):

loop

line ← READ\_LINE(client\_fd)

if (line is empty or error) then BREAK endif

parsed ← PARSE\_COMMAND(line) // OPEN, DEPOSIT, WITHDRAW, etc.

reply ← PROCESS\_COMMAND(parsed) // call your network API

WRITE\_LINE(client\_fd, reply)

if (parsed.cmd == QUIT or parsed.cmd == CLOSE) then

BREAK

endif

end loop

CLOSE(client\_fd)

**READ\_LINE**: reads up to \n, handles partial reads.

**PARSE\_COMMAND** / **PROCESS\_COMMAND**: command\_processor logic.

**WRITE\_LINE**: appends \n, handles partial writes if needed.