Section Plan

- What's event-driven programming?
- How to use libasync, libarpc
- Look at some event-driven code
- Reference materials:
  - libasync handout on web page
  - First lecture's slides on the rpcc compiler

- Answer questions about libasync,
  or about class papers for the exam

Simple TCP server

```
int s = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in sin;
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(5555);
bind(s, (struct sockaddr *) &sin, sizeof(sin));
listen(s, 5);

for (;;) {
    socklen_t len = sizeof(sin);
    int c = accept(s, (struct sockaddr *) &sin, &len);
    /* process request from c... */
    close(c);
}
```

Concurrent connections

- Must handle multiple clients concurrently
  - Reading from client's socket can block
- One process per client: `accept(), fork(), close()`
  - Slow, hard to share state
- One thread per client: `accept(), pthread_create()`
  - Can be hard to program: race conditions; aborting requests is tricky; have to manage threads
- Use non-blocking sockets
  - Unusual programming model

Non-blocking sockets

- Make a socket non-blocking:
  - `fcntl(s, F_SETFL, fcntl(s, F_GETFL) | O_NONBLOCK);`
  - `make_async(s);` when using libasync
- Non-blocking semantics:
  - `read(s)` could return -1, errno = EAGAIN
  - `write(s)` could write partial data, or return EAGAIN
  - `connect(s)` could return EINPROGRESS
  - `accept(s)` could return EAGAIN

Non-blocking sockets

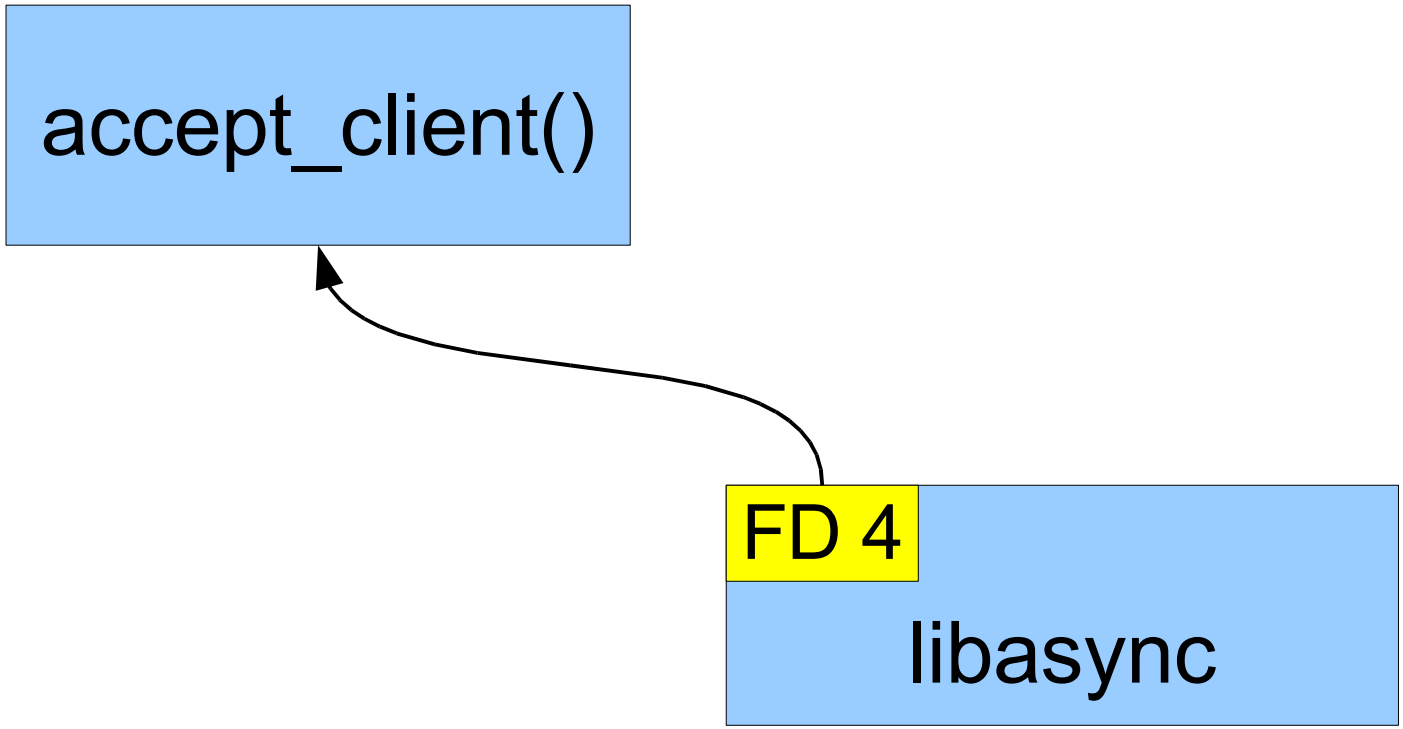
- When do we call read or write?
  - ```
    int select(int nfd, fd_set *read_fds,
                        fd_set *write_fds,
                        fd_set *except_fds,
                        struct timeval *timeout);
    ```
  - ```
    void FD_SET(int fd, fd_set *set);
    void FD_CLR(int fd, fd_set *set);
    bool FD_ISSET(int fd, fd_set *set);
    void FD_ZERO(fd_set *set);
    ```

Event-driven programming

- Overall plan:
  - Call `select()` to get pending events
  - Handle each event in turn
  - Repeat
- libasync
  - Register callbacks for events of interest
  - Run `amain()`: keeps calling `select()` and callbacks
  - Can add or remove callbacks at any time

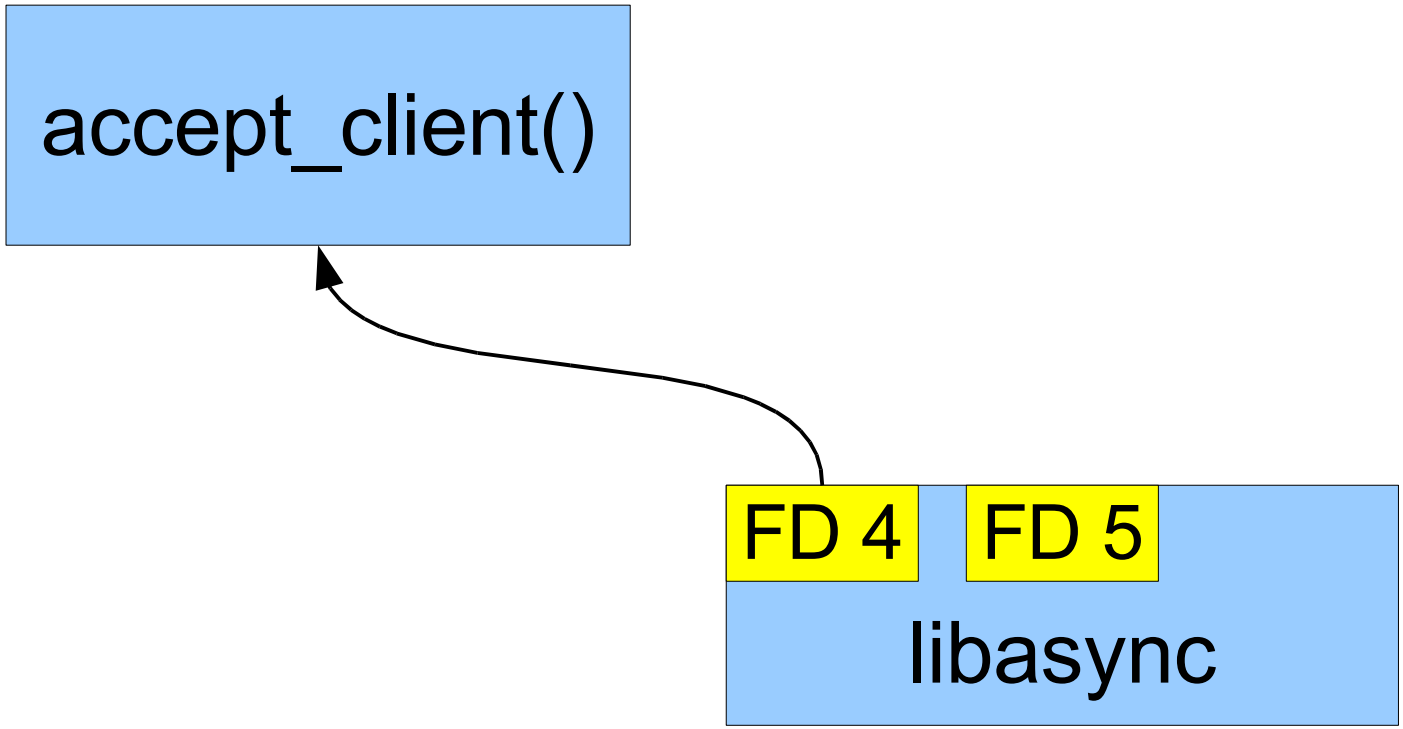
Designing an echo server

- Accept TCP connection, read message, echo the same message back



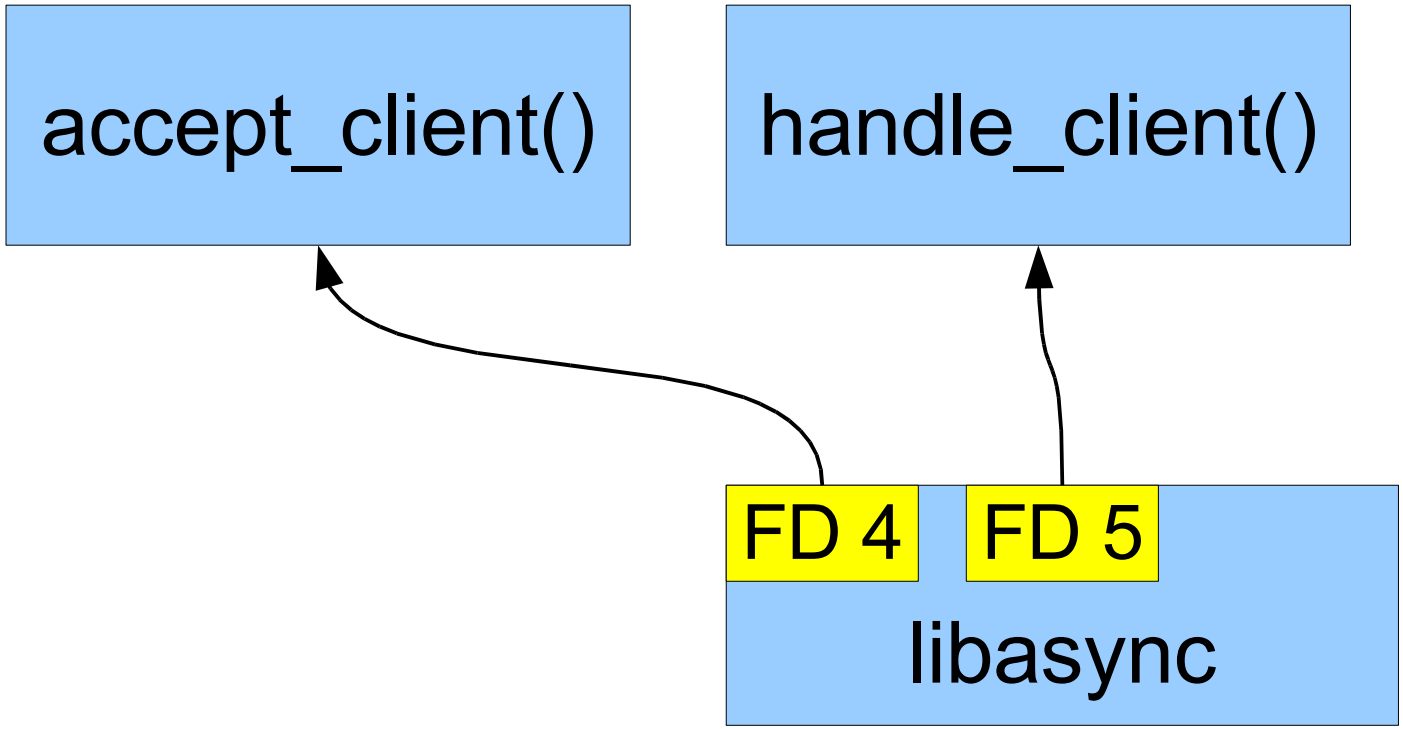
Designing an echo server

- Accept TCP connection, read message, echo the same message back



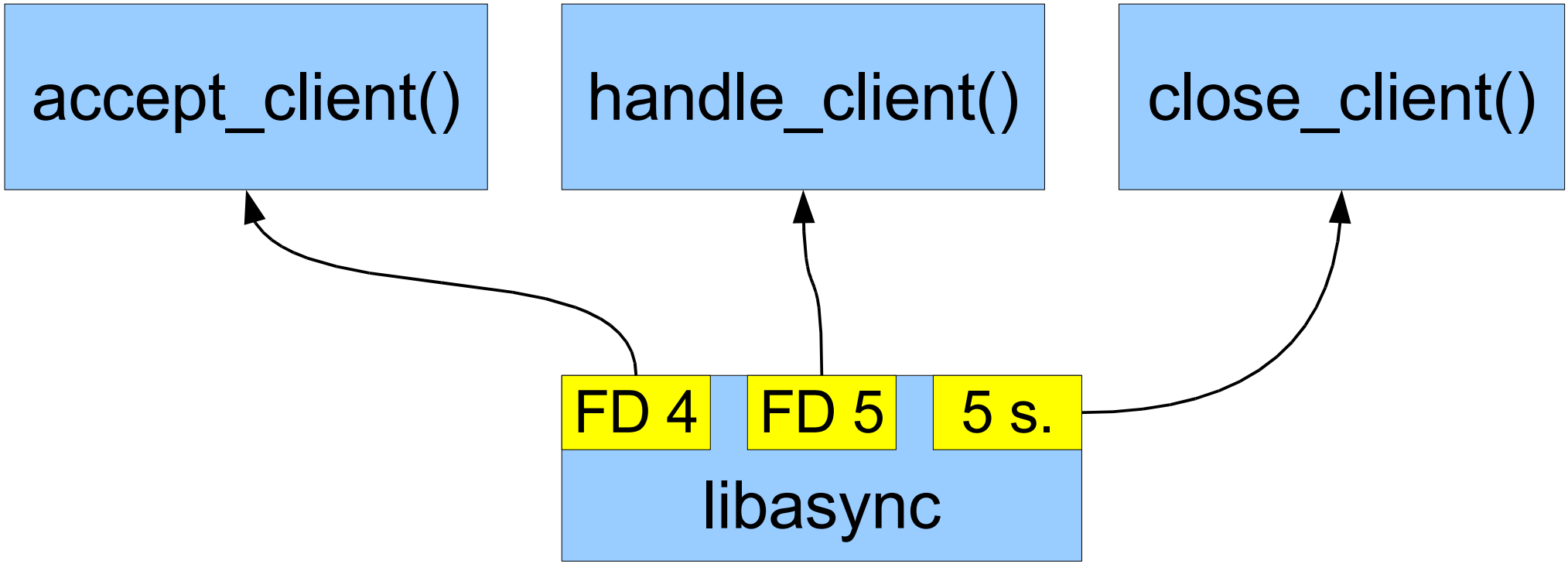
Designing an echo server

- Accept TCP connection, read message, echo the same message back

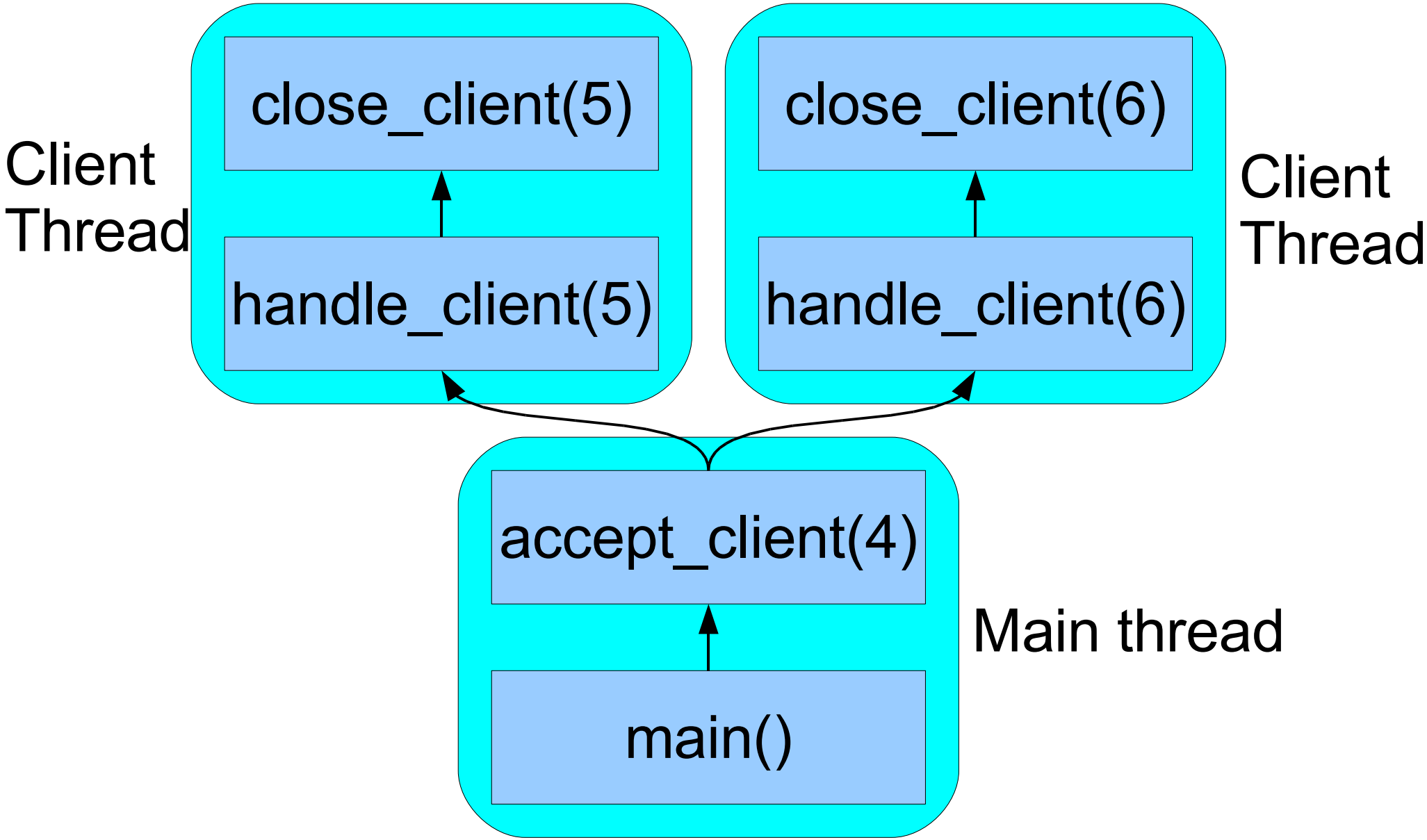


Designing an echo server

- Accept TCP connection, read message, echo the same message back



Same server using threads



libasync callbacks

- Specifies a function *and* arguments
  - ```
    void twoprint(int x, int y) {
        printf("%d %d\n", x, y);
    }
    callback<void>::ptr a = wrap(&twoprint, 1, 2);
    a();      /* prints 1 2 */
    ```
- Can have some arguments specified later
  - ```
    callback<void, int>::ptr b = wrap(&twoprint, 1);
    b(2);     /* also prints 1 2 /
    ```
- Callback type `callback<R, A1, A2>::ptr`
  - Takes two arguments of type A1, A2; returns type R

Registering callbacks

- `void fdcb(int fd, selop op, callback<void>::ptr cb`
  - `op` is either `selread` or `selwrite`

- `void delaycb(time_t sec, callback<void>::ptr cb)`
  - Invokes callback after `sec` seconds

Let's look at some real code

- Simple echo server implemented in libasync

Event-driven RPC

- libarpc: companion library that simplifies RPC
- Main abstractions: transport, client, server

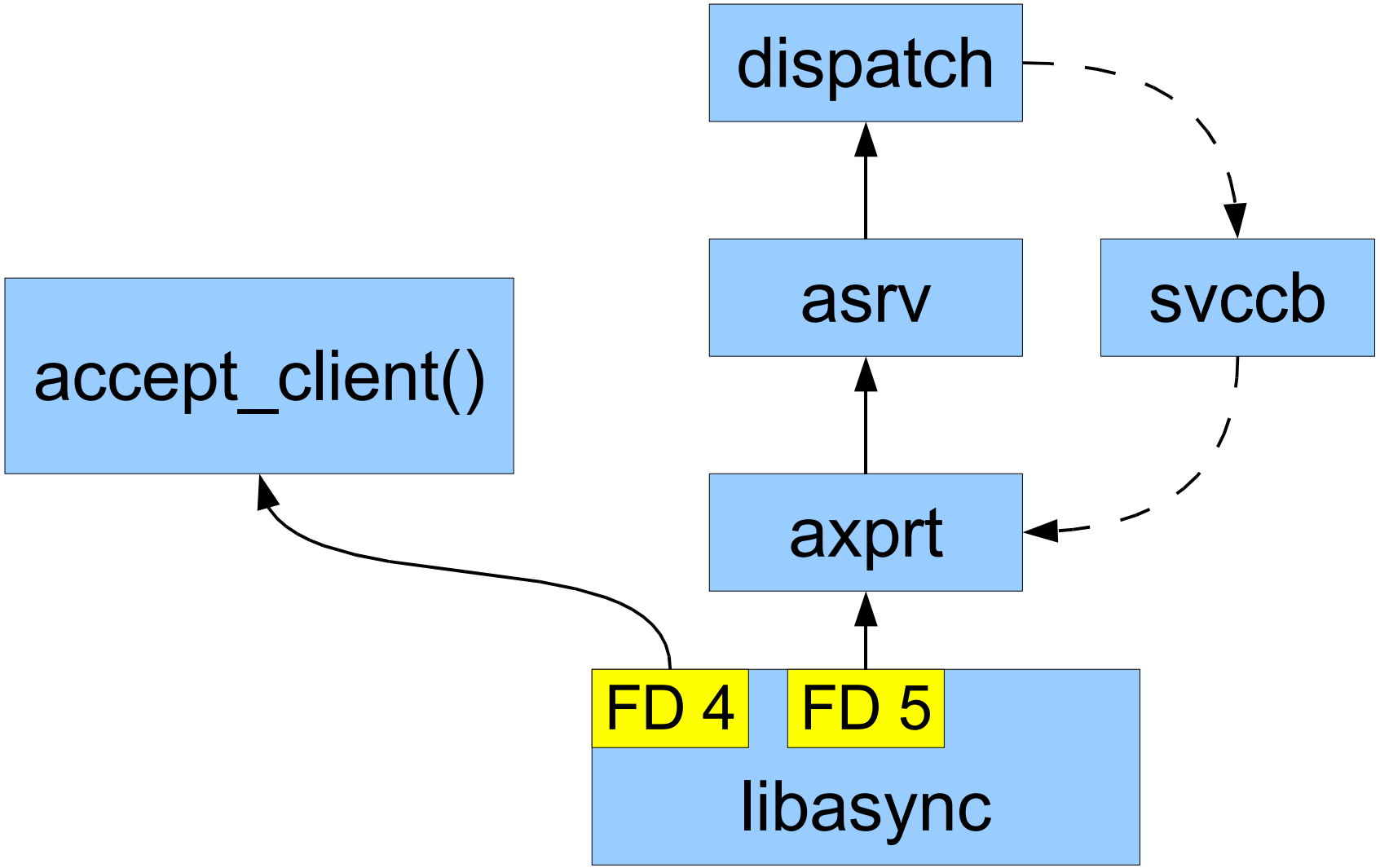
- axprt: “asynchronous transport”
  - Sends and receives variable-length messages
    - Not just a byte-stream
  - Register a callback for when a message is received
  - Outgoing messages will be queued until they can be sent

Event-driven RPC

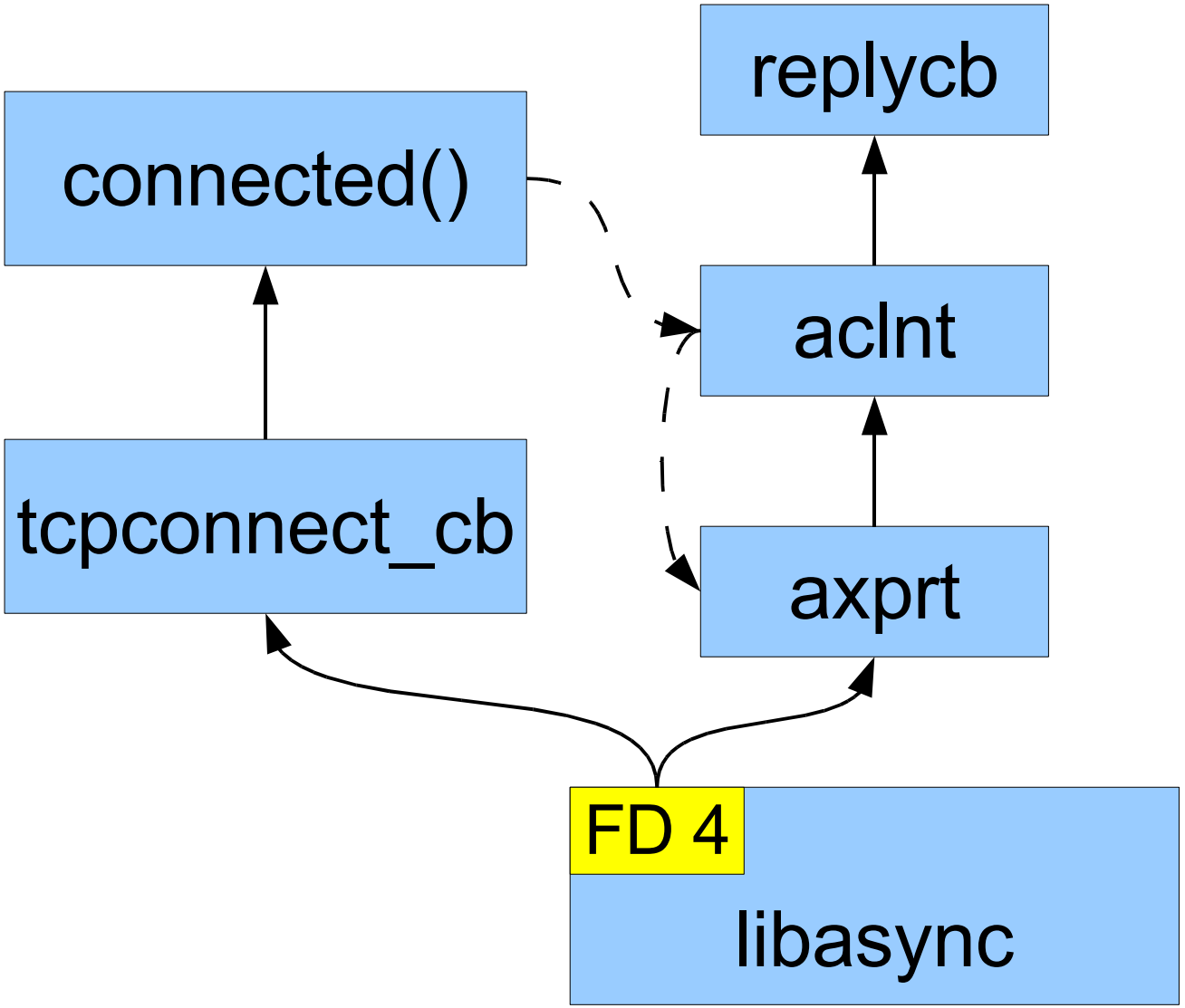
- aclnt: “asynchronous client”
  - Associated with a particular transport (axprt)
  - Issue RPC requests – callback for when we get reply
  - Note: must pre-allocate space for return value

- asrv: “asynchronous server”
  - Associated with a particular transport (axprt)
  - Register a callback for when a request is received
  - Replies are marshaled and sent back to the client

Event flow in RPC server



Event flow in RPC client



Simple RPC client / server

- Look through fetch-and-add service code
- Useful RPC debugging facilities:
  - ASRV_TRACE=10
  - ACLNT_TRACE=10

libasync provides garbage collection

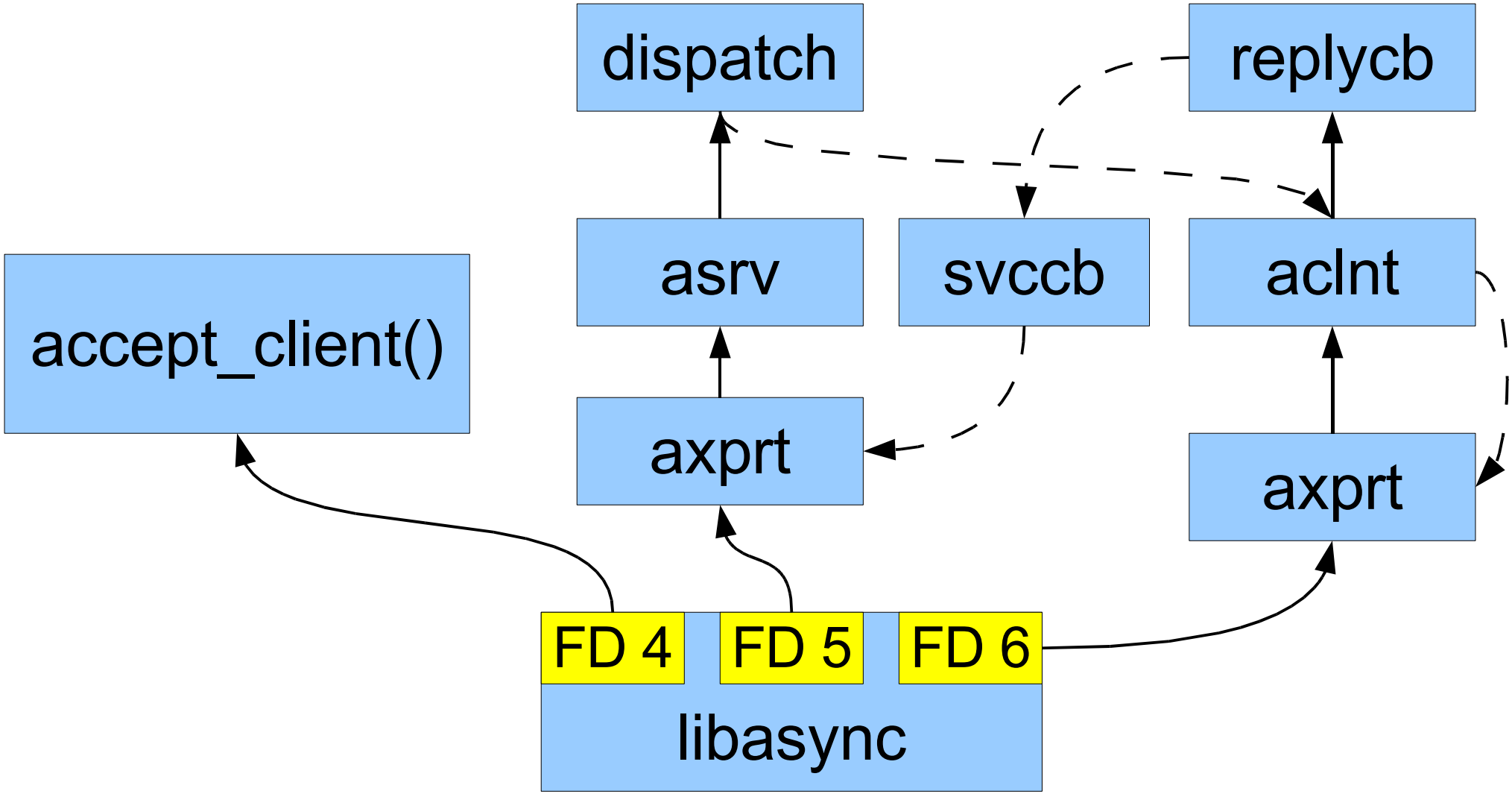
```
class A : public virtual refcount {
    void method() { ... }
};
class B : public A {};

int main() {
    ptr<A> p = New refcounted<A>();
    p->method();  // behaves much like A*

    ptr<B> q = New refcounted<B>();
    p = q;        // the first object gets deleted
}
```

- Avoid using non-ptr<> pointers to refcounted things
- To get a reference-counted pointer to “this” from within a method, use `mkref(this)`

RPC proxy server



That's about it

- For more details, look at the course web site
  - libasync handout under reference materials
  - First lecture notes for rpcc details
  - This section and code will be there too

- Questions about libasync, or midterm quiz?