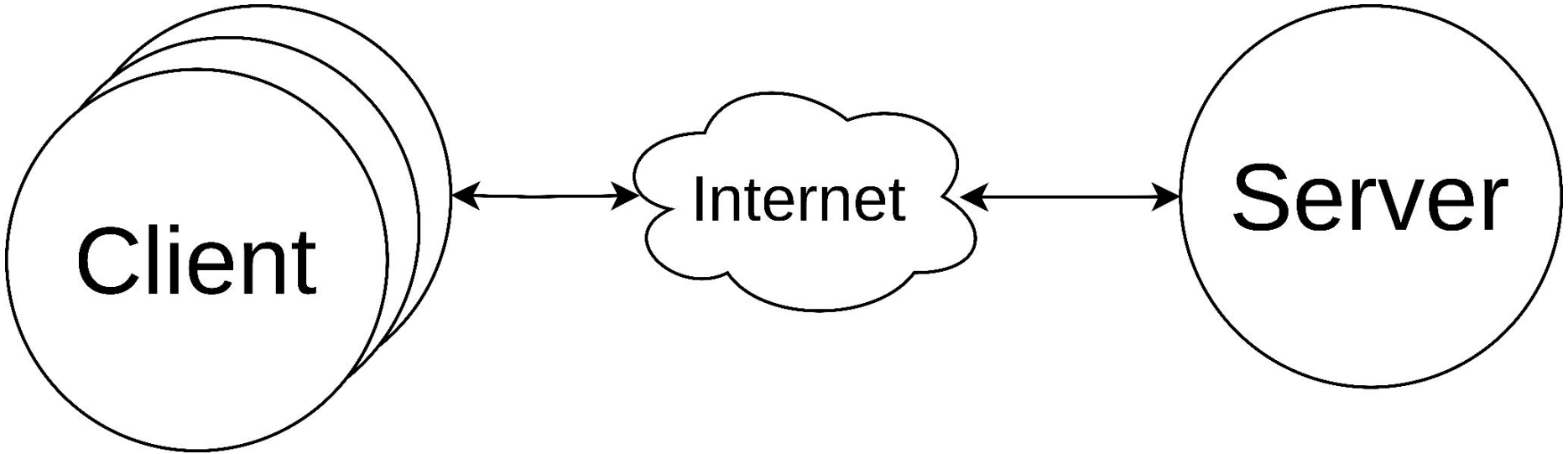
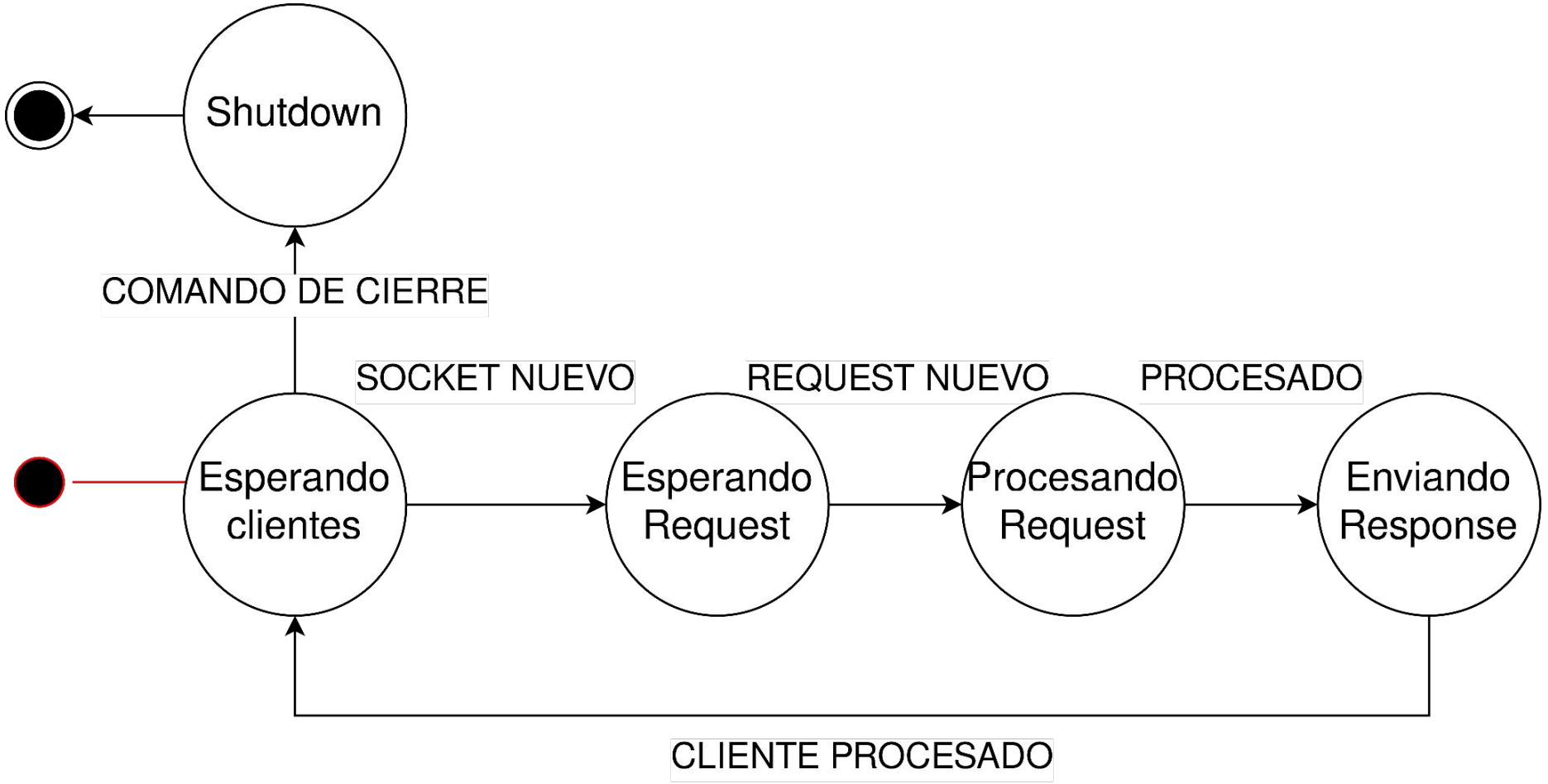
Arquitectura Client-Server

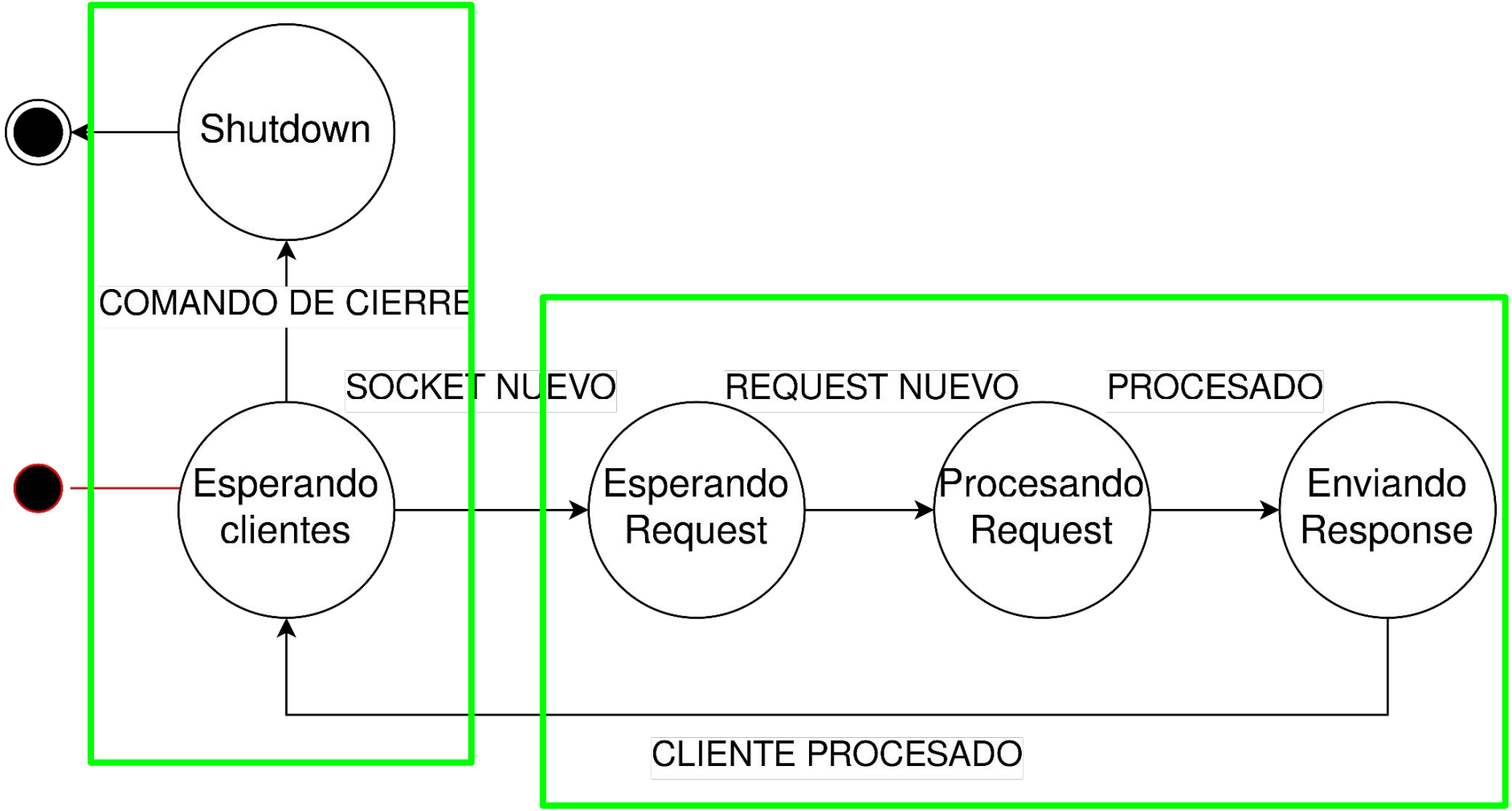
Taller de Programación I - Cátedra Veiga - FIUBA

Recap: Queremos manejar más de un cliente

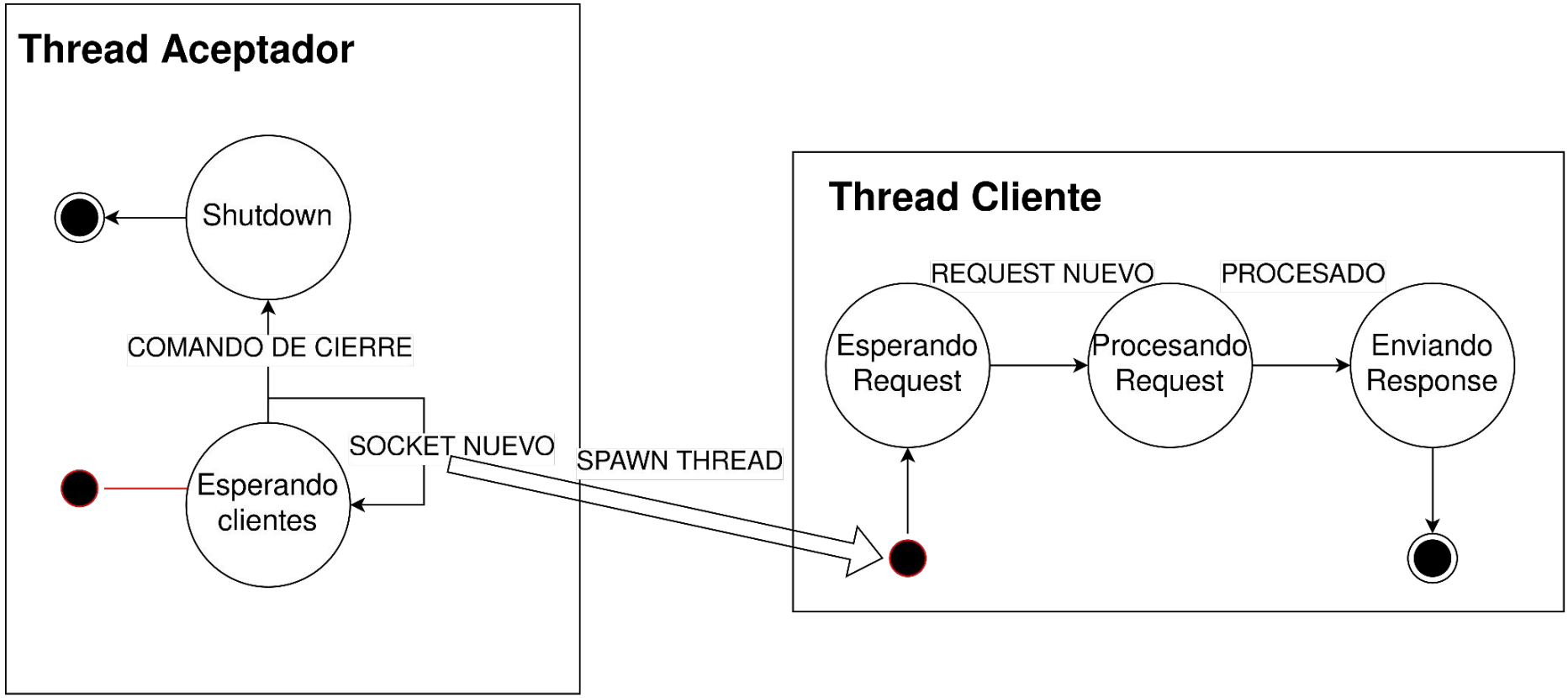


```
void serve() {
  while (_keep_running) {
    Socket peer = acceptor_socket.accept();
    Request req = peer.receive_request();
    Response res = process_request(req);
    peer.send_response(res);
  }
}
```

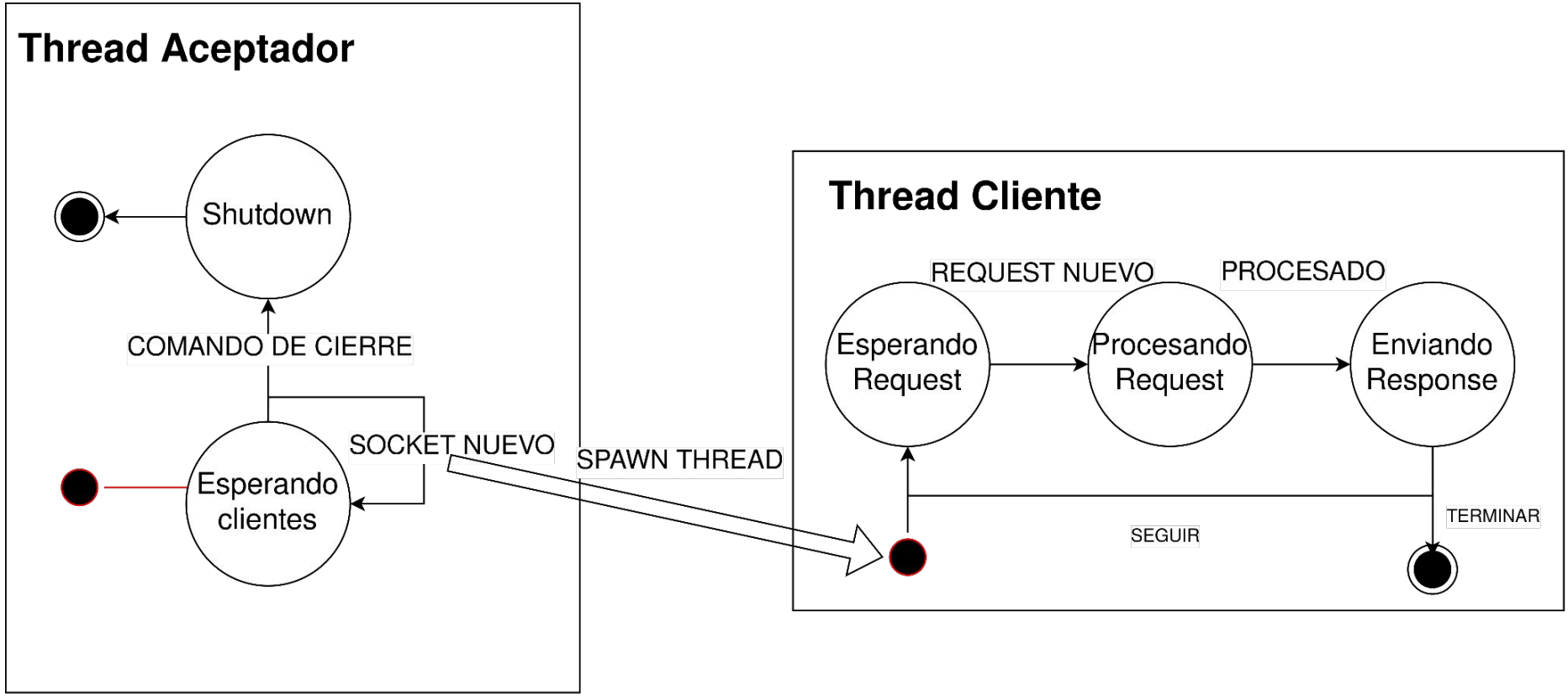

Shutdown
COMANDO DE CIERRE
SOCKET NUEVO
REQUEST NUEVO
PROCESADO
Esperando clientes
Esperando Request
Procesando Request
Enviando Response
CLIENTE PROCESADO


Shutdown
COMANDO DE CIERRE
SOCKET NUEVO
REQUEST NUEVO
PROCESADO
Esperando clientes
Esperando Request
Procesando Request
Enviando Response
CLIENTE PROCESADO

Caso 1: Thread por cliente (síncrono)



Caso 1.1: Thread por cliente Keep-Alive (síncrono)

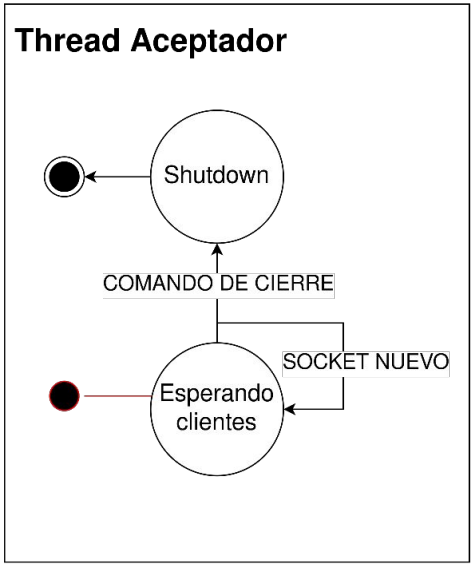


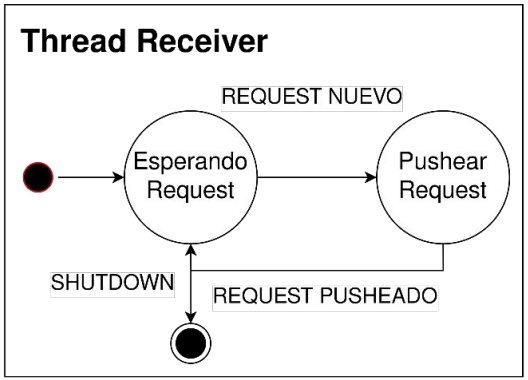
Asincronismo

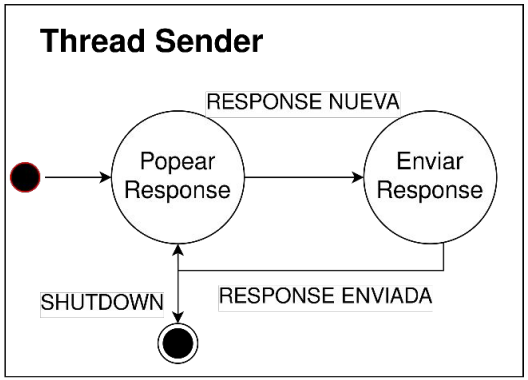
En vez de recibir, procesar el request y generar el response en el mismo hilo, podemos splitear en tres partes concurrentes:

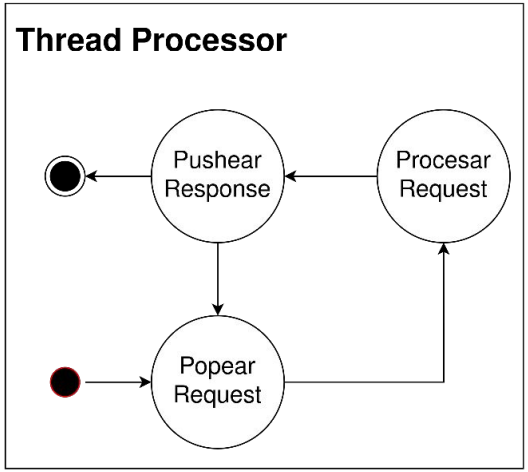
- Recibir request.
- Procesar request.
- Enviar response.

Caso 2: Dos threads por cliente Keep-Alive (asíncrono)







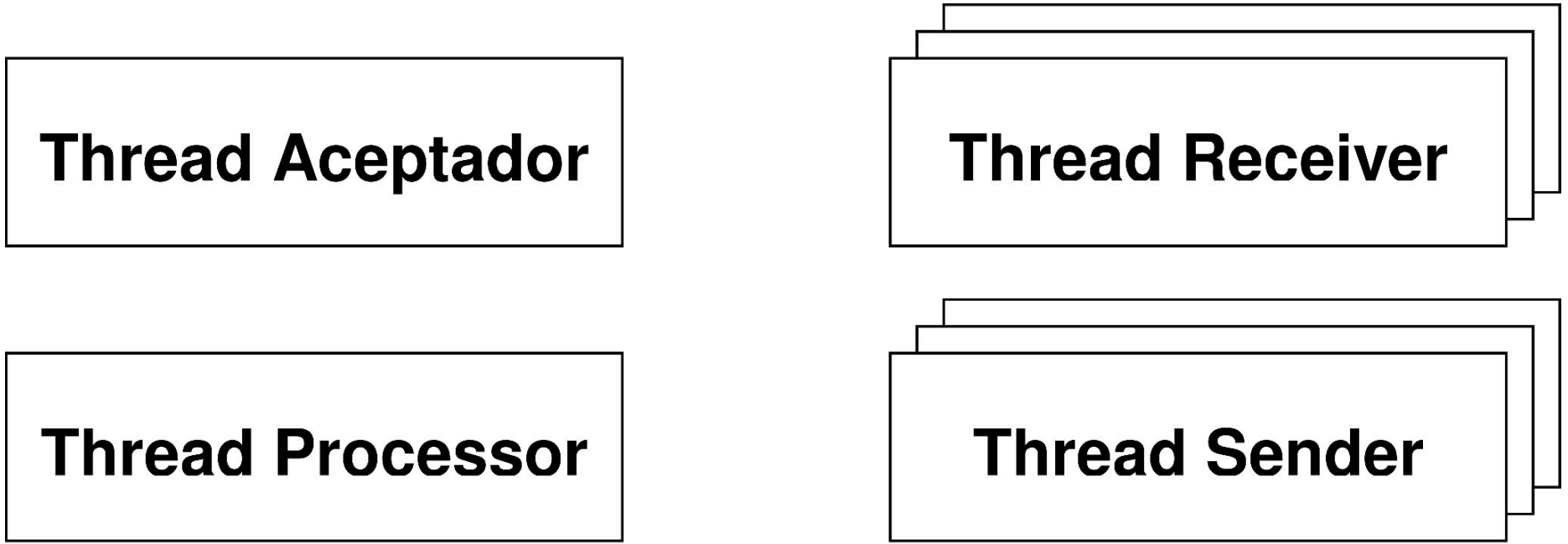


Receiver y sender comparten el mismo socket: ¿RC?

La implementación de POSIX garantiza que llamadas a recv y send son atómicas, con lo cual, si un hilo lee y otro escribe, **no hay RC**.

Más de un hilo leyendo o escribiendo sobre el mismo socket -> RC.

Servidor multithreaded

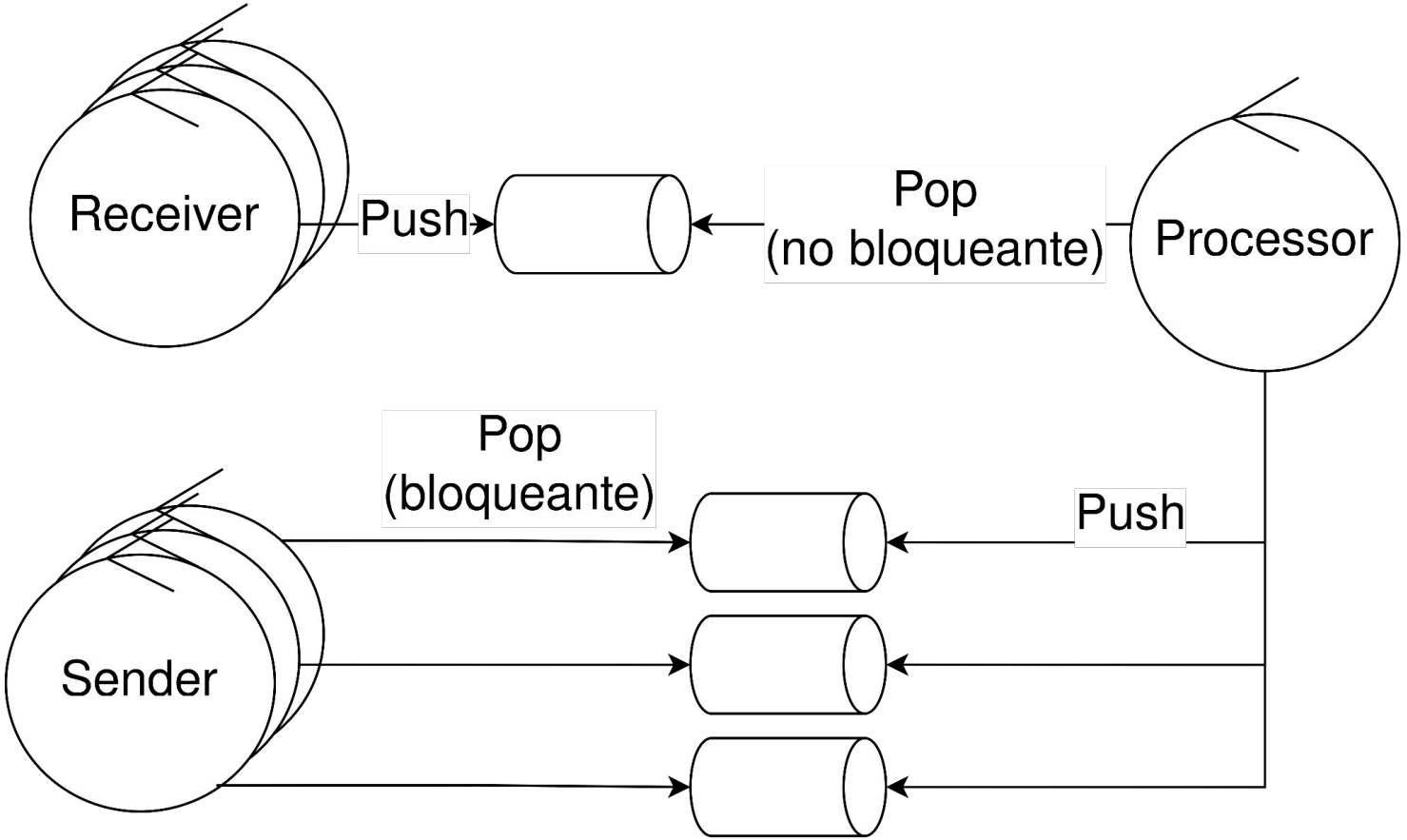


Share memory by communicating

La comunicación entre hilos receiver, sender y processor será mediante queues thread safe.

A estos hilos (receiver, sender, aceptador) los llamamos IO Threads.

Hilos, sockets y queues



Otras formas de implementar esto

- Thread pools
- Programación orientada a eventos (non-blocking IO)
- Combinación de ambas

```
class Acceptor : public Thread {
private:
    std::list<Client*> clients;
    Socket& sock;

    void run() {
        while (_keep_running) {
            auto peer = sock.accept();
            auto* client = new Client(peer);
            clients.push_back(client);
            client->start();
        }
    }
};
```

```
class Client : public Thread {
  Socket peer;

  // sync client
  void run() {
    while (_keep_talking){
      // peer.recv/peer.send
    }
  }
}
```

Problemas con esta implementación

- No podemos hacer un ***graceful shutdown*** del servidor
- **Leak** de la lista de clientes.

Graceful shutdown: Cierre ordenado del servidor. Se liberan ***todos*** los recursos (hilos, sockets, queues abiertas).

```
class Acceptor : public Thread {
    ...
    void reap_dead() {
        clients.remove_if([](auto* client) {
            auto client_dead = client->is_dead();
            if (client_dead) {
                client->join();
                delete client;
            }
            return client_dead;
        });
    }

    void shutdown() {
        for (auto* c : clients) {
            c->kill();
            c->join();
            delete c;
        }
    }
    ...
};
```

```
class Acceptor : public Thread {
    ...
    void run() {
        try {
            while (_keep_running) {
                ...
                reap_dead();
            }
        } catch (LibError& le) {
            // "falló" accept, es esperado
        } catch (std::exception& e) {
            // loggear error
        }
        shutdown();
    }
    ...
};
```

```
int main(int argc, const char** argv) {
  ...
  Socket socket(port);
  Acceptor acceptor(socket);

  std::string in;
  while (std::cin >> in; in != "q") {}

  socket.close();
  acceptor.join();

  return 0;
}
```