Arquitectura multithreading

Di Paola Martín
martinp.dipaola <at> gmail.com

Facultad de Ingeniería
Universidad de Buenos Aires

Caso 1: obtener 1 página web (sync)

Imaginate un programa de línea de comandos que descarga una única página web como lo son `httpie`, `wget` o `curl`

```
void fetch_web_page(url) {
  send_request_web_page(url);
  page = recv_web_page();

  save(page, url);
}
```

- Que operaciones son **bloqueantes**?
- Mientras el thread está bloqueado, que se podría hacer en el **mientras tanto**?

- Las operaciones bloqueantes de este (pseudo) código son `send`, `recv` y `save`.
- No se puede hacer nada mientras se envia el request ni tampoco mientras se espera la página web.
- En ambos casos esta perfecto que el thread se bloquee ya que no hay nada que se pueda hacer concurrentemente (aka, "en el mientras tanto")
- Cuando un programa hace un pedido y no avanza hasta no obtener una respuesta se dice que **la comunicación es sincrónica**.
- Comunicación sincrónica es ineficiente (no aprovechas los tiempos muertos) pero muy fácil de usar y esta presente en todos lados.
- Por ejemplo cuando haces un `file.read()` estas haciendo un pedido al OS y tu programa no continua hasta no obtener una respuesta. Es una comunicación sincrónica con el OS.

Caso 2: web scrawler (sync)

Imaginate que ahora tenes un web scrawler: un programa de se descarga todo un sitio web.

```
void download_web_site(urls) {
  for (url in urls) {
    fetch_web_page(url);
  }
}
```

Y ahora?

- Que operaciones son **bloqueantes**?
- Mientras el thread está bloqueado, que se podría hacer en el **mientras tanto**?

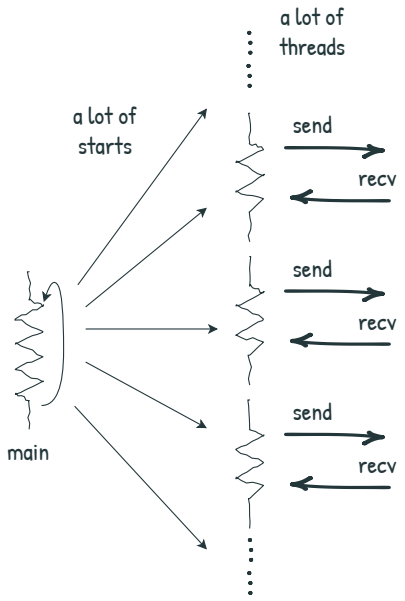
- La comunicación es sincrónica nos fuerza a descargar de a una página a la vez: **secuencial y lento**.
- Es muy probable que ni la red ni el servidor esten saturados y podrían permitir más tráfico.
- Lease se debería poder enviar más requests y recibir más páginas web en **paralelo**.

Caso 3: web scrawler (sync + threads)

```
void download_web_site(urls) {
  std::list<Fetcher*> threads;

  for (url in urls) {
    th = new Fetcher(url);
    th->start();

    threads.push_back(th);
  }
  // hacer joins de los threads
}
```

```
struct Fetcher: public Thread {
  Socket skt;
  void run() {
    fetch_web_page(this->url);
  }
}
```

- Se hacen varias descargas en paralelo por lo que es mejor q la versión secuencial.
- **Pero** se fuerza a que cada `Fetcher` tenga **su propio socket** (de otro modo habría una RC sobre el socket compartido) lo que implica tener **múltiples conexiones**.
- Establecer muchas conexiones es costoso: el OS tiene un **máximo de conexiones posibles** y el servidor puede imponernos también un límite.
- Lanzar muchos threads también tiene un **costo en memoria**: recordar que cada thread tiene su propio stack y eso requiere memoria.

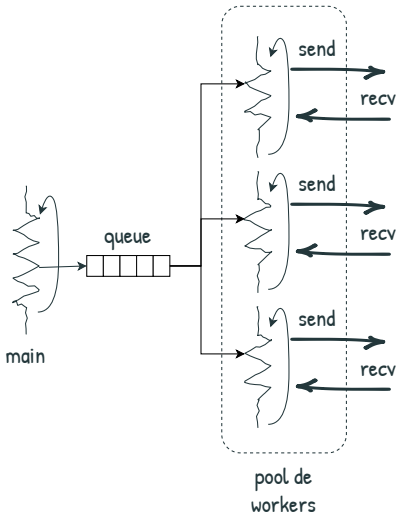


Caso 4: web scrawler (sync + workers)

```
void download_web_site(urls) {
  // creamos el pool de workers
  std::vector<Fetcher*> workers(N);
  for (int i = 0; i < N; i++) {
    workers[i] = new Fetcher(q);
    workers[i]->start();
  }

  // cargamos la queue,
  // cada worker ira tomando
  // una url y la procesara
  for (url in urls) {
    q.push(url);
  }

  // hacer joins de los threads
}
```

```
struct Fetcher:public Thread {
  Socket skt;
  Queue q;

  void run() {
    while (...) {
      url = q.pop()
      fetch_web_page(url);
    }
  }
}
```


- Se hacen varias descargas en paralelo pero hay un **límite autoimpuesto de N threads**: se requiere N sockets distintos con N conexiones distintas.
- Los threads que esperan por una tarea, la resuelven y vuelven a esperar son llamados **workers**. En este caso todos los workers hacen la misma tarea (fetchear una página web) pero podría no ser asi. Podrían recibir **objetos polimórficos**.
- Cada worker toma de **la misma queue compartida** una task (url) y la procesa tan rápido como puede (hay un balanceo natural de trabajo entre los workers).
- Un grupo de thread workers se lo conoce como **pool de workers**. Típicamente son N threads pre-creados aunque en ocaciones se hace el N variable.
- Un pool de workers se usa cuando se quieren hacer tareas en background **pero** para enviar/recibir datos tienen un par de issues.
- Con N threads, habra a lo sumo N sends / recvs paralelos y **puede que la red quede poco usada** (o sea, podríamos enviar/recibir mucha mas data por la red) o el **servidor quede sub-usado** (podría manejar muchos más pedidos dentro de una misma conexión).

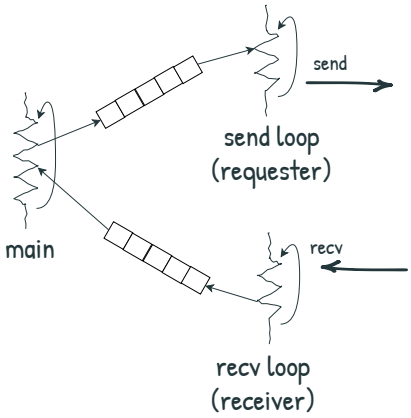


Caso 5: web scrawler (async)

```
void download_web_site(urls) {
  Socket skt(...); // 1 conexion

  req_th = new Requester(
                    requests_q, skt)
  rec_th = new Receiver(
                    responses_q, skt)

  // hacer los starts

  for (url in urls) {
      requests_q.push(url);
  }

  for (url in urls) {
      page = responses_q.pop();
      save(page, url);
  }
```

```
struct Requester:public Thread {
  Socket& skt;
  void run() {
    while (...) {
      url = q.pop();
      send_request_web_page(url);
    }
  }
}

struct Receiver:public Thread {
  Socket& skt;
  void run() {
    while (...) {
      page = recv_web_page();
      q.push(page);
    }
  }
```

- Los threads que sólo se dedican a enviar y a recibir datos se los llaman **threads de comunicación**.
- Al tener un único thread que envia, este estará enviando data tan rápido como es posible, **hasta el punto de o saturar la red o saturar el servidor** (lo que primero suceda). Lo mismo pasa con el otro thread.
- Poner mas threads en paralelo para enviar/recibir **no** mejorará la situación aunque hay 2 excepciones.
- Podría pasar que la red y el servidor tienen **mucha** capacidad y el thread de comunicación llegue a un tope **antes** de saturarlos. **Ahi tu CPU es el factor limitante** y si tenes mas cores poner otro thread mejoraría la performance. En la práctica 99% de las veces **nunca** se da este caso (tu CPU es mucho más rápida).
- Podría pasar que el servidor te imponga un límite en la transferencia por conexión (bandwidth). Tener múltiples threads implica múltiples conexiones y mejoraría la perforamnce (usarias N threads de comunicación como lo viste en el pool de workers). `aria2c` usa esta estrategia. Ojo que los servidores puede **no gustarle esto** y pueden bannearte ya que en escencia estas "sorteando" un límite impuesto por ellos.!
- Hay RC sobre `skt` **compartido**? **No**. Podes hacer sends en **un solo thread** y recvs en **otro solo thread** (más de 1 thread y tendras RC)



Viste el deadlock?

Ves el deadlock?

```
void download_web_site(urls) {
  req_th = new Requester(
                    requests_q, skt)
  rec_th = new Receiver(
                    responses_q, skt)

  // hacer los starts

  for (url in urls) {
      requests_q.push(url);
  }

  for (url in urls) {
      page = responses_q.pop();
      save(page, url);
  }

  // hacer los joins
```

```
struct Requester:public Thread {
  Socket& skt;
  void run() {
    while (...) {
      url = q.pop();
      send_request_web_page(url);
    }
  }
}

struct Receiver:public Thread {
  Socket& skt;
  void run() {
    while (...) {
      page = recv_web_page();
      q.push(page);
    }
  }
```


- El thread principal pushea las urls (1) que son pop'eadas por el `Requester` quien envia a su ves los requests (2).
- El servidor acepta y procesa los requests y envia las respuestas. El `Receiver` las recibe y las pushea a la otra queue (3).
- **Pero,** el thread principal sigue pusheando urls y **no** esta haciendo ningun pop (4).
- `responses_q` eventualmente se llenara, el push se bloqueara y el `Receiver` dejara de leer del socket (5).
- Eventualmente los buffers del servidor se llenaran y este dejara de enviar respuestas y de aceptar nuevos requests.
- El `Requester` entonces se bloquera en el send y dejara de pop'ear urls (6) lo que hara que `requests_q` se llene y el main se bloque (7).
- Solución? O el main balancea algunos push con algunos pop o los pops los hace otro thread.
- Otra opción sería usar `UnboundedQueues`: el deadlock teórico seguiría ahi pero si la cantidad de urls no es infinita, las queue "nunca" se llenaría ni los push se bloquearían.
- La moraleja es que **es difícil** diseñar una arquitectura robusta, performante y libre de bugs: **la simplicidad es tu aliada**.

Caso 6: cliente de chat (sync)

Imaginate un programa con interfaz gráfica para chat

```
void main() {
  while (not quit) {
    msj = read_from_keyboard();
    if (msj) {
      send_my_message(msj);
    }

    msj = recv_theirs_messages();
    draw(msj);
  }
}
```

- Que operaciones son **bloqueantes**?
- Mientras el thread está bloqueado, que se podría hacer en el **mientras tanto**?

- **`read_from_keyboard`** puede bloquearse asi como **`send_my_message`** y **`recv_theirs_message`** (podemos suponer que el **`draw`** es rápido y no bloqueante).
- Deberíamos poder **recibir** los mensajes aun cuando no enviemos ninguno nosotros!
- Deberíamos poder **enviar** nuestros mensajes aun cuando nadie nos escriba a nosotros!
- En resumen: el recibir y enviar **no estan correlacionados**

Caso 7: cliente de chat (async + bug)

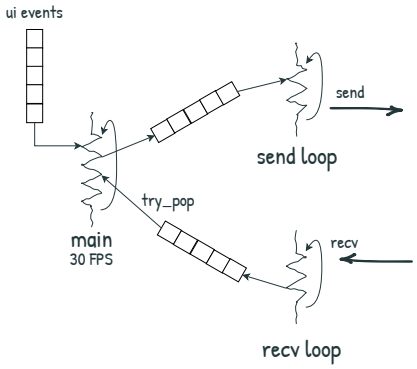
```
void main() {
  while (not quit) {
    msj = non_blocking_read_from_keyboard();
    if (msj) {
      sender_q.push(msj);
    }

    msj = receiver_q.try_pop();
    draw(msj);
  }
}
```


- Tendremos 2 threads de comunicación. El **`main`** pushea los mensajes a la **`sender_q`** para que un thread lo envie por la red; el **`main`** saca de la queue **`receiver_q`** los mensajes recibidos por el otro thread.
- Podríamos usar el mismo truco para la lectura del teclado (tendríamos un thread q se bloquea en el **`read_from_keyboard`** y nos pushea los mensajes mientras que **`main`** los saca **pero...**
- Casi todas las librerías gráficas exigen que su código sea llamado **explícivamente desde el thread principal**.
- Por suerte estas librerías ofrecen ademas versiones **no-bloqueantes**: **`non_blocking_read_from_keyboard`**
- Ahora **`main`** **no tiene ninguna operación bloqueante** por lo que se transformó en un un **busy loop** (y nos va a quemar la CPU). Hay q arreglarlo.

Caso 8: cliente de chat (async)

```
void main() {
  while (not quit) {
    msj = non_blocking_read_from_keyboard();
    if (msj) {
      sender_q.push(msj);
    }

    while (msj = receiver_q.try_pop()) {
      draw(msj);
    }

    sleep(1/30); // fix me
  }
}
```


- El loop del **`main`** trabaja 30 veces por segundo o **30 frames por segundo (FPS)**. En la práctica hardcodear un **`1/30`** es **mala idea**.
- Cuanto dormir con el **`sleep`** **debe ser ajustado en cada iteracion** para **compensar defasajes** y asi mantener un **FPS relativamente constante**.
- Esto esta explicado con detalle en este post: `https://book-of-gehn.github.io/articles/2019/10/23/Constant-Rate-Loop.html`
- Ahora que le pusimos un freno al busy loop, que pasa si nos envian 300 mensajes? Si hacemos **un único `try_pop` por ciclo** tardaríamos 300 * 1/30 = 10 segundos en verlos!
- Nada nos obliga a sacar de **`receiver_q`** de aun mensaje por ciclo: lo que hacemos entonces es **sacar todos** los mensajes hasta q la queue quede vacía.



Caso 9: singleclient server (falta como cerrarlo)

```
void main() {
  while (...) {
    peer = aceptador_sk.accept();

    // se habla con un cliente
    while (...) {
      // peer.send() / peer.recv()
    }

    peer.shutdown(); peer.close()
  }
}
```

- Que operaciones son **bloqueantes**?
- Mientras el thread está bloqueado, como harías para **desbloquearlo y cerrar** el servidor?

- Un singleclient server atiende y conversa de a un cliente a la vez. Suelen usarse para implementar servicios simples en embebidos.
- Como operaciones bloqueantes tiene el `accept`, `send` y `recv`
- Como se haría para cerrar este servidor? Típicamente se le envia una señal (`sigint` o Ctrl-C). El servidor debe programar un signal handler para atrapar la señal y cerrar ordenamdamente, de otro modo un Ctrl-C termina en un crash.
- Pero hay otra solución: como el thread estara **bloqueado** en `accept`, `send` o `recv`, **otro thread** debe esperar la condición de cierre.

Caso 10: singleclient server

```
void main() {
  Socket sk; // socket aceptador
  acep_th = Aceptador(sk);
  acep_th.start()

  while (std::cin.getc() != 'q') {
  }

  sk.shutdown() ; sk.close()
  acep_th.join()
}
```

```
struct Aceptador:public Thread{
 Socket& sk;

 void run() {
  while (/*sk not closed*/) {
    peer = sk.accept();

    // se habla con un cliente
    while (...) {
      // peer.send() / peer.recv()
    }

    peer.shutdown(); peer.close()
  }
}
```

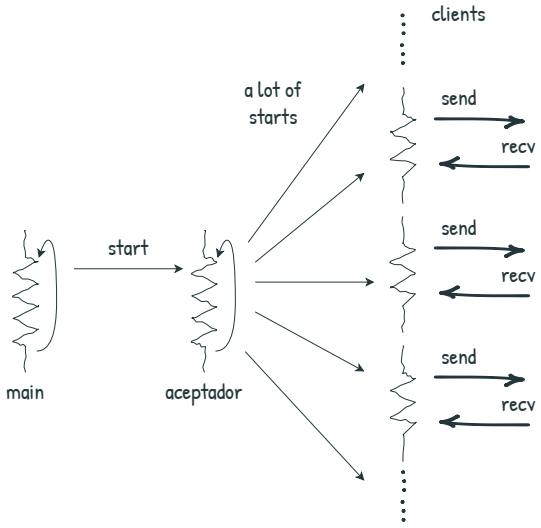

- El thread `Aceptador` es quien acepta y habla con los clientes de a uno a la vez.
- El thread principal es quien lanza a el `Aceptador` **pasandole una referencia** del socket. Podrías hacerlo al revés: el `Aceptador` tiene un socket y el main le pide una referencia, como prefieras.
- El main luego espera por la **condición de cierre**. En este caso, recibir el caracter `'q'` de la entrada estandar.
- El `Aceptador` estara eventualmente bloqueado en `sk.accept()`. Para desbloquearlo **el main le cierra el socket del aceptador**.
- El socket aceptador esta **compartido** por el main y el thread `Aceptador`, hay RC? No: el OS garantiza que un socket puede ser cerrado por un thread mientras es usado por otro.
- Que sucede si `Aceptador` esta bloqueado en `send` o `recv` mientras habla con un cliente? Necesitamos separar el `sk.accept()` del `send` y `recv`.

Caso 11: multiclient server (con leaks)

```
void main() {
  Socket sk; // socket aceptador
  acep_th = Aceptador(sk);
  acep_th.start();

  while (std::cin.getc() != 'q') {
  }

  sk.shutdown() ; sk.close()
  acep_th.join()
}

struct Client:public Thread{
 Socket sk

 void run() {
    // sk.send() / sk.recv()
 }
```

```
struct Aceptador:public Thread{
 Socket& sk;
 std::list<Client*> clients;

 void run() {
  while (/*sk not closed*/) {
    peer = sk.accept();

    th = new Client(
                std::move(peer)
              )
    th->start()

    clients.push_back(th);
  }
}
```


- El `Aceptador` acepta pero no habla con los clientes. La **comunicación** queda a cargo de `Client`.
- Como es `Aceptador` quien obtiene el socket `peer`, debe **pasarle el ownership** a `Client`.
- Si el servidor se comunica de forma **sincrónica** con el cliente, **un solo thread** `Client` te servira. Es el mismo caso del `Fetcher` del caso 4. Por ejemplo un servidor web simple que recibe 1 request, lo procesa y envia 1 respuesta, siempre de a 1 a la vez.
- Si el servidor se comunica de forma **asincrónica** (recibe y envia mensajes independientemente), **necesitaras 2 threads** `ClientSender` y `ClientReceiver`, los mismos **2 threads de comunicación** del caso 5. Por ejemplo un servidor web recibe requests y los despacha mientras que en paralelo envia al cliente las respuestas. Un proxy web es un ejemplo.
- El `Aceptador` mantiene una **lista de clientes**. Ves el leak?
- El leak más obvio es que al finalizar no se hacen los `delete` ni `join`
- El leak más sutil es que durante la vida del servidor **muchos clientes iran finalizando antes de que el servidor cierre**: hay que **recolectarlos** durante.



Reaper (Thread Aceptador)

```
void Aceptador::reap_dead() {
  clients.remove_if([](Client* c){
      if (c->is_dead()) {
          c->join();
          delete c;
          return true;
      }
      return false;
  });
}
void Aceptador::kill_all() {
  for (auto& c : clients) {
      c->kill();
      c->join();
      delete c;
  }
  clients.clear();
}
```

```
struct Aceptador:public Thread{
 Socket& sk;
 std::list<Client*> clients;

 void run() {
  while (/*sk not closed*/) {
    peer = sk.accept();

    th = new Client(
                std::move(peer)
            )
    th->start()

    reap_dead();
    clients.push_back(th);
  }
  kill_all();
}
```


- Luego de aceptar a un cliente, **`Aceptador`** recorre los clientes en busqueda de clientes muertos (threads que ya terminaron), los joinea y libera sus recursos sacandolos de la lista (ver **`std::remove_if`**). Esto es un **`reap`** o "garbage collection".
- Una variante sería q **`Aceptador`** tenga una **`deads`** queue y que cada thread cliente se registre en ella (**`push`**). Luego **`reap_dead`** hace **`pop`** y los libera. Es más eficiente pero cuidado que también hay que sacarlos de la lista **`clients`** y ya no es tan eficiente.
- Cuando el servidor y el **`Aceptador`** finalizan, este mata o frena todos los clientes aun vivos en **`kill_all`**.
- Por que "matarlos"? En principio un cliente puede hablar con el servidor **indefinidamente**: es necesario que se entere que el servidor se esta cerrando.
- **Ojo!**. Algunas implementaciones de bajo nivel de **`pthread`** (POSIX threads) permiten matar a un thread. **No hacerlo**. Esos kills de bajo nivel te destruyen el thread sin que liberen los recursos.
- Hay que **implementar uno mismo el kill para no tener leaks ni corrupciones**.

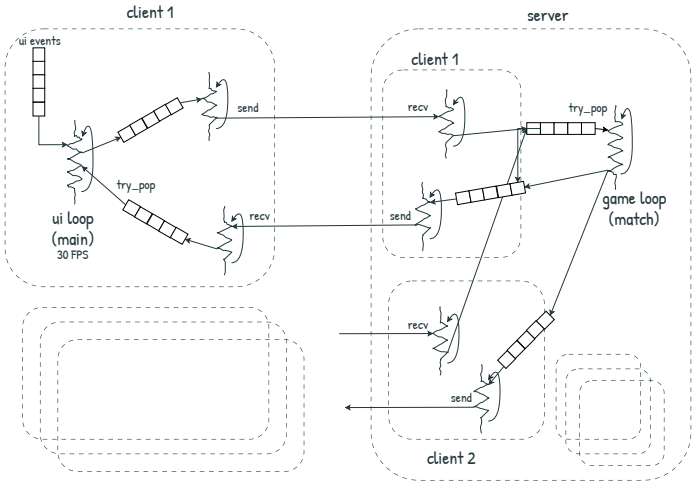
is_dead y kill (Thread Client)

```
bool Client::is_dead() {
  return not is_alive;
}

// violento pero garantizado
void Client::kill() {
   keep_talking = false;
   sk.shutdown();
   sk.close();
}

// polite pero no garantizado
void Client::kill() {
   keep_talking = false;
}
```

```
struct Client:public Thread{
 Socket sk; // peer skt

 std::atomic<bool> keep_talking;
 std::atomic<bool> is_alive;

 void run() {
  is_alive = keep_talking = true;
  while (keep_talking) {
      // sk.send() / sk.recv()
  }

  is_alive = false;
}
```


- Frenar un thread correctamente **depende de la aplicación en cuestión**, no hay una solución general.
- Para el caso de un thread de comunicación que esta hablando con un cliente hay que **marcar que no debe hablar más** (**`keep_talking=false`**). **`Client`** detectará la condición y podrá finalizar la comunicación incluso podrá enviarle un **mensaje de despedida/cierre** al cliente. Esta técnica es "polite pero sin garantias".
- "sin garantias"? El thread puede **bloquearse** en un **`send`** o **`recv`**. Para asegurase el fin del thread, hay que **forzar el cierre de su socket** lo que destraba el bloqueo. La contra es que la comunicación se corta abruptamente, es "violento pero con garantias".
- Y si el thread esta bloqueado en otra operación? Como en un **`queue.pop()`** o **`file.read()`**? **No hay una solución genérica**. Tendras que diseñar e implementar un **`kill`** específico.
- En cambio **`is_alive`** y **`is_dead()`** es genérico y podría (deberían) estar en la clase padre **`Thread`**.
- **`is_dead()`** y **`kill()`** **se llaman del thread `Aceptador`** y acceden a vars **compartidas** con el **thread `Client`**. Hay RC? **No**. El OS garantiza no RC en **`sk`** si se llama a **`shutdown`**/**`close`**; La stdlib garantiza **modificaciones atómicas** sobre **`is_alive`** y **`keep_talking`** por ser **`atomic<bool>`** (no RC tampoco)

Caso 12: cliente - servidor (async + game loop)



Resumen - 1, 2 o N Threads?

- Detectar que se **bloquea** y preguntarse si se puede hacer algo en el **mientras tanto**.
  - Verificar que realmente haya ganancia (puede que el cuello de botella este en otro lado)
  - Si hay ganancia usar threads para ganar concurrencia y/o operaciones non-blocking para no bloquearse.
  - Y nunca abusar de lanzar threads por que si (caso 3)
- Cada situación es **distinta**.
  - Hay escenarios puramente sincrónicos (caso 1) y otros puramente asincrónicos (caso 5); hay veces q no hay una solución sino múltiples con sus pros y contras (casos 4 y 5)
  - No es trivial (ver deadlock del caso 5)
- **Pools** y **threads de comunicación** son 2 diseños **pero hay más** tanto en diseño de 1 aplicación multithread o de una aplicación distribuida (multihost). Incluso hay diseños sin threads, **orientados a eventos**.

Resumen - Compartir o no compartir?

- Reconocer que objetos son compartidos por los threads
  - Preferir no compartir y en cambio pasarlos entre los threads via blocking queues (caso 5)
  - Sino, preguntarse, hay race condition? Justificar siempre con documentacion que lo respalde
  - Ante una posible RC, usar monitores y locks.
- Recordar que para un mismo socket:
  - Hacer `send` en un thread y `recv` en otro esta OK.
  - Hacer `shutdown`/`close` en un thread y `send`/`recv`/`accept` en otro esta OK.
  - Cualquier otra cosa y tendras una RC.

Resumen - Cliente - Servidor

- El aceptador debe
  - Aceptar nuevos clientes.
  - Recolectar clientes finalizados (reap)
  - Al finalizar, forzar el cierre de los clientes (kill)
- Se puede tener 1 o 2 threads de comunicación por cliente.
- Aunque en implementaciones más eficientes, se usa un pool de workers y un dispatcher por eventos (se trabaja con sockets no bloqueantes).