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**?

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);
  }
}
```

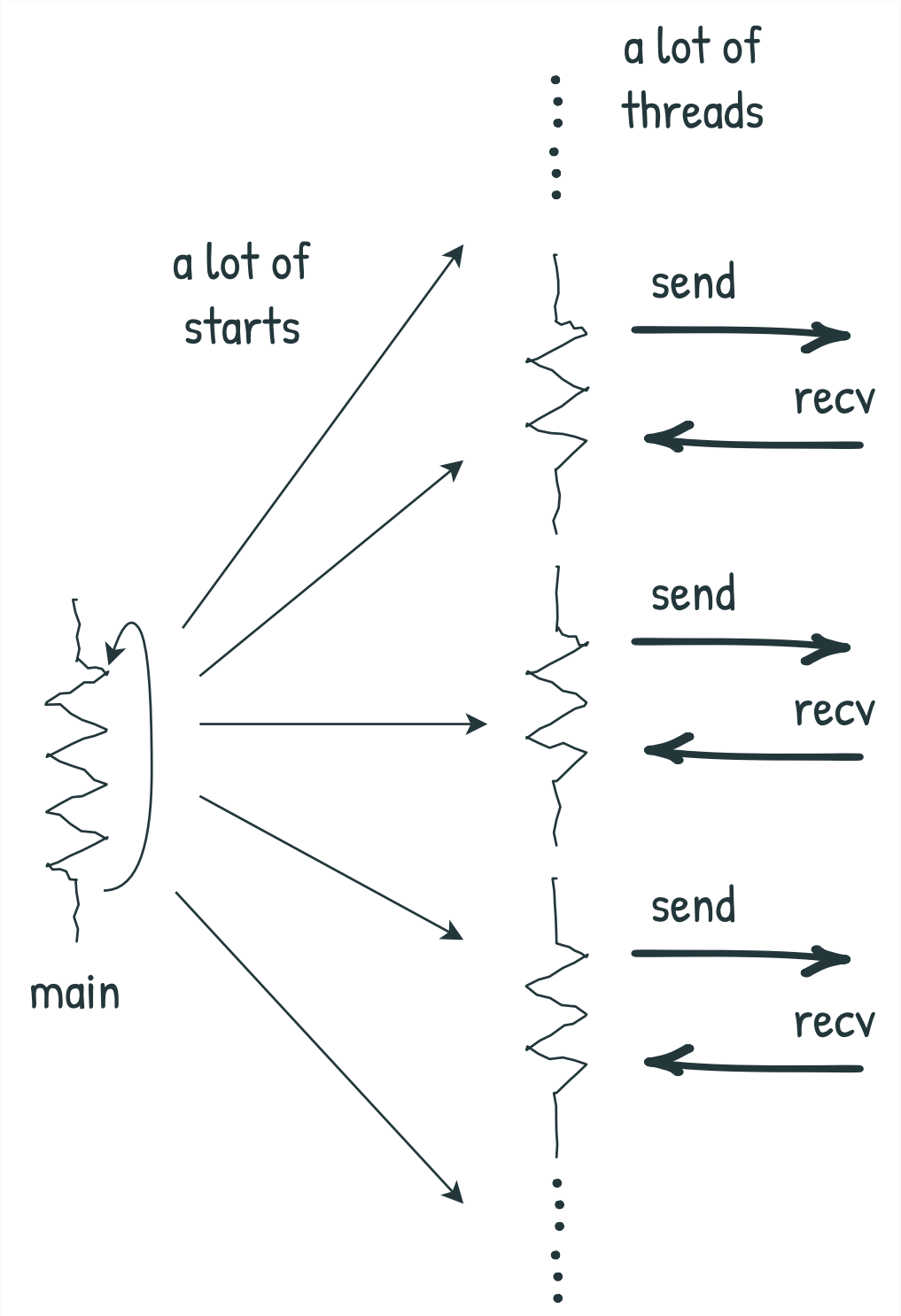
Y ahora?

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

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
}
```

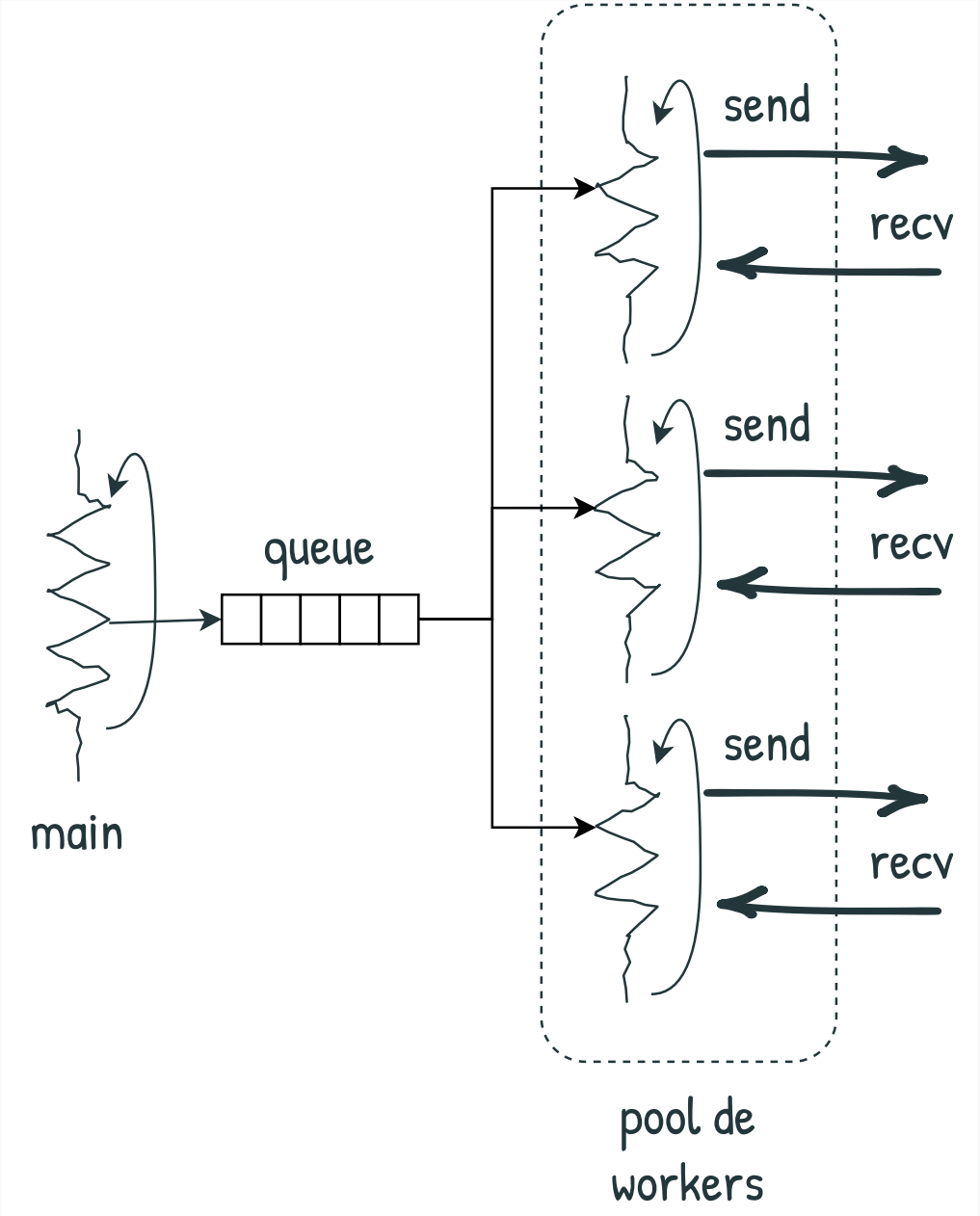
```
struct Fetcher: public Thread {
  Socket skt;
  void run() {
    fetch_web_page(this->url);
  }
}
```


a lot of
threads
a lot of
starts
send
recv
send
recv
send
recv
main

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
}
```

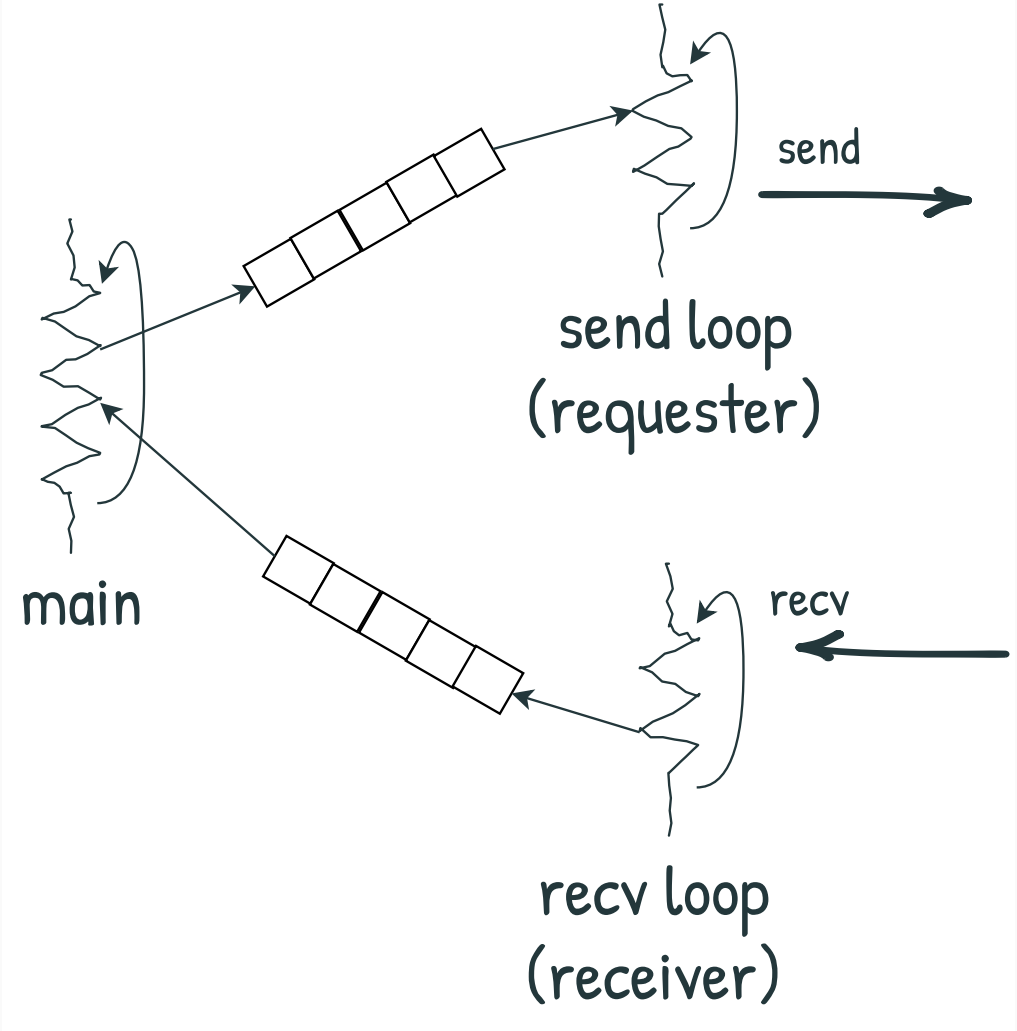
```
struct Fetcher:public Thread {
  Socket skt;
  Queue q;

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


send
recv
send
recv
send
recv
queue
main
pool de
workers

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);
    }
  }
```


send
send loop
(requester)
main
recv
recv loop
(receiver)

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);
    }
  }
```

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);
    }
  }
```

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);
    }
  }
```

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);
    }
  }
```

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);
    }
  }
```

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);
    }
  }
```

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);
  }
}
```

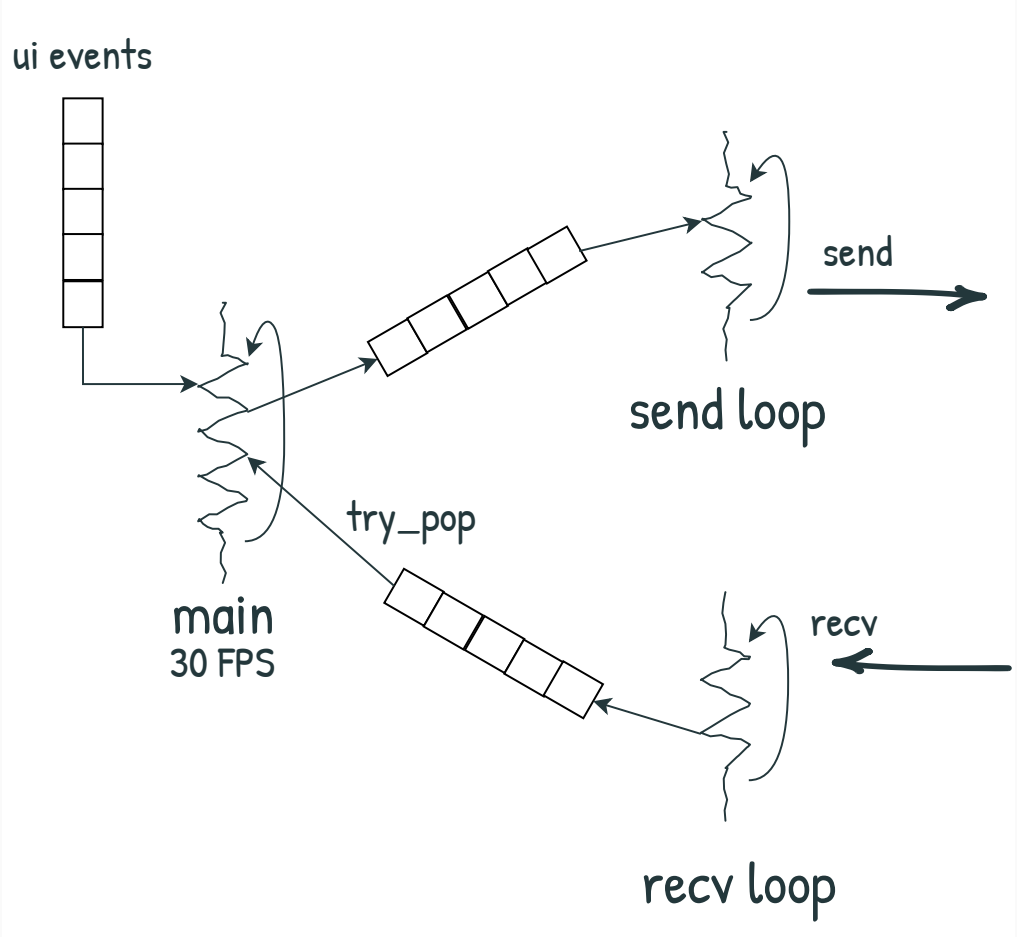
- Que operaciones son **bloqueantes**?
- Mientras el thread está bloqueado, que se podría hacer en el **mientras tanto**?

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);
  }
}
```

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
  }
}
```


ui events
send
send loop
try_pop
main
30 FPS
recv
recv loop

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?

Caso 10: singleclient server

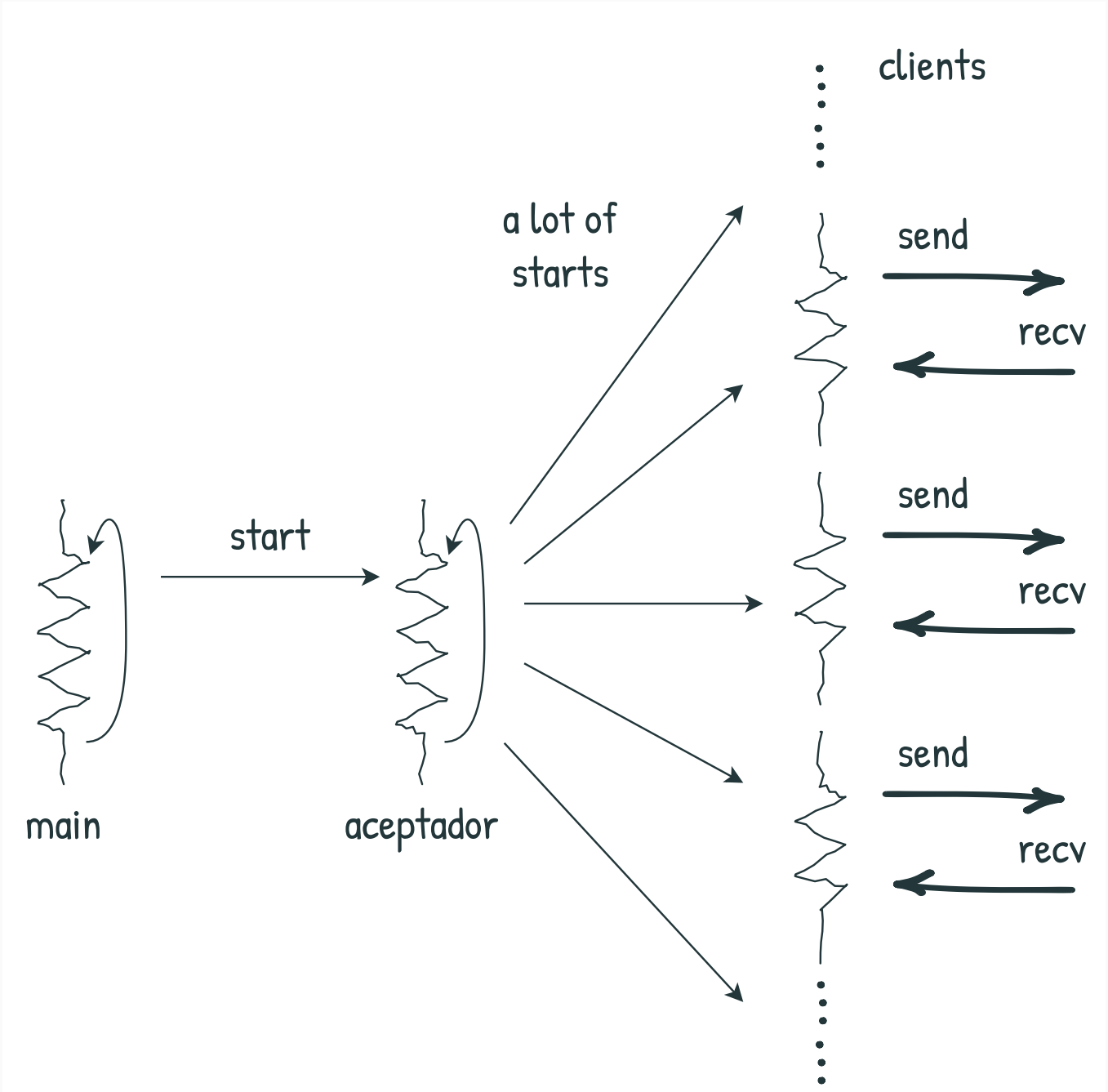
```
void main() {
  Socket sk; // socket aceptador
  acep_th = Aceptador(sk);
  acep_th.start()

  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()
  }
}
```

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);
  }
}
```


clients
a lot of
starts
send
recv
send
recv
send
recv
start
main
aceptador

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();
}
```

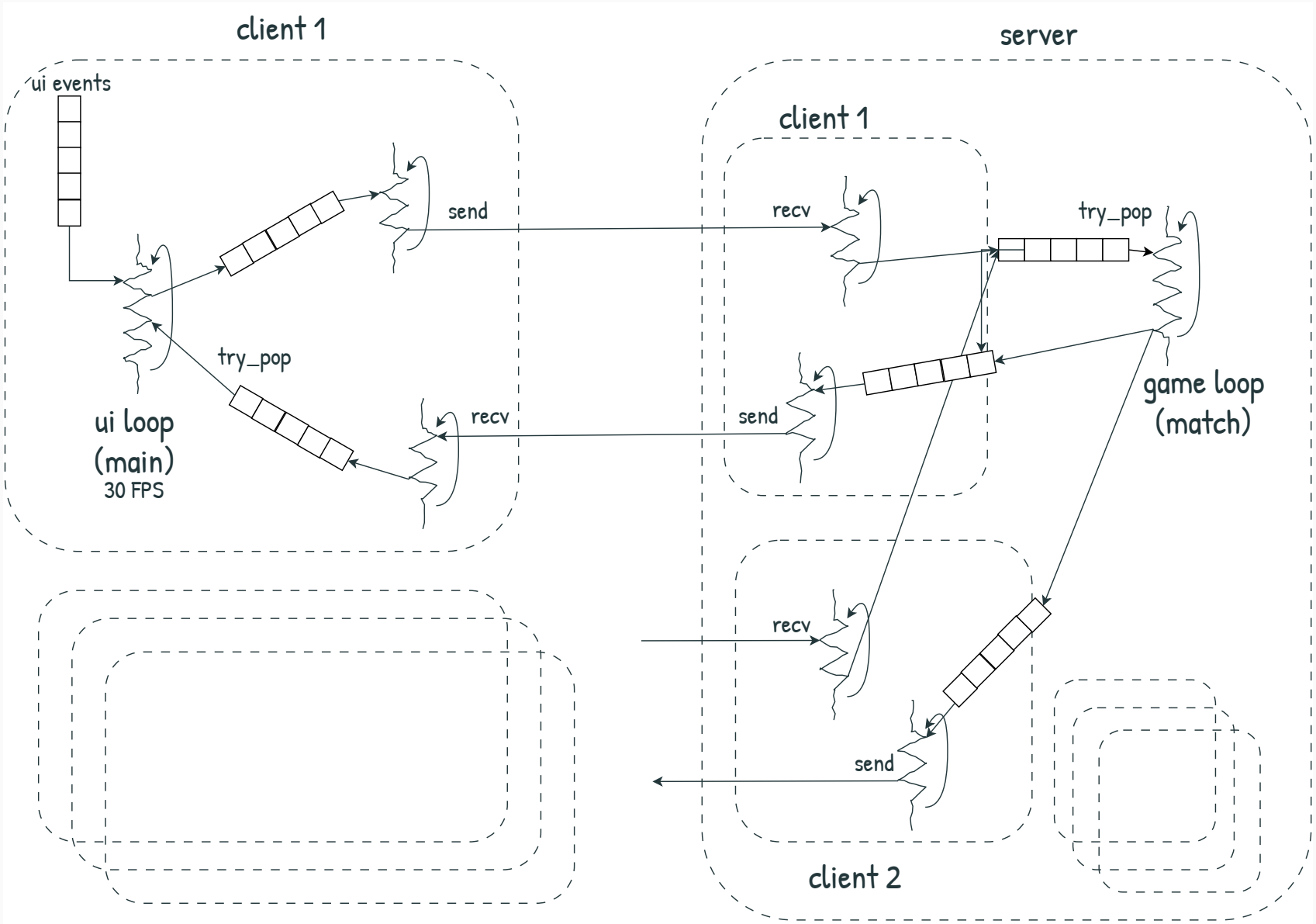
```
struct Aceptador:public Thread{
 Socket& sk;
 std::list<Client*> clients;

 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();
}
```

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;
}
```

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**.

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)

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**.

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)

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

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.

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)

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.

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).