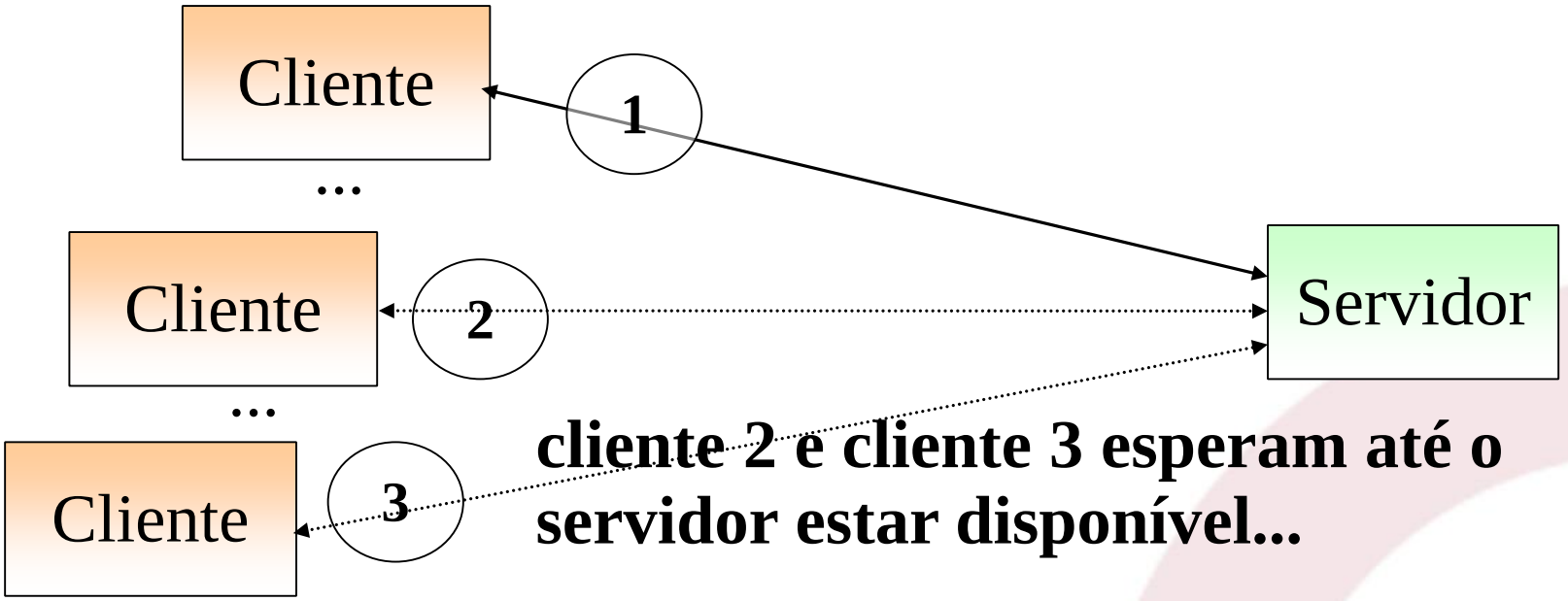
Sockets TCP

Autor: Patricio Domingues

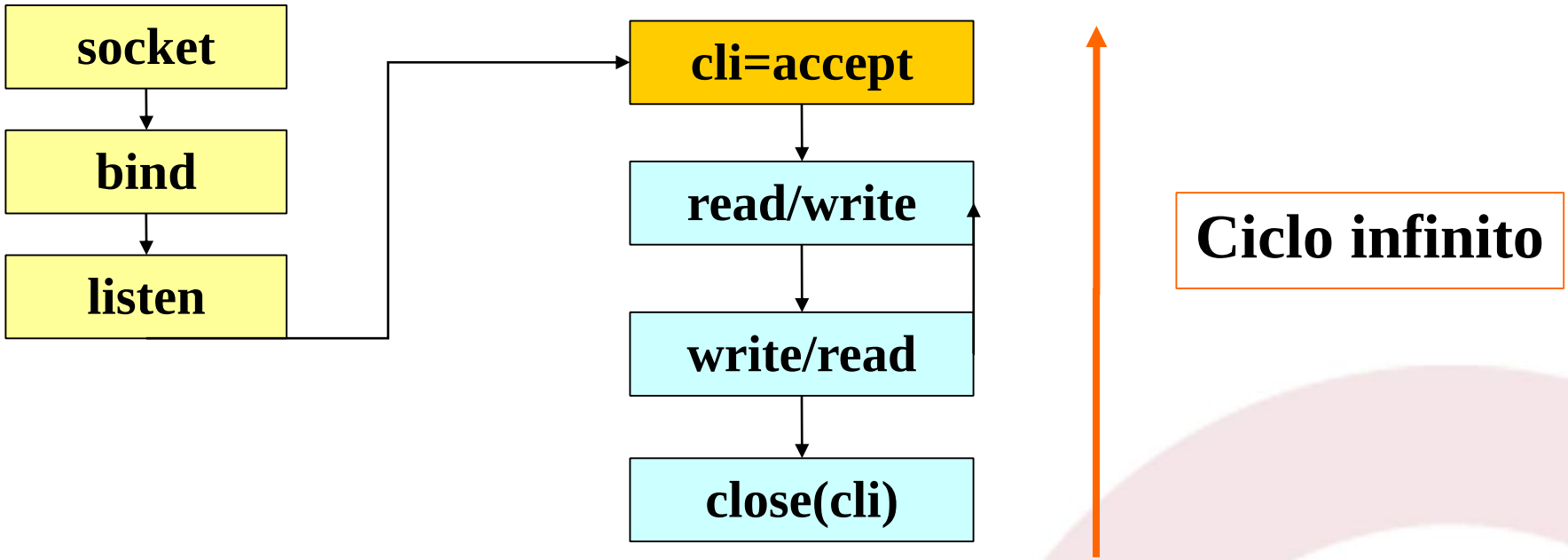
Servidor iterativo

- Um único processo servidor processa os pedidos de cada cliente
  - Entretanto, os outros clientes tem que esperar...
- Apropriado para operações de curta duração



Servidor gere os vários clientes sequencialmente

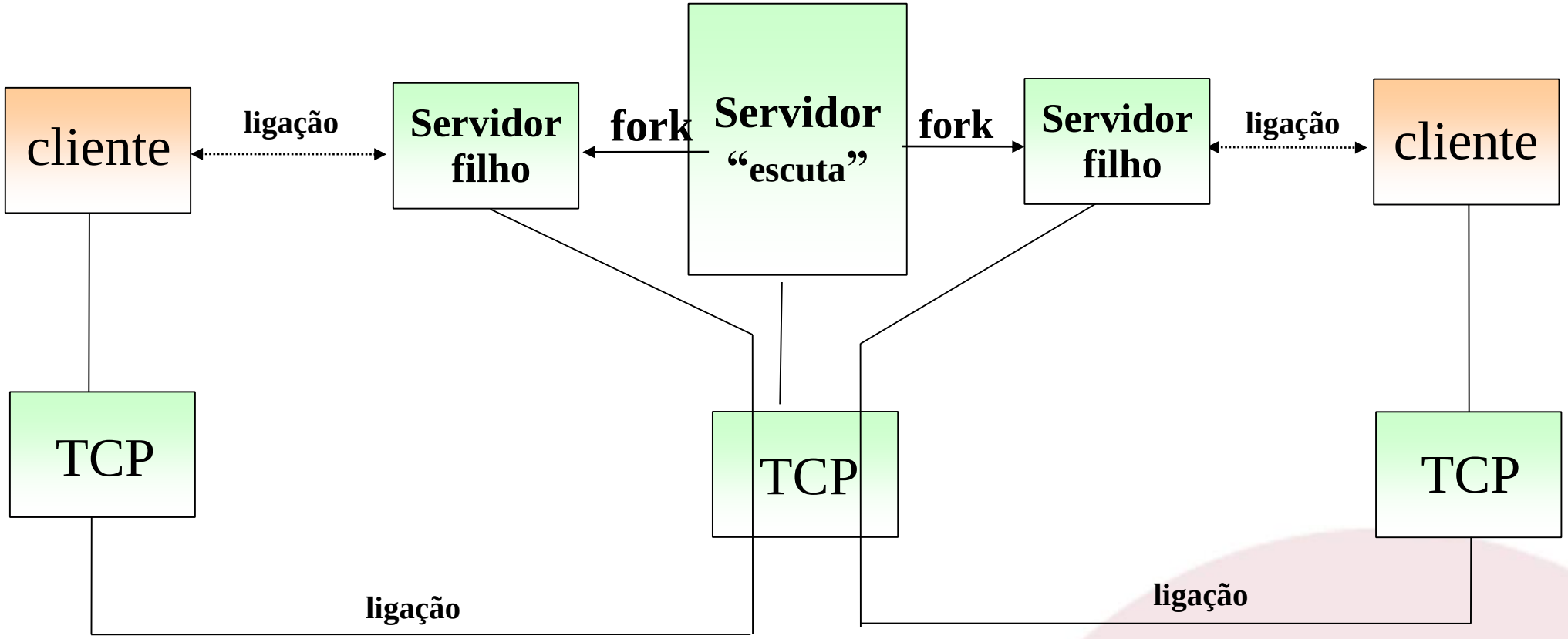
Servidor iterativo TCP



Servidor concorrente (#1)

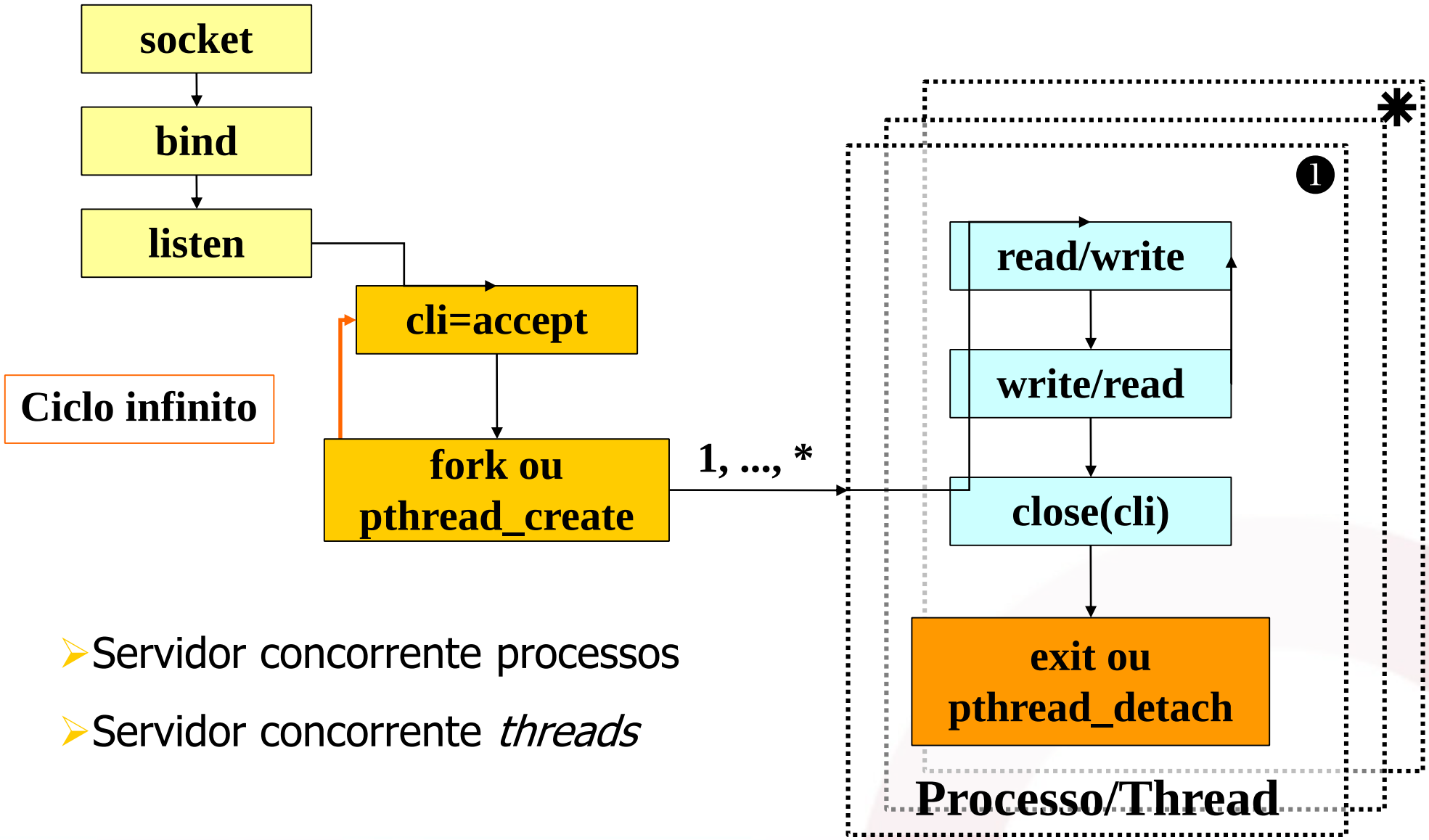
- Processo servidor recorre à criação de um processo (ou thread) por cada cliente
- `accept`
  - `fork`: processo filho trata do cliente
  - `pthread_create`: o *thread* trata do cliente
  - Processo pai (thread main) regressa ao `accept` para tratamento dos próximos pedidos de ligação
  - Vários clientes podem ser atendidos ao mesmo tempo
- Servidor concorrente
  - Apropriado para operações de média/longa duração
    - Exemplo
      - Sessões ssh, ftp, etc.

Servidor concorrente (#2)



Servidor TCP concorrente e dois clientes

Servidor concorrente TCP (1)



- Servidor concorrente processos
- Servidor concorrente *threads*

Servidor concorrente TCP (2)

- Código da parte concorrente

```
while(1){
clifd = accept(serverfd, (struct sockaddr *)&cliaddr, &addrlen);
      if (clifd == -1) {
/* Caso o accept tenha sido o interrompido, volta a efectuar o accept */
         if (errno == EINTR)
            continue;
         ERROR(-1,"Estabelecimento de ligacao invalido");
      } else if (addrlen != sizeof(cliaddr)) {
         WARNING("O endereco IP do cliente nao pertence ...: %s\n",
 get_protocol_family(cliaddr.sin_family));
         close(clifd); /* Fecha a ligacao com o cliente e liberta o recurso */
      } else {
         switch (fork()) {
            case 0:
               close(serverfd); /* Liberta os descritores desnecessarios */
               trata_cliente(clifd);
               exit(0);
            case -1:
               WARNING("Nao foi possivel criar um novo processo");
               close(clifd); /* Liberta os descritores desnecessarios */
            default:
               close(clifd); /* Liberta os descritores desnecessarios */
         }
      }
}
```

Híbridos: “prefork” e “prethread” (#1)

- A criação de um novo processo para cada cliente é uma operação cara do ponto de vista computacional

- Cria-se à priori (antes que sejam efetivamente precisos) um número predeterminado de processos (filhos), cada um encarregue de tratar um cliente
  - Cada processo filho atua com um servidor iterativo.

Híbridos: "prefork" e "prethread" (#2)

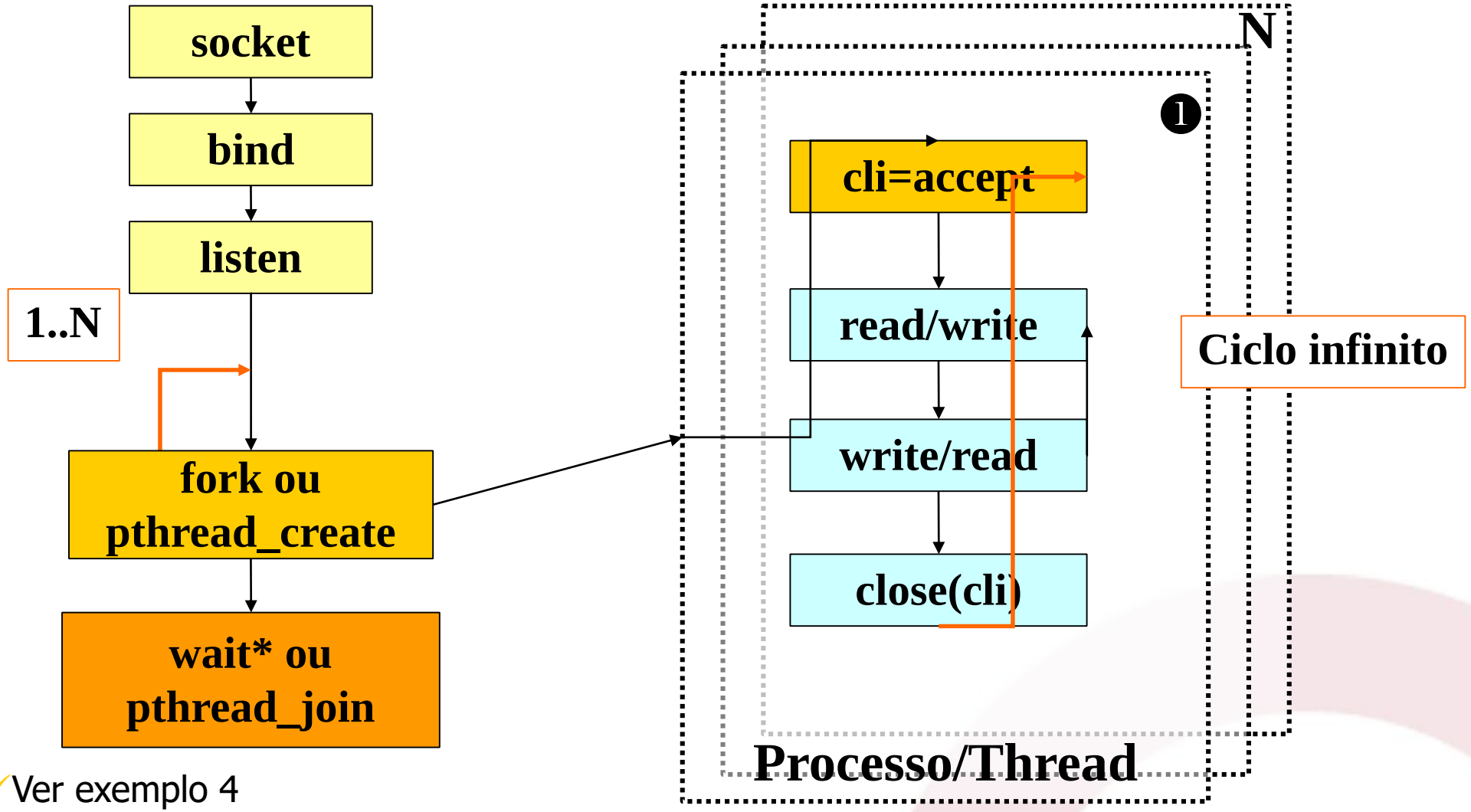
- Funcionamento
  - Processo inicial cria o socket de escuta e regista-se (`bind`) no sistema local (no porto do serviço).
  - Processo cria vários processos filhos, através da chamada ao sistema `fork`()
  - Cada processo filho chama o `accept`(), ficando bloqueado
  - O próximo pedido será tratado por um cliente (aquele no qual o sistema operativo activará a chamada `accept`)

- Servidores `prefork`
  - servidores híbridos que utilizam processos

- Servidores `prethread`
  - servidores híbridos que utilizam threads

Híbridos: processos/threads são criados antes de ocorrer a ligação com o cliente

Híbridos: "prefork" e "prethread" (#3)



- ✓Ver exemplo 4
  - Servidor híbrido processos
  - Servidor híbrido threads

Como escolher?

- Devo utilizar processos ou threads?
  - Threads caso o sistema operativo o permita
- Como devo fazer o meu servidor ?
  - Iterativo, concorrente, híbrido,…
  - Resposta
    - Depende…
    - Fatores a considerar
      - Número esperado de clientes simultâneos
      - Tamanho da transacção (tempo requerido para a computação)
      - Variabilidade no tamanho da transacção
      - Recursos dos sistema disponíveis

Funções read/write

- A função read devolve:
  - 0: caso a ligação tenha sido terminada corretamente
  - -1: em caso de erro (ter em conta a observação sobre os sinais)
  - > 0: quantidade de octetos lidos
- A função write devolve:
  - -1: em caso de erro (ter em conta a observação sobre os sinais)
    - Para além de devolver -1, e caso se trate de um erro de escrita, é enviado ao processo o sinal SIGPIPE. Se este não tratar o sinal, o processo termina.
  - <> -1: quantidade de octetos escritos
- Sempre que um processo recebe um sinal, as chamadas bloqueantes são interrompidas e devolvem -1
- Nestas chamadas estão incluídas as funções read e write
  - Se read ou write devolver -1 há que verificar se errno == EINTR e se sim, repetir a operação
- Exemplo

```
while(1){
        ret = read(sock,buffer,sizeof(buffer));
        if (ret == -1) {
           /* Caso o read tenha sido o interrompido, volta a efetuar a operação */
           if (errno == EINTR)
                    continue;
           ERROR(-1,"Estabelecimento de ligacao invalido");
        }
 }
```

Término de uma ligação (1)

- Os sockets TCP criam um canal de comunicação bidireccional
  - A função `close` fecha o canal nos dois sentidos
- Para fechar o canal apenas num dos sentidos, utiliza-se a função **shutdown**
  - Exemplo
    - Fecho do socket para escrita, mas esse ainda permanece aberto para leitura
  - `close` vs. `shutdown`
    - close
      - Decrementa o contador de referência do descritor, fechando-o se o valor for zero.
    - shutdown
      - Fecha apenas um sentido de comunicação – leitura ou escrita

Término de uma ligação (2)

```
#include<sys/socket.h>
    int shutdown(int sockfd, int howto);
    /* return : 0 if OK, -1 on error */
```

- Parâmetro *howto*
  - SHUT_RD
    - Fecho do sentido de leitura (processo pode escrever mas não ler)
  - SHUT_WR
    - Fecho do sentido de escrita (processo pode ler mas não escrever)
  - SHUT_RDWR
    - Fecho de ambos os sentidos
    - Equivalente ao close quando o contador de descritores é 1

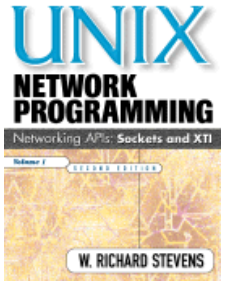
Exemplos – sockets TCP

- Ficheiros
  - cliente.c
  - servidor_concorrente_processos.c
  - servidor_concorrente_threads.c
  - servidor_hibrido_processos.c
  - servidor_hibrido_threads.c
  - servidor_iterativo.c

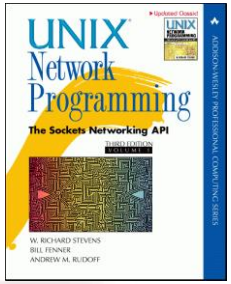
- https://tinyurl.com/ycfohxz5

Bibliografia

- Leitura recomendada
  - *UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI*, Prentice Hall, 1998, ISBN 0-13-490012-X.
    - Capítulo 27



  - *Unix Network Programming: The Sockets Networking API*, Volume 1, 3rd edition, 2003, 1024 pages, Addison-Wesley.ISBN: 0-13-141155-1



  - *Beej's Guide to Network Programming - Using Internet Sockets*, Brian "Beej Jorgensen" Hall, 2016 (http://beej.us/guide/bgnet/)

  - `man 7 socket`
  - `man 7 tcp`