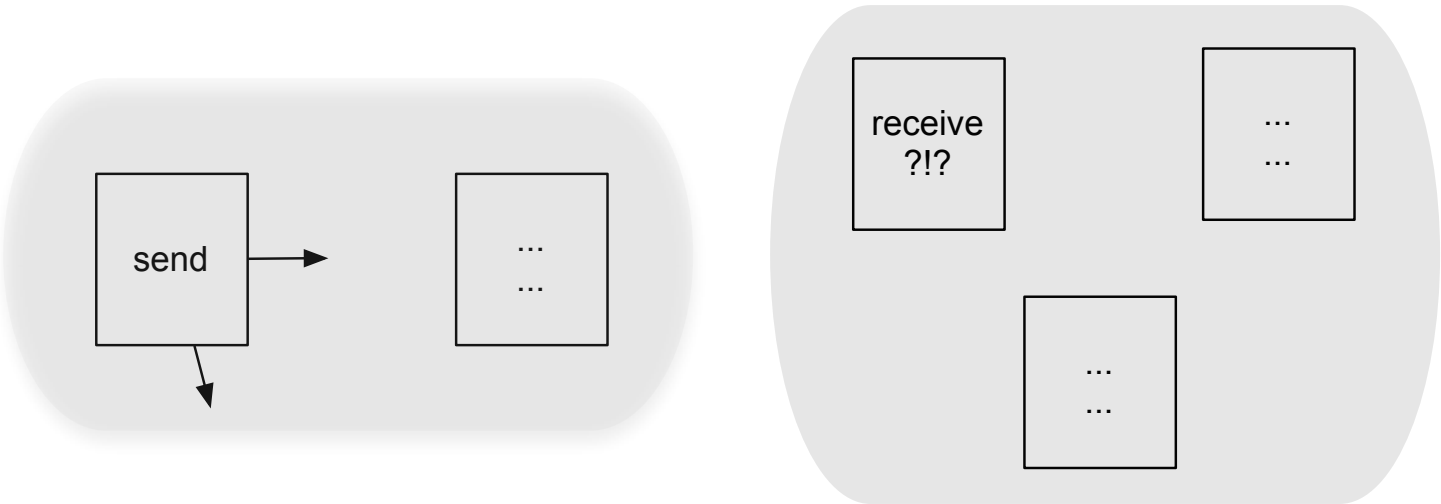
Sistemas Distribuídos

Comunicação e Coordenação – Clientes e Servidores

março de 2015

Comunicação entre Processos Distribuídos

- troca de mensagens exige coordenação



Coordenação

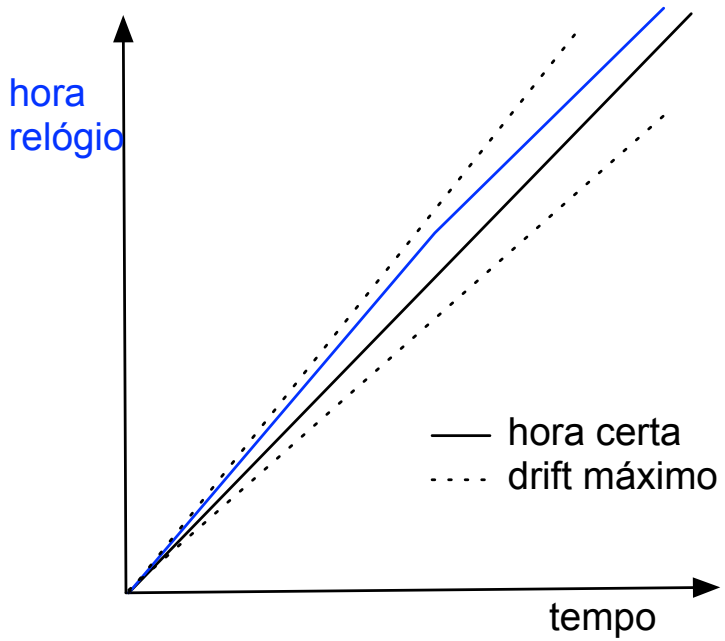
- passos para execução de ações de processos diferentes contribuindo para um objetivo comum
  - muitas vezes, necessidade de *sincronização*

Tempo em sistemas distribuídos

- hora certa é conceito extremamente útil para sincronizar atividades no "mundo real"
- em sistemas computacionais:
  - ordenação
  - depuração
  - escalonamento de atividades
  - medidas
  - ...
- necessidade de uma hora certa *global*

Relógios físicos

- cada máquina tipicamente tem um relógio físico
  - circuito que gera interrupções a cada *n* oscilações
- relógios físicos exibem uma taxa de variação em relação a hora real



Sincronização de relógios físicos

interna

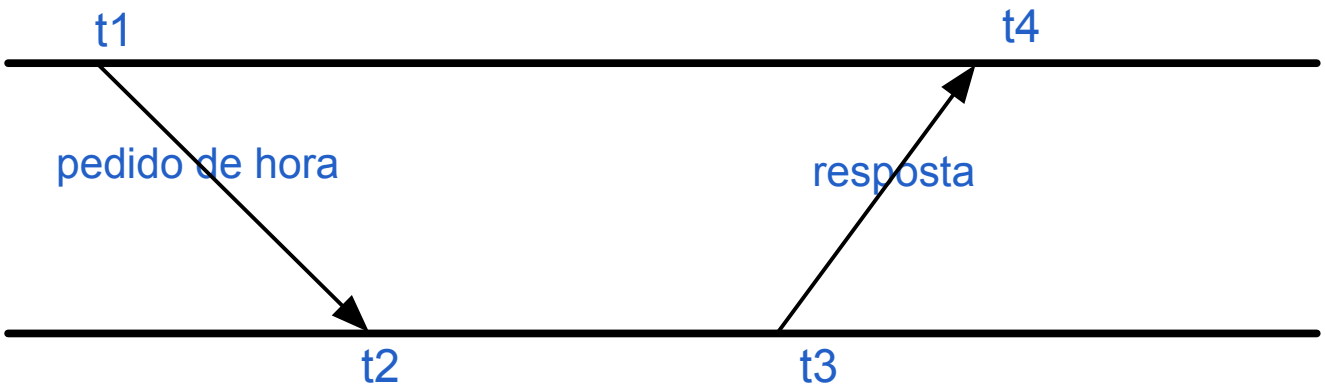
- diferença máxima entre horas de dois relógios do sistema

externa

- diferença máxima entre horas de qualquer relógio do sistema e uma fonte externa

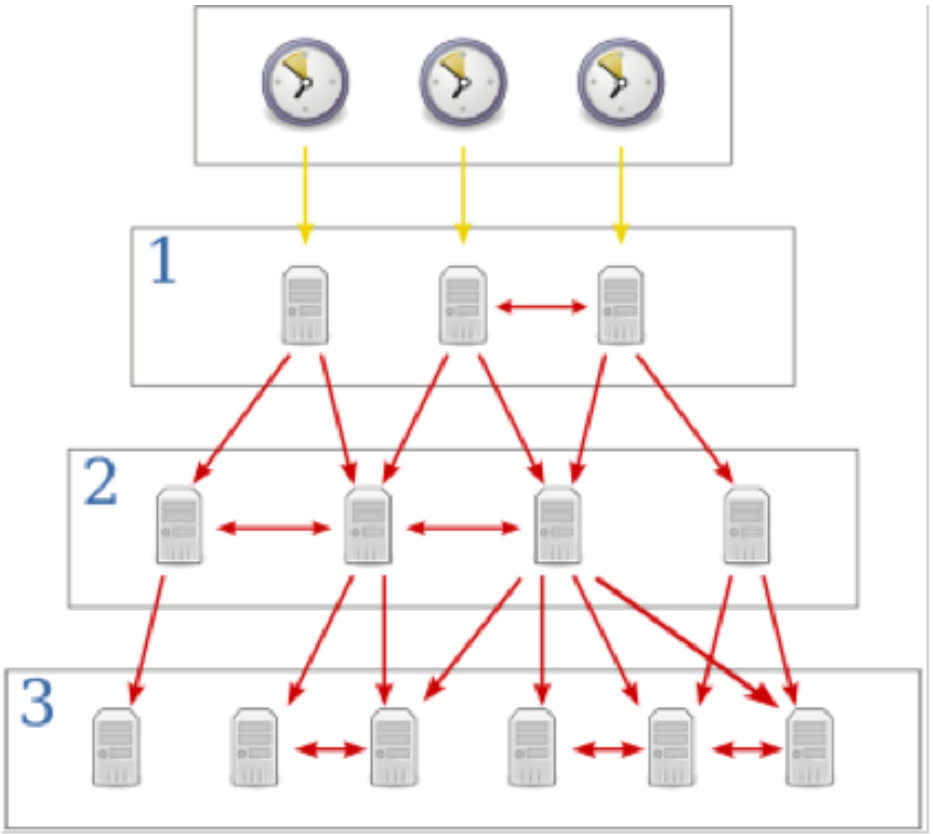
Algoritmos de sincronização de relógio

- trocas de mensagens repetidas para obtenção de diversas amostras
- protocolo *NTP* atualmente o mais difundido



NTP – Network Time Protocol

- uso de técnicas estatísticas para aproximar relógios físicos da hora correta



Sincronização de relógios físicos

motivação:

- sistemas de arquivos
- autenticação
- auditoria
- segurança

Sincronização na Comunicação

- uso de *padrões* de comunicação
  - cliente-servidor
  - filtros
  - p2p
    - simetria e padrões de algoritmos
    - replicação de código / SPMD

relação entre primitiva de comunicação e padrão usado na aplicação

Padrão cliente-servidor

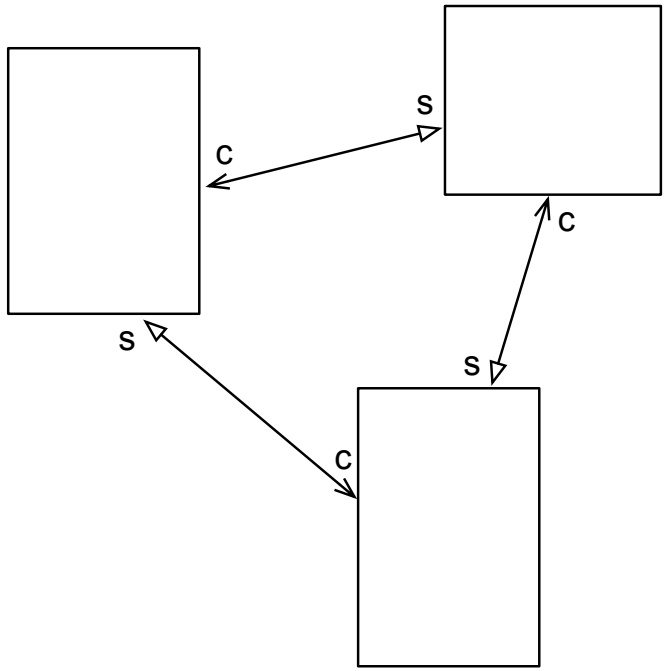
- servidor sempre a espera de comunicação
  - em programas sequenciais: bloqueado
- cliente sempre inicia a comunicação
  - assimetria: cliente deve saber localizar servidor
  - envio de requisição associado a um ou mais recebimentos

pseudo-servidor

```
while (true) do {
  aguarda requisição de qualquer cliente C
  processa requisição
  responde a cliente C
}
```

Padrão cliente-servidor

- nomenclatura faz sentido dentro de uma interação

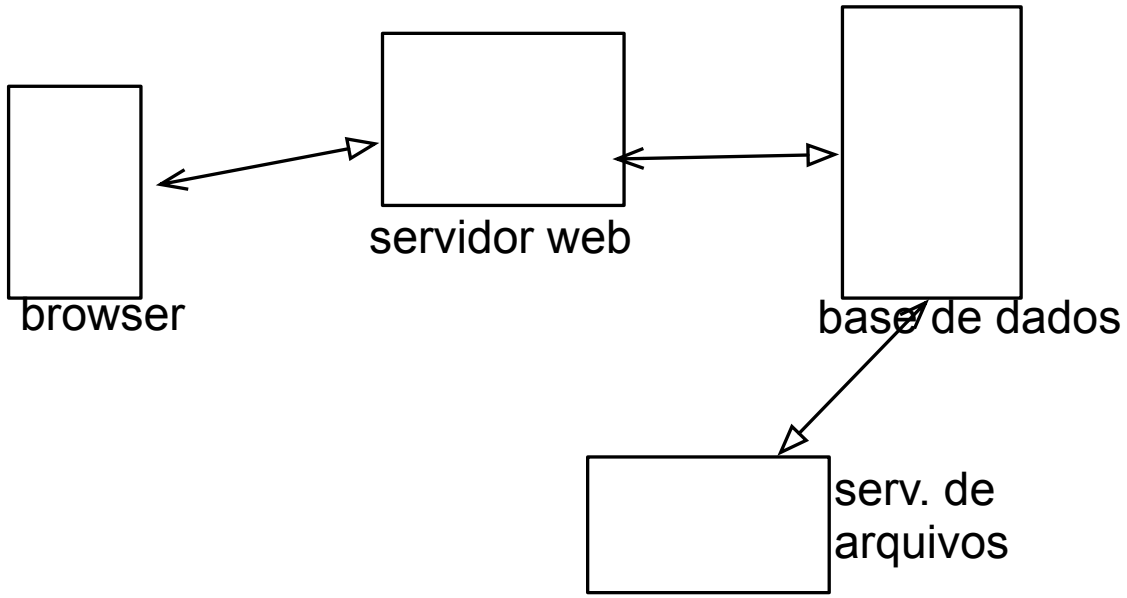


padrão de comunicação mais difundido

- servidores clássicos (BD, autenticação, hora certa, ...)
- uso genérico do padrão

Arquiteturas de n camadas

- hierarquia de clientes e servidores
  - bem difundidas em aplicações web



Modelos de servidores

- servidores iterativos: processamento (completo) de um pedido por vez
- servidores concorrentes: simultâneo a vários clientes
  - multitarefa
    - multiprocesso
    - multithread
  - monotarefa
    - atendimento a vários clientes "misturado" no código
    - usado em sistemas (1) com interação entre clientes e (2) com recursos muito limitados

Servidores iterativos

- em programas sequenciais: estrutura do "pseudo-servidor" já garante um atendimento por vez

```
while (true) do {
  aguarda requisição de qualquer cliente C
  processa requisição
  responde a cliente C
}
```

Servidores iterativos

- em programas sequenciais: estrutura do "pseudo-servidor" já garante um atendimento por vez

```
-- load namespace
local socket = require("socket")
-- create a TCP socket and bind it to the local host, at any port
local server = assert(socket.bind("*", 0))
-- find out which port the OS chose for us
local ip, port = server:getsockname()
-- print a message informing what's up
print("Please telnet to localhost on port " .. port)
print("After connecting, you have 10s to enter a line to be echoed")
-- loop forever waiting for clients
while 1 do
  -- wait for a connection from any client
  local client = server:accept()
  -- make sure we don't block waiting for this client's line
  client:settimeout(10)
  -- receive the line
  local line, err = client:receive()
  -- if there was no error, send it back to the client
  if not err then client:send(line .. "\n") end
  -- done with client, close the object
  client:close()
end
```

alocador com rendezvous

```
module Allocator
  op request (), release();
body
  process Alloc {
    bool free = true;
    while (true) {
      in request (prio)  -> free = false; ni
      in release () -> free = true; ni;
    }
  }
end Allocator
```

- notação e exemplo inspirados em *Multithreaded, Parallel, and Distributed Programming, Greg Andrews*.

alocador com rendezvous

- ou, dependendo do sistema de comunicação:

```
module Allocator
  op request (int prio), release();
body
  process Alloc {
    bool free = true;
    while (true)
      in request (prio)  and free by prio -> free = false;
      [] release () -> free = true
      ni
  }
end Allocator
```

Servidores iterativos (2)

- em programas orientados a eventos:
  - estado deve registrar servidor como ocupado
  - servidor pode ou não avisar cliente de seu estado

Servidores concorrentes multitarefa

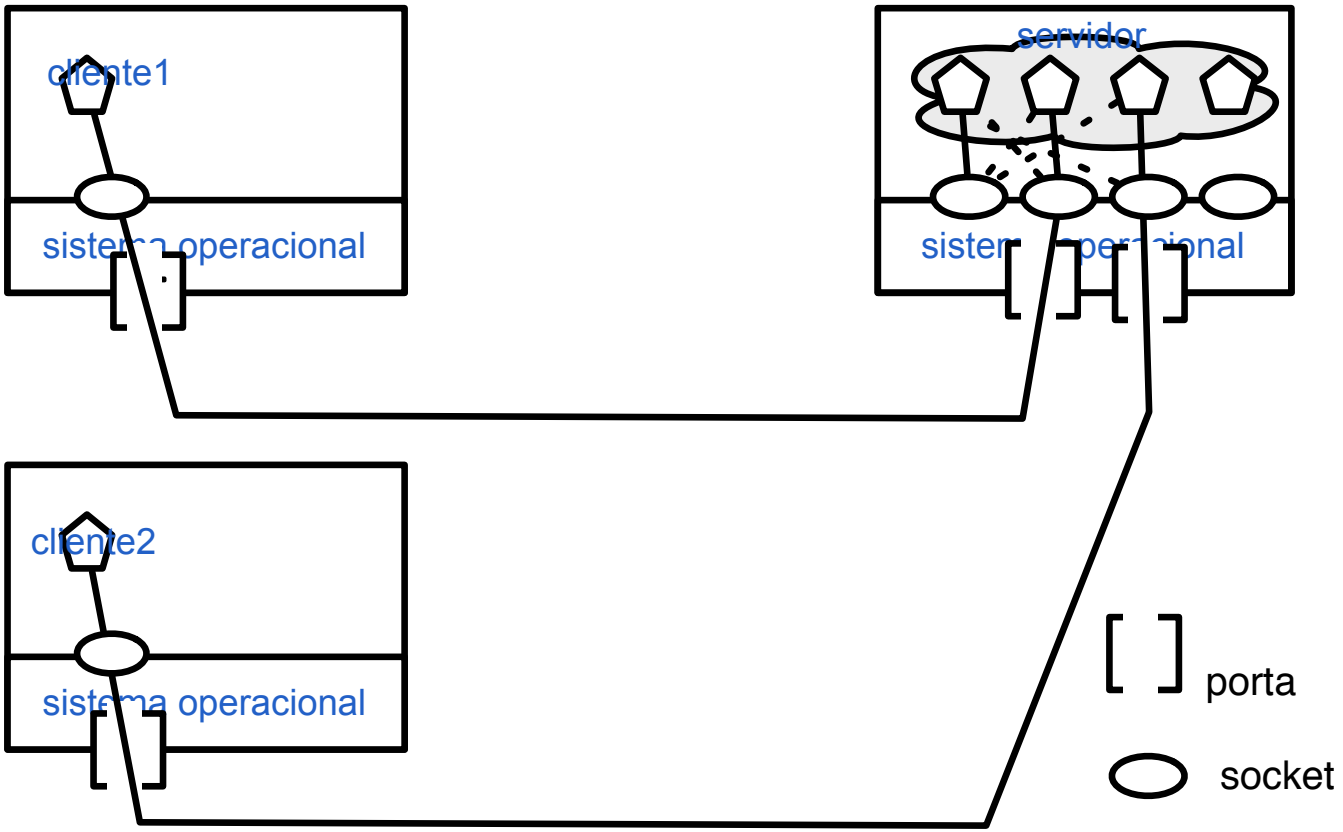
- threads x processos
- escalabilidade

Servidores multitarefa sob demanda

- disparo de tarefas *sob demanda*, isto é, quando chega uma requisição
- uma tarefa para interação com cada cliente

- elasticidade no atendimento
- sobrecarga de criação e destruição de tarefas

Sockets: multitarefa sob demanda

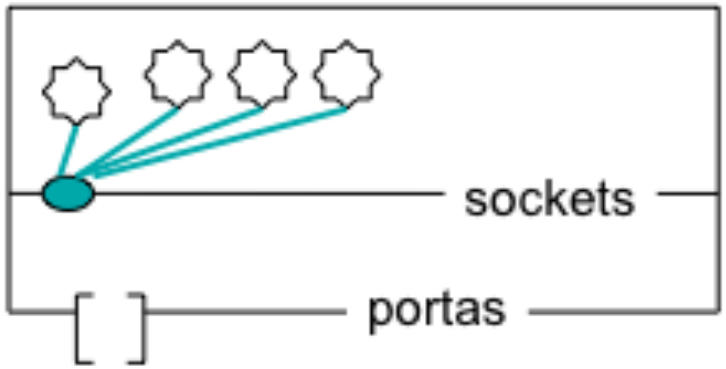


Servidores multitarefa com pré-alocação

- unidades de execução pré-existentes (pool de tarefas) prontas para atender requisições
  - modelo iterativo em cada tarefa
  - outras estruturas mais complexas

- atendimento imediato se tarefas disponíveis
- maior dificuldade no ajuste do tamanho do *pool*

Sockets: multtarefa com pré-alocação

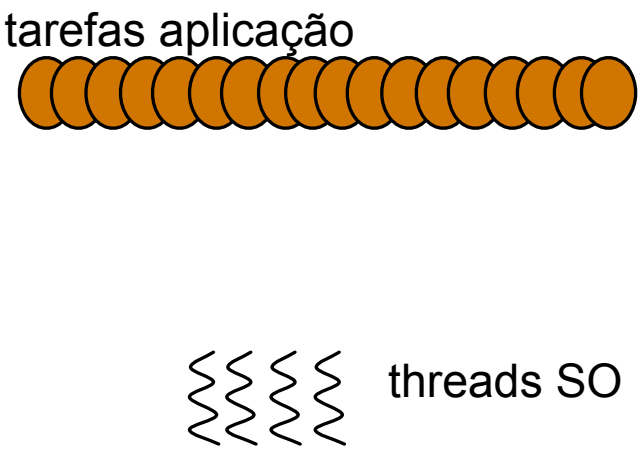


Exemplo Andrews: servidor de arquivos

```
type kind = enum(READ, WRITE, CLOSE)
chan open(fname : string[*], clientid : int)
chan access[1:n](kind, other types)      # other types give buffer, number of bytes, etc.
chan open_reply[1:n](int)      # field is server index or error indication
chan access_reply[1:n](result types)      # result types are file data, error flags, etc.
File_Server[i: 1..n]:: var fname : string[*], clientid : int
                       var k : kind, args : other argument types
                       var file_open : bool := false
                       var local buffer, cache, disk address, etc.
                       do true →
                            receive open(fname, clientid)
                            # open data file; if successful then:
                            send open_reply[clientid](i); file_open := true
                            do file_open →
                                 receive access[i](k, args)
                                 if k = READ → process read request
                                 [] k = WRITE → process write request
                                 [] k = CLOSE → close file; file_open := false
                                 fi
                                 send access_reply[clientid](result values)
                            od
                       od
Client[j:1..m]:: ...
                 send open("foo", j)      # open file "foo"
                 receive open_reply[j](serverid)      # get back id of server
                 # use and eventually close file by executing the following
                 send access[serverid](access arguments)
                 receive access_reply[j](results)
                 ...
```

Arquiteturas com pools de threads

- uso de tarefas no nível de aplicação e threads de sistema operacional
  - escalonamento cooperativo



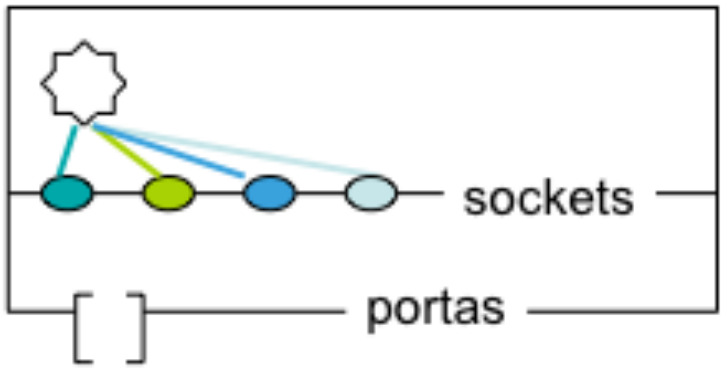
Servidores Monotarefa

- integração natural com modelo de eventos
  - "próxima interação pode ser com qualquer um"
- caso simples: atendimento completamente contido em um tratador de evento

```
event <Request | clid, rqid, rq>
  res = handle (rq);
  trigger <Reply | clid, rqid, res>
end
```

- caso geral: atendimento requer "diálogo" entre cliente e servidor
  - estado do diálogo deve ser mantido em variáveis globais

Sockets: monotarefa



Servidores monotarefas — exemplo (bobo) com 2 clientes de uso de *select*

```
local client1 = server:accept()
client1:settimeout(0); client1:setoption('tcp-nodelay', true)
local client2 = server:accept()
client2:settimeout(0); client2:setoption('tcp-nodelay', true)
local obs = {}
table.insert(obs,client1); table.insert(obs,client2)
while 1 do
  local clst1, clst2, err = socket.select(obs, {}, 1)
  for _, clt in ipairs(clst1) do
    local line, err = clt:receive('*l'))
    if line then
      clt:send(line)
    else
      print("ERRO: ".. err)
    end
  end
end
```

Servidores monotarefas — exemplo (bobo) com 2 clientes de uso de *select*

```
local client1 = server:accept()
local client2 = server:accept()
...
while 1 do
  local clst1, clst2, err = socket.select(obs, {}, 1)
  for _, clt in ipairs(clst1) do
    local line = ''
    local newpart, err, finalpart = clt:receive(64)
    while (not err) do
      line = line .. newpart
      newpart, err, finalpart = clt:receive(64)
    end
    if (err == "timeout") then
      if newpart then line = line .. newpart end
      line = line .. finalpart
      clt:send(line)
    else
      print("ERRO: ".. err)
    end
```

Servidores com e sem estado

- servidor pode manter infos sobre clientes:
  - dentro de uma "requisição"
  - entre requisições

simplicidade X desempenho

Clientes magros e gordos

- controle da quantidade de processamento realizada por clientes e servidores
  - condições de processamento e de comunicação
  - variação dinâmica de comportamento