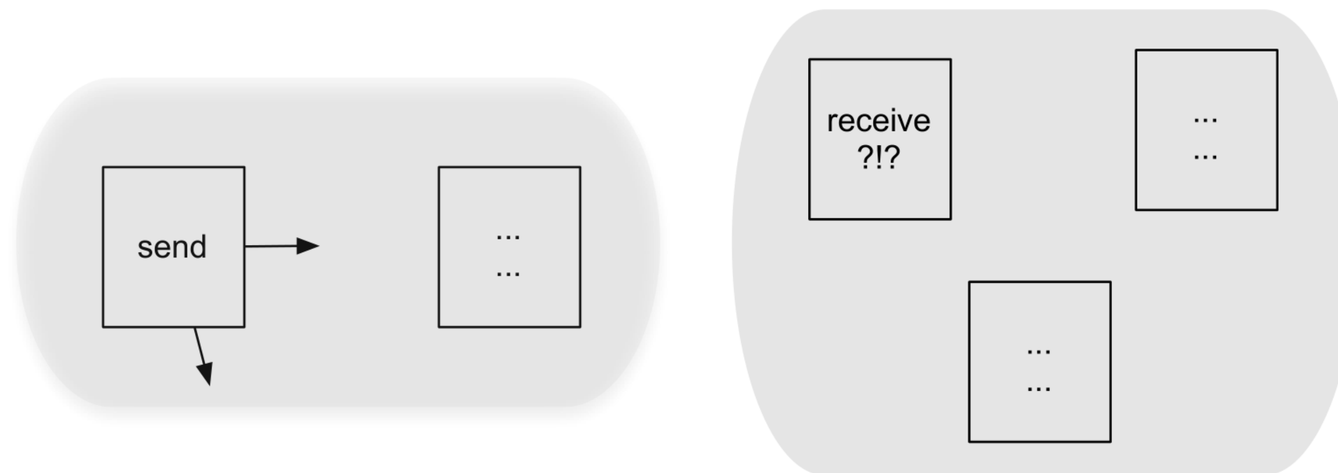
# Sistemas Distribuídos

## Comunicação e Coordenação – Clientes e Servidores

março de 2017

# Comunicação entre Processos Distribuídos

- troca de mensagens exige coordenação



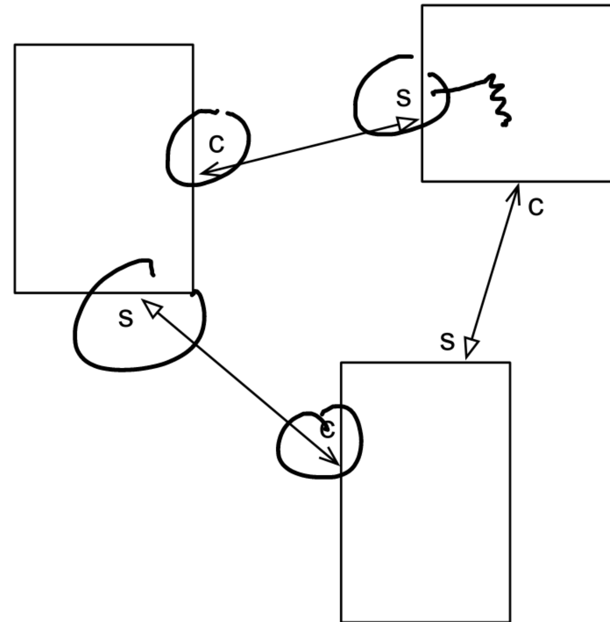
# Padrão cliente-servidor

- servidor sempre a espera de comunicação
  - em programas sequenciais: bloqueado
- cliente sempre inicia a comunicação
  - assimetria: cliente deve saber localizar servidor endereço
  - 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



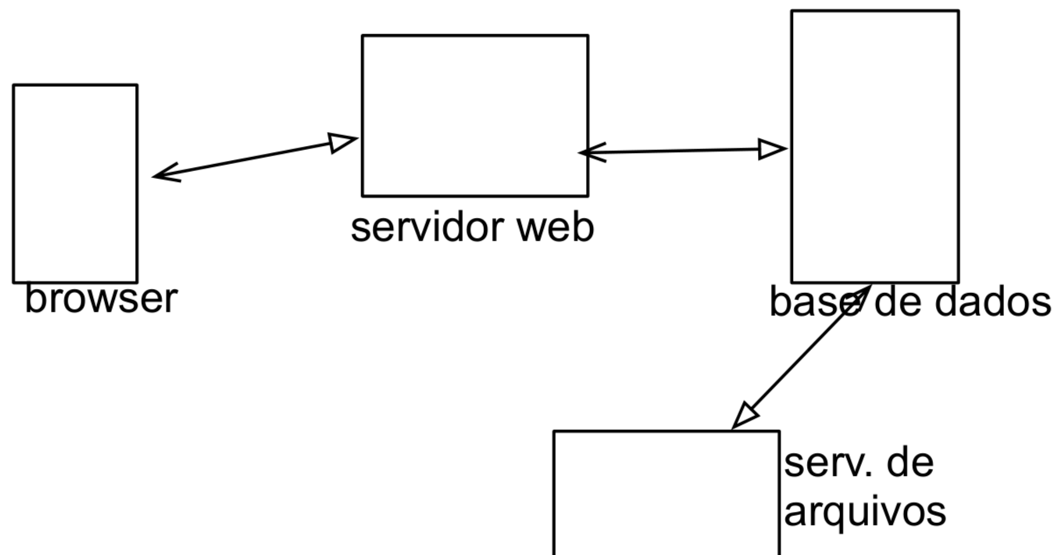

- nomenclatura faz sentido dentro de uma interação

## 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




N-tiers
N camadas

# Modelos de servidores

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

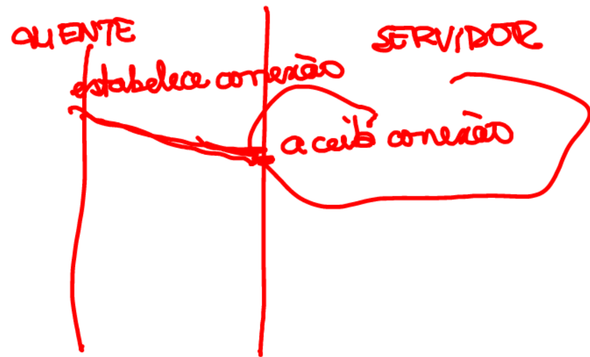
# Servidores iterativos

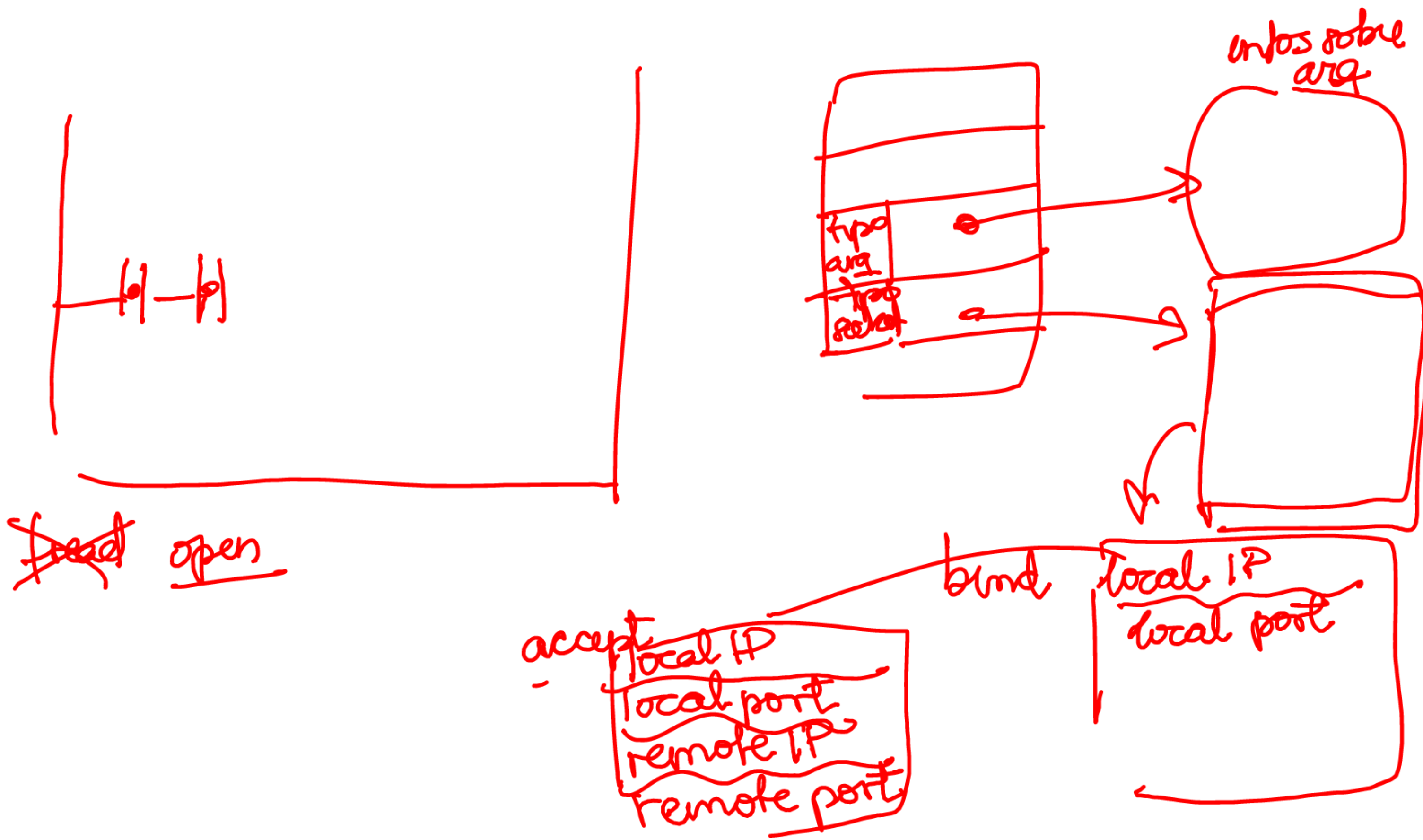
- em programas sequenciais: estrutura do “pseudo-servidor” já garante um atendimento por vez

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

TCP, UDP, IP, outros protocolos
UNIX domain protocol, .

embute o modelo cliente servidor





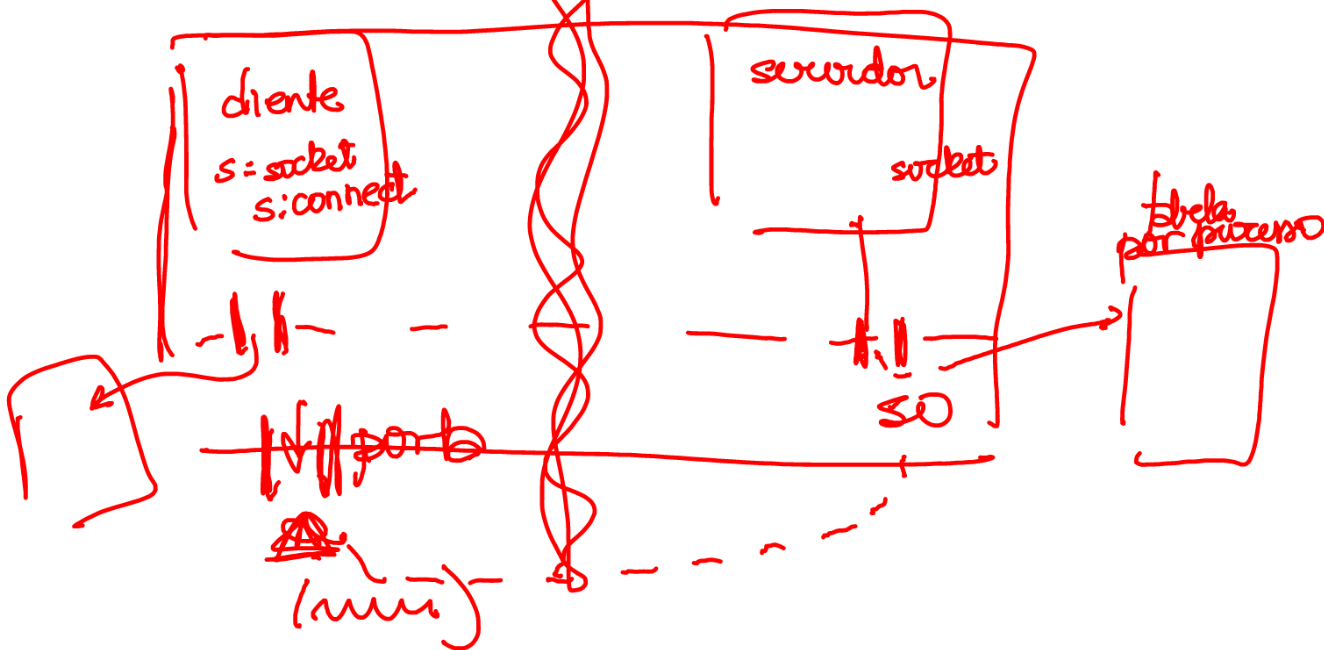
open
infos sobre arq
tipo arq
tipo socket
bind
local IP
local port
accept
local IP
local port
remote IP
remote port

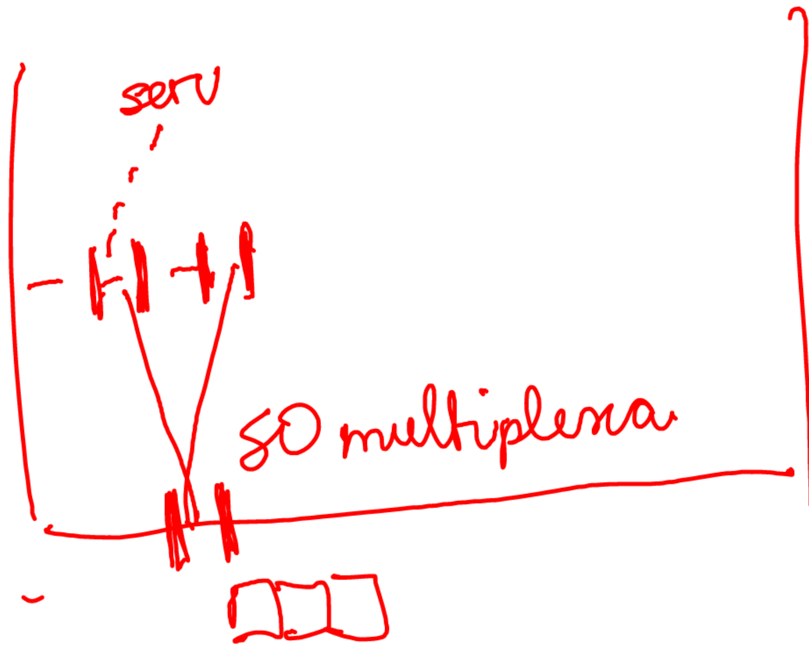
interface C

s = socket()

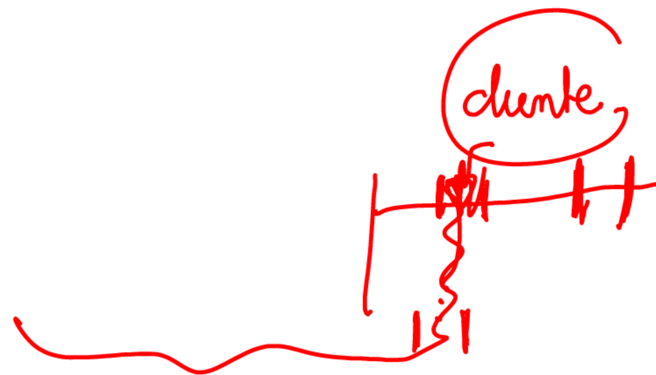
bind (s, IP + porta)

listen (s, ~~~ ) /* avisa ao SO que este socket é "servidor" */






cli1 = accept



# Servidores iterativos – Lua e luasocket

luasocket

```
-- 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
```

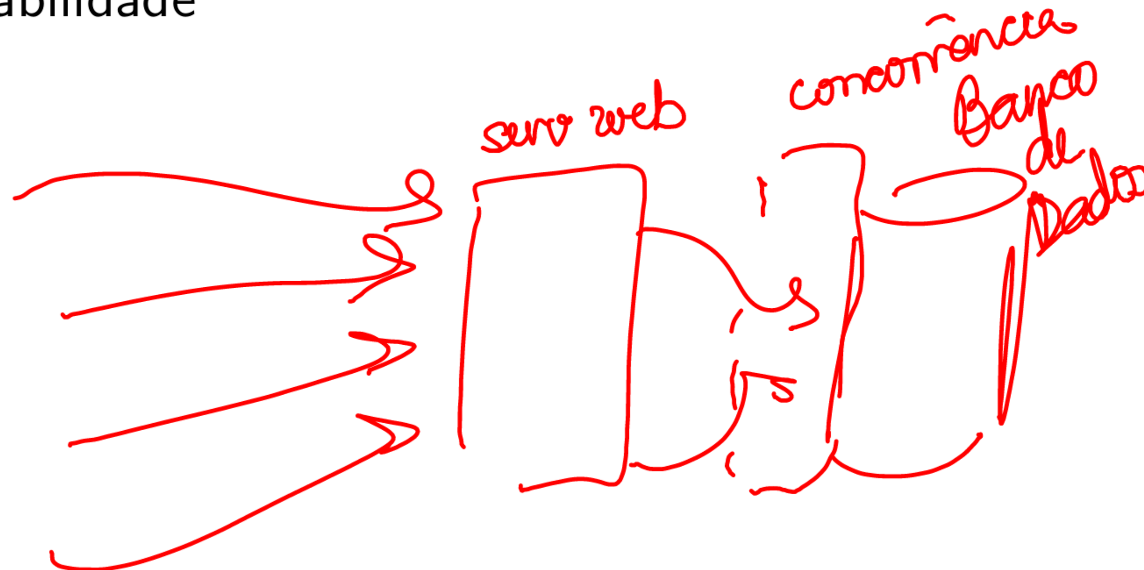
# Servidores iterativos com eventos

- como postergar tratamento de novo cliente?
  1. servidor só volta a tratar eventos depois de tratamento do cliente anterior totalmente finalizado
  2. servidor registra estado ocupado e pode responder a outros clientes comunicando estado

# Servidores concorrentes multitarefa

threads de SO / threads de aplicação

- threads x processos
- escalabilidade



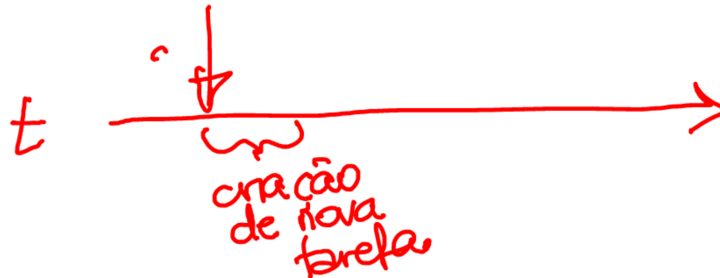
multitarefa com processos UNIX

```
cli = server:accept()
if ((pid = fork()) == 0) {
  /* filho */
  server:close()
  trata cli;  --> cli:close()
  exit();
}
else /* pai */
  cli:close()
.
```

com threads (pthreads)

```
*void trataCliente (*void arg) {
    trata cliente
    fecha socket cliente
} /* morre thread */
```

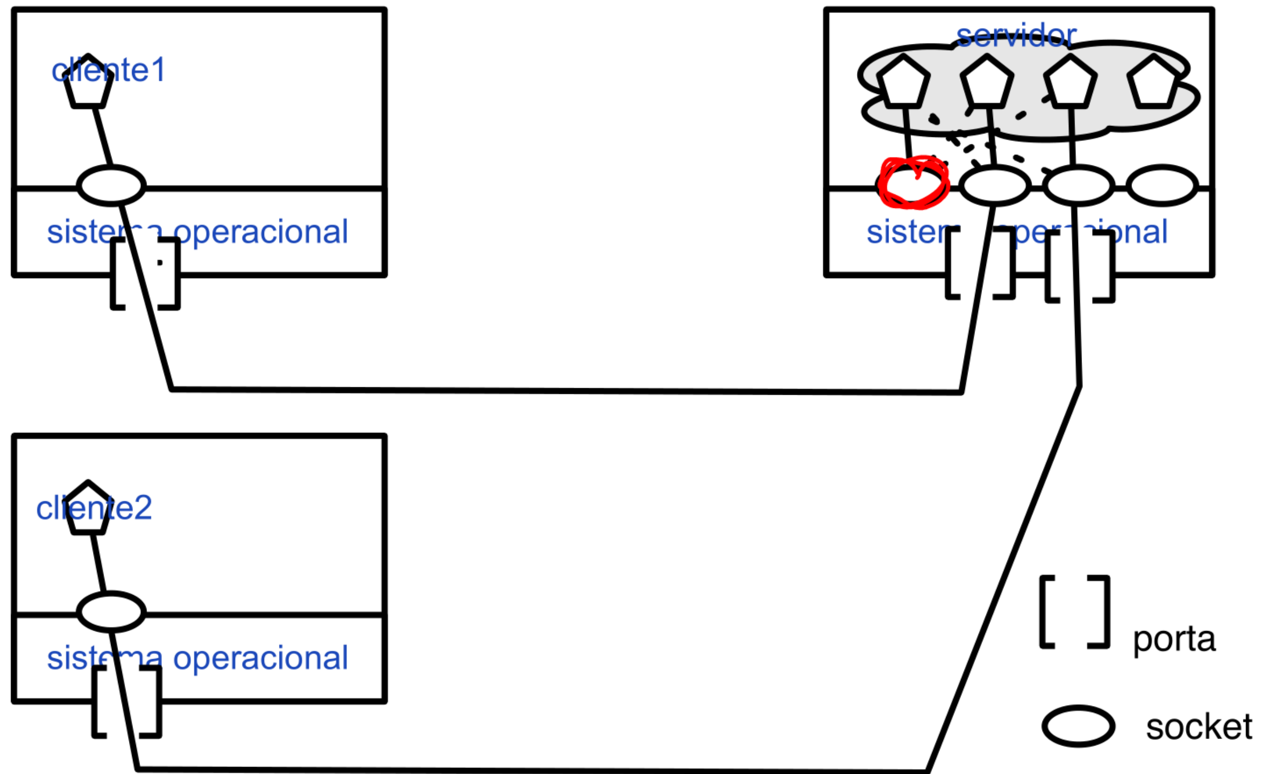
# Servidores multitarefa sob demanda




- 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



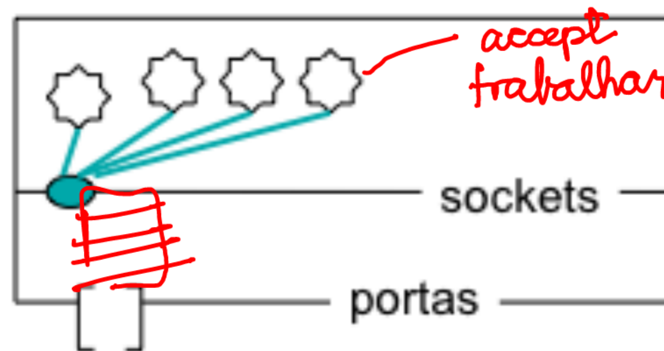
# Servidores multitarefa com pré-alocação

- 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: multitarefa com pré-alocação

server = socket ( ) . . .
:
fork



# Exemplo Andrews: servidor de arquivos

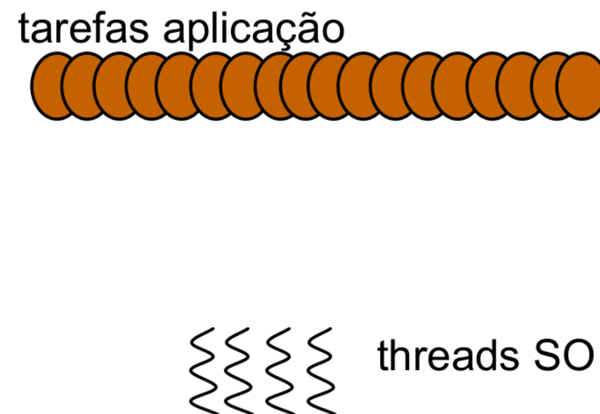
```
type kind = enum(READ, WRITE, CLOSE)
chan open(fname : string[*], clientid : int)
chan access[1:n](kind, other types)    # other types give buffer, number of bytes, etc.
chan open_reply[1:n](int)    # field is server index or error indication
chan access_reply[1:n](result types)    # result types 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)
                 ...
```

Greg Andrews

SR

# Arquiteturas com pools de threads

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



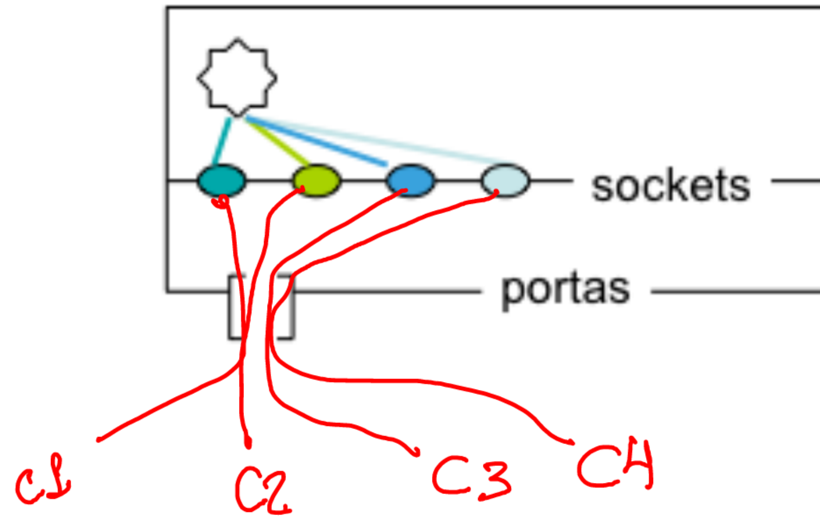
# Servidores Monotarefa

- 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
```

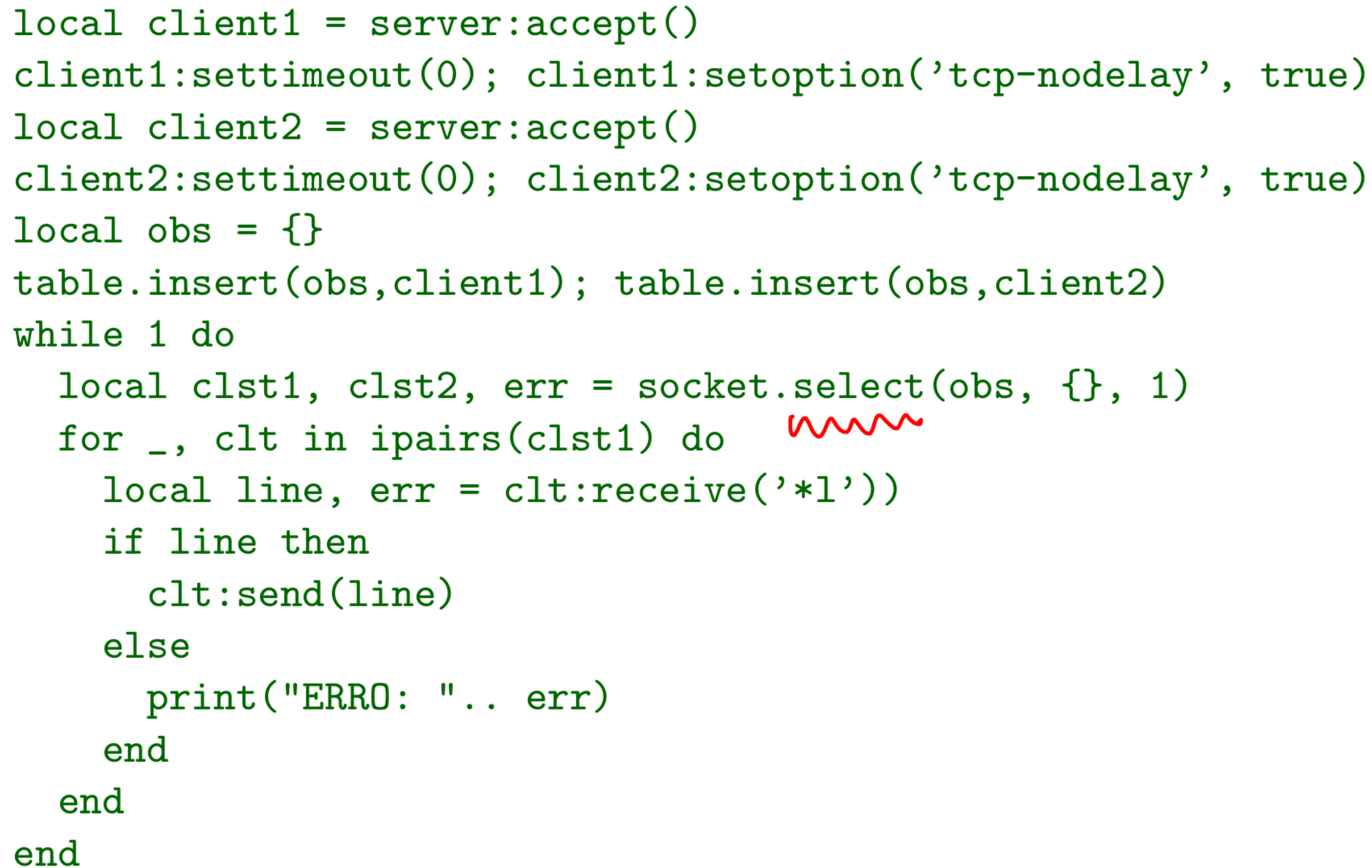
- caso geral: atendimento requer "diálogo" entre cliente e servidor
  - estado do diálogo deve ser mantido em variáveis globais

# Sockets: monotarefa




servidor

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



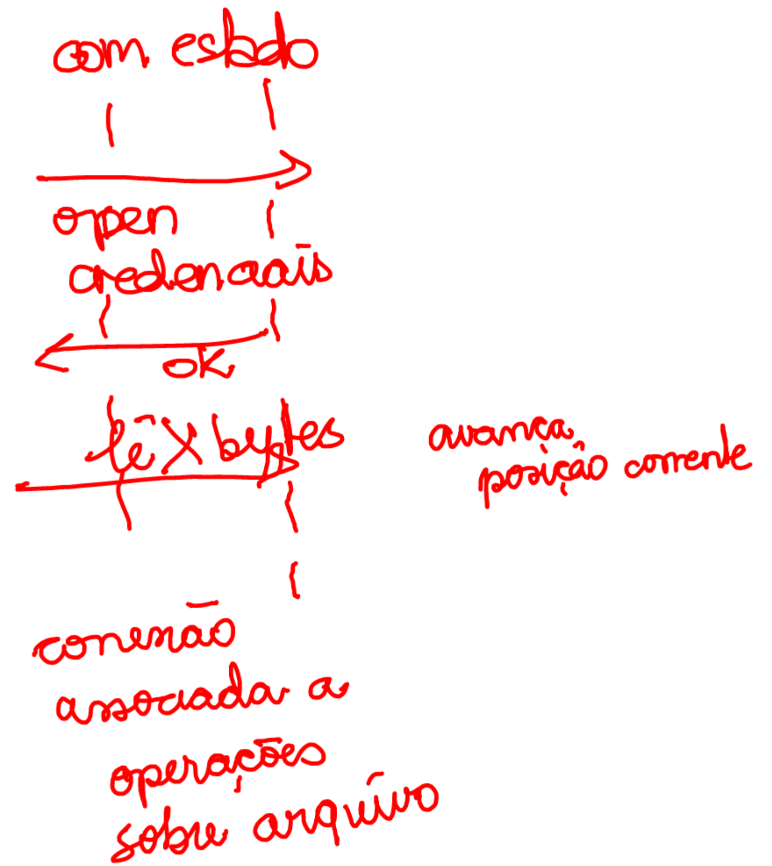
```
local client1 = server:accept()
client1:settimeout(0); client1:setoption('tcp-nodelay', true)
local client2 = server:accept()
client2:settimeout(0); client2:setoption('tcp-nodelay', true)
local obs = {}
table.insert(obs,client1); table.insert(obs,client2)
while 1 do
  local clst1, clst2, err = socket.select(obs, {}, 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 com e sem estado

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

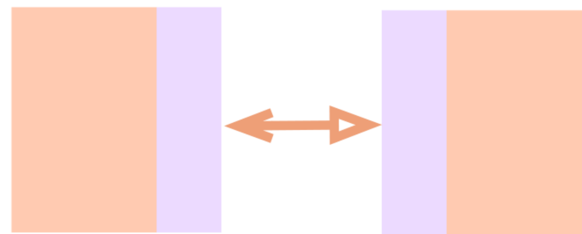
simplicidade X desempenho

SERV ARQ.
req


com estado
open
credenciais
ok
lê X bytes
avança posição corrente
conexão associada a operações sobre arquivo

# Clientes magros e gordos

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



+processamento cliente pode conseguir diminuir transf. dados