

# Sistemas Distribuídos

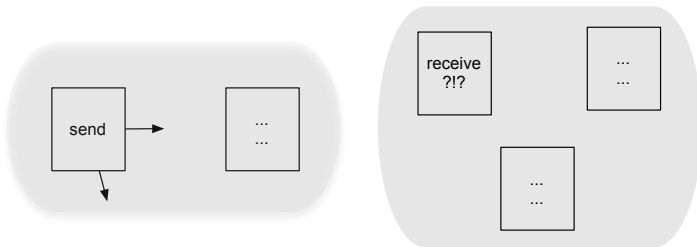
## Comunicação e Coordenação – Clientes e Servidores

março de 2021



# Comunicação entre Processos Distribuídos

- troca de mensagens exige coordenação



- quem envia? quem recebe? como é o fluxo de mensagens?
- usos de *padrões* arquiteturais facilita compreensão



# Padrão cliente-servidor

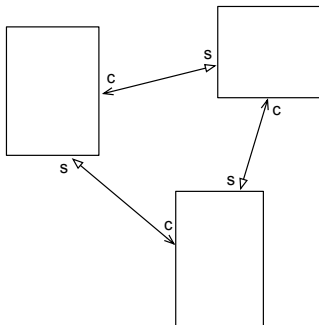
- servidor sempre a espera de comunicação
  - em programas convencionais: bloqueado
- cliente sempre inicia a comunicação
  - assimetria: cliente deve saber localizar servidor
  - envio de requisição associado a um ou mais recebimentos

## 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

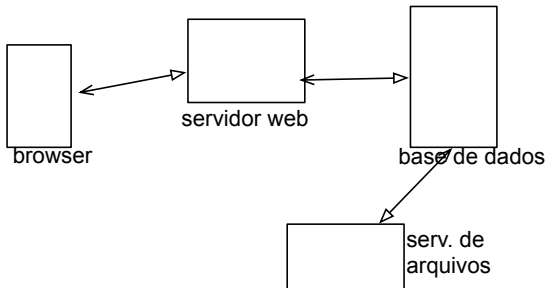
## padrão de comunicação mais difundido

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



# Arquiteturas de n camadas

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



- 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 limitados
    - relação com multi-tarefa cooperativo



- 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 – Lua e 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
```





- uso de endereços bem conhecidos
  - padronização
- servidores de nomes



# Servidores concorrentes multitarefa

- threads x processos
  - simplicidade do modelo de processos
- escalabilidade

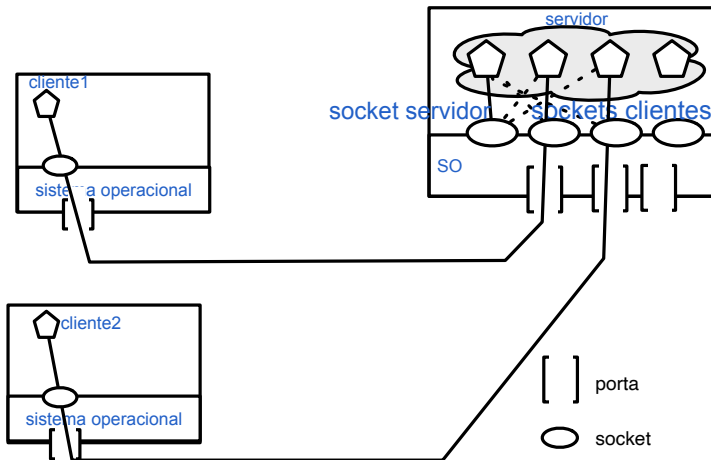


# Servidores multitarefa sob demanda

- 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

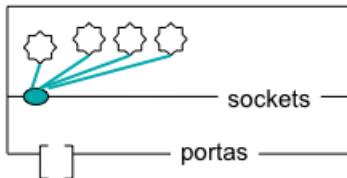


# Servidores multitarefa com pré-alocação

- 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: multitarefa com pré-alocação



# Arquiteturas com pools de threads

- uso de tarefas no nível de aplicação e threads de sistema operacional
  - escalonamento cooperativo entre tarefas de aplicação?
    - ou bloqueio de thread de SO

tarefas aplicação



threads SO



# 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

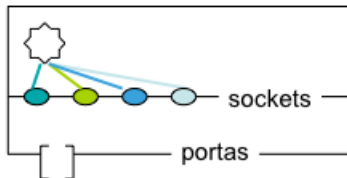
```
event <Request | clid, rqid, rq>  
    res = handle (rq);  
    trigger <Reply | clid, rqid, res>  
end
```

- 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
end
```



# Servidores com e sem estado

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

simplicidade X desempenho



# (conexões como estado)

- dificuldades de gerenciar e coletar conexões
- conexão entre manutenção de conexão e histórico de cliente



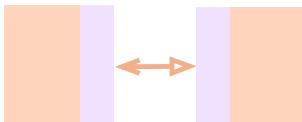
# Caches em Clientes

- uso comum em muitas aplicações
- com e sem estado



# Clientes magros e gordos

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



# REST (Representational State Transfer)

- padrão de arquitetura proposto para serviços Web
- Roy Fielding, 2000

- cliente-servidor
- sem estado
- cache
- camadas
- código sob demanda
- interface uniforme

