

ALua

asynchronous communication in Lua

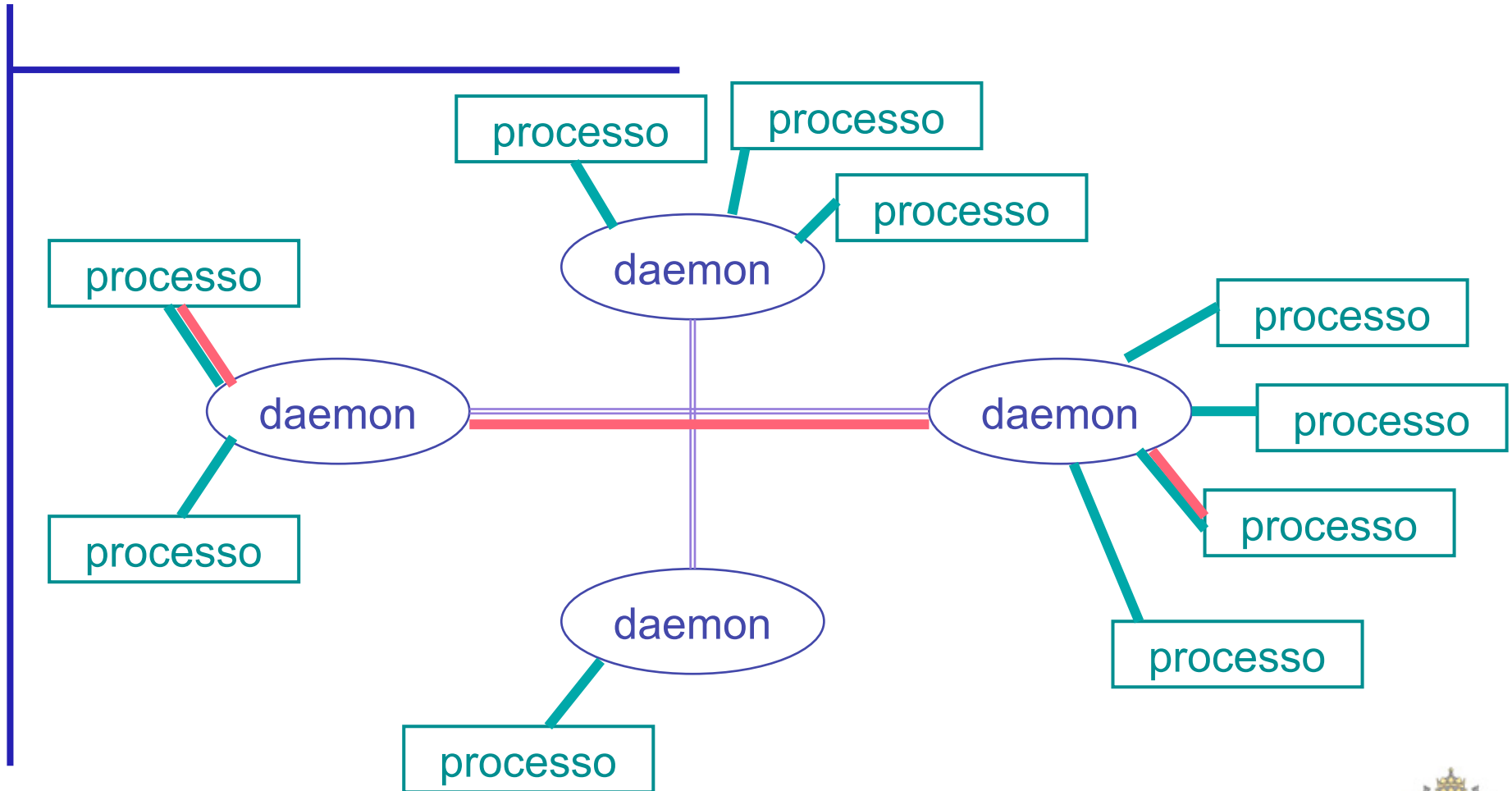


o que é

- modelo de programação
 - troca de mensagens Lua
 - assincronismo
- modelo de concorrência
 - cada evento tratado até o final
- estrutura da rede
 - daemons intermediando comunicação



ALua



— alua.send



exemplo de uso

```
function send_callback(reply)
    ...
end

function spawn_callback(reply)
    for id, proc in pairs(reply.processes) do
        if proc.status == "ok" then
            alua.send(id, [[ print("Hello World!") ]], send_callback)
        end
    end
end

alua.open()
alua.spawn(12, spawn_callback)
alua.loop()
```



- funções da biblioteca sempre com callbacks



redes de daemons

- daemons criados com open ou create
- conexão entre eles com link
- daemons também usados para extensões



envio - exemplo 1

- processo A executa:
 - `send (pid_B, [["print (" .. minhavar .. ")"]])`
- em B:
 - `"print (" .. minhavar .. ")"`
 - processando: `print (3)`



envio - exemplo 2

- processo A executa:
 - send (pid_B, [[send (pid_A, "print (" .. minhavar .. ")")]])
- em B:
 - send (pid_A, " print (" .. minhavar .. ")")
 - send (pid_A, " print (3) ")
- em A:
 - print (3)



modelo de máquina de estados

```
function envia ( ... )
  ...
  alua.send (lista_procs,
    [[ ... alua.send (pid_A, "recebida(m,"
      .. meuld .. ")")]])
  pendentos = -- tamanho da lista
end

function recebida (msg, quem)
  print (quem .. " recebeu msg " .. msg)
  pendentos = pendentos - 1
  if pendentos == 0 then print ("todos receberam!")
  end
end
```



uso de redefinição de fçs

```
function envia ( ... )  
  ...  
  alua.send (lista_procs,  
    [[alua.send (pid_A, "recebida(m," .. meuld .. ")")]])  
  pendentos = -- tamanho da lista  
end
```

```
function recebida (msg, quem)  
  -- trata msg  
  pendentos = pendentos - 1  
  if pendentos <= MINIMO then  
    recebida = function () end  
  end  
end
```

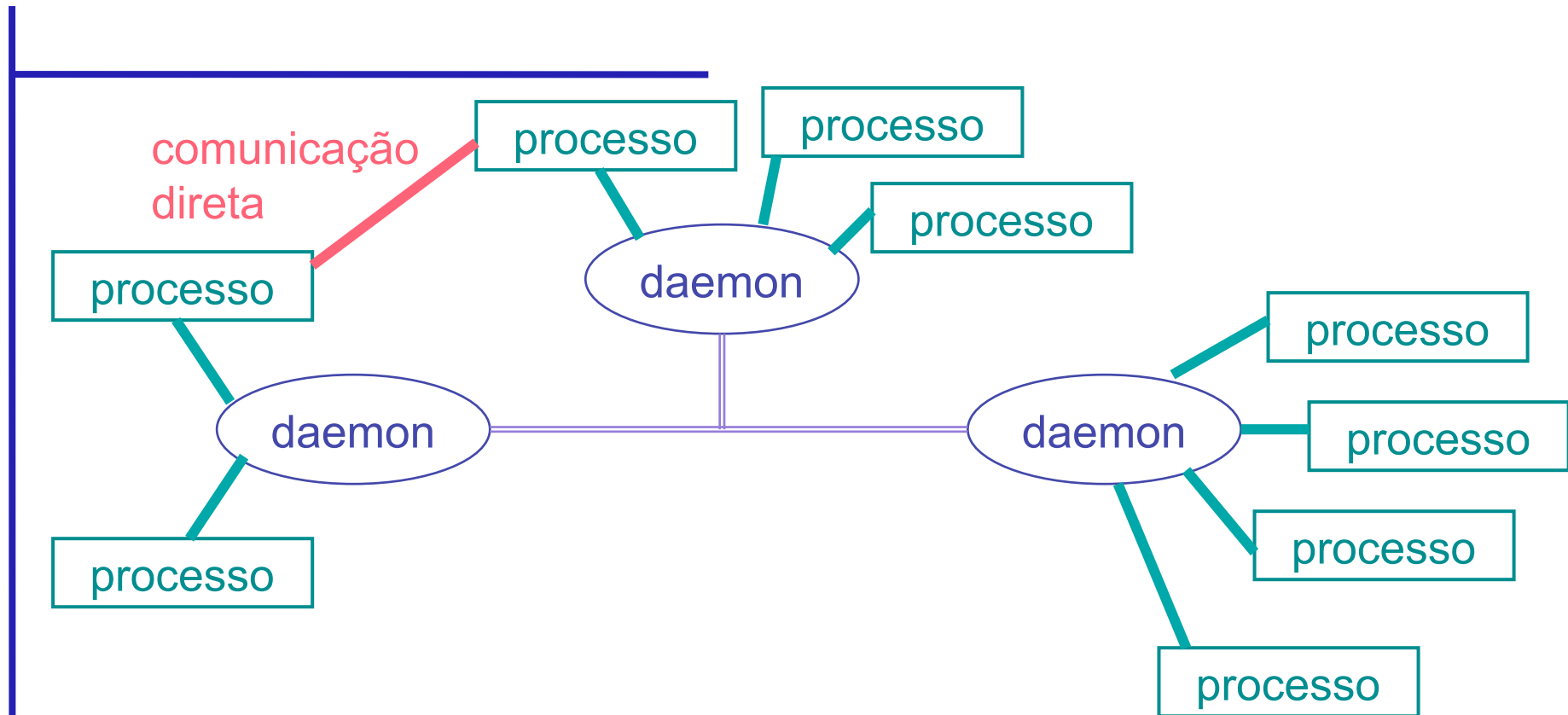


modelo de concorrência

- código de mensagem não pode conter chamadas bloqueantes
 - problemas com comunicação bloqueante
- API de canais



canais



- callbacks para leitura e escrita definidas pelo programador



ainda modelo de programação

- programação com strings
 - complexidade e propensão a bugs
 - flexibilidade X falta de estrutura
- abstrações sobre o Alua
 - luats: espaço de tuplas
 - luarpc: chamada remota de procedimentos
 - dalua: algoritmos distribuídos
 - ...



daíua

- facilidades para criação e coordenação de grupos de processos



Módulo *dalua*

- Envio de mensagens

- `dalua.send(dest_procs, nome_func, arg1, arg2, ...)`
 - Cada mensagem é uma chamada de função remota
 - `dest_procs` pode ser um ou mais identificadores de processo
 - Argumentos podem ser `number`, `string`, `boolean` ou `table`
- `dalua.self()`
 - Obtém o identificador do processo corrente
- Exemplo

```
require("dalua")
```

```
dalua.init("127.0.0.1", 4321)
```

```
dalua.send(dalua.self(), "print", "hello  
world!")
```

```
dalua.loop()
```



Módulo *events*

- Modelo de eventos x Modelo de callbacks
- Quando uma ação é finalizada, um evento correspondente é disparado
- Os processos podem se registrar para receber os eventos desejados e tratá-los se necessário
 - `dalua.events.monitor(nome_evento, tratador_do_evento)`
 - `dalua.events.ignore(nome_evento, tratador_do_evento)`
 - *nome_evento* é uma string
 - *tratador_evento* é uma função
- É possível disparar seus próprios eventos
 - `dalua.events.raise(nome_evento, procs, arg1, arg2...)`



Módulo events

- Exemplo

```
require("daluva")
local contador = 5

function envia()
    daluva.send(daluva.self(), "print",
"hello world!")
    contador = contador - 1
    if contador == 0 then
        daluva.events.ignore("daluva_send",
envia)
    end
end

daluva.events.monitor("daluva_init", envia)
```



end



Módulo *mutex*

- Suporte a exclusão mútua distribuída
 - `dalua.mutex.create(nome_mutex, procs)`
 - Cria um Mutex com nome *nome_mutex* dentre os processos especificados na tabela *procs*
 - `dalua.mutex.enter(nome_mutex, mutex_cs, arg1, arg2...)`
 - Efetua um pedido para entrar na região crítica
 - Ao conseguir, chama a função *mutex_cs* e seus argumentos
 - `dalua.mutex.leave(nome_mutex)`
 - Libera o acesso à região crítica a outros processos
 - `dalua.mutex.add(nome_mutex, proc)`
 - `dalua.mutex.remove(nome_mutex, proc)`
 - Adiciona ou remove processos no Mutex existente



Módulo app

- Aplicação: um grupo de processos
- Processos podem criar, entrar e sair de aplicações
 - `lua.app.init()`
 - `lua.app.create(nome_app)`
 - `lua.app.join(nome_app)`
 - `lua.app.leave(nome_app)`
 - `lua.app.destroy(nome_app)`
 - `lua.app.link(ip, porta)`
- Cada processo da aplicação conhece a lista de processos participantes
 - `lua.app.processes(nome_app)`
 - `lua.app.applications()`



Módulo app

- Exemplo

```
-- PROCESSO 1 --

function inicio()
    dalua.app.init()
end

function appinit()
    dalua.app.create("Grupo")
end

function joined(event, status, app, proc)
    print("Processo "..proc.." entrou em "..app)
    dalua.send(dalua.app.processes("Grupo"), "print", "Olá membros do Grupo!")
end

dalua.events.monitor("dalua_init", inicio)
dalua.events.monitor("dalua_app_init", appinit)
dalua.events.monitor("dalua_app_join", joined)

-- PROCESSO 2 --

function appinit()
    dalua.app.join("Grupo")
end
```



Módulo timer

- Permite executar tarefas periodicamente
 - `dalua.timer.add(proc, periodo, nvezes, func, arg1, arg2...)`
 - Cria um timer que executa a função *func* e seus argumentos no processo *proc* por *nvezes* a cada *periodo* segundos.
 - *func* e os argumentos têm o mesmo formato do *dalua.send*
 - Se *nvezes* for igual a zero, executa indefinidamente
 - Retorna um identificador de timer
 - `dalua.timer.remove(timerid)`
 - Pára e remove o timer especificado (se ainda existir)

- Exemplo

```
dalua.timer.add(dalua.self(), 1, 10, "print",  
               "1 segundo se passou...")
```



Módulos causal e total

- Oferecem serviço de ordenação de mensagens enviadas por multicast
- Ordem Causal
 - Garante que uma mensagem M2 que foi enviada em resposta a M1 será entregue depois de M1 em todos os processos
- Ordem Total
 - Garante que a ordem em que as mensagens são entregues seja a mesma em todos os processos
- Mesma maneira de usar que o *dalua.send* após inicialização do módulo



chamada remota

- chamada remota assíncrona como base
- criação dinâmica de função que realiza a chamada



chamada assíncrona

- valor retornado pode ser manipulado como qualquer função local
 - e funções podem ser passadas como argumentos

```
function register (val)
  currentVal = val
end
```

```
local get = rpc.async(servers, "getValue", register)
-- Invoke the remote function
get()
```



chamada assíncrona - implementação

```
function async(proc, func, cb)
  local f = function (...)
    -- put the arguments into a Lua table
    local args = {...}
    local idx = set_pending(cb) -- register the callback
    marshal(args)              -- process the arguments
    local chunk = string.format("rpc.request(%q, %s, %q, %q)",
                                func, alua.tostring(args), alua.id, idx)
    alua.send(proc, chunk)      -- send the request
  end
  return f
end
```



chamadas assíncronas e callbacks

```
function avrg(val)
  acc = acc + val
  count = count + 1
  if count == expected then
    print("Average: ", acc/count);
  end
end
```

uso de globais - perda da pilha!

```
-- Request the remote values.
for i = 1, expected do
  -- create function values
  local get = rpc.async(servers[i], "getValue", avrg)
  -- Invoke the remote function
  get()
end
```



em Lua: escopo léxico

```
function getavrg (servers)
  local acc = 0, local count = 0
  local expected = #servers
  function avrg(val)
    acc = acc + val
    count = count + 1
    if count == expected then
      print("Average: ", acc/count);
    end
  end
  -- Request the remote values.
  for i = 1, expected do
    -- create function values
    local get = rpc.async(servers[i], "getValue", avrg)
    -- Invoke the remote function
    get()
  end
end
```



chamada síncrona

- casamento de visão síncrona com modelo assíncrono

queremos escrever

```
res = chamadaRemota(arg1, ..., argn)
```

usa res

mas sem que o processo fique completamente bloqueado...

e sem as confusões de multithreading!

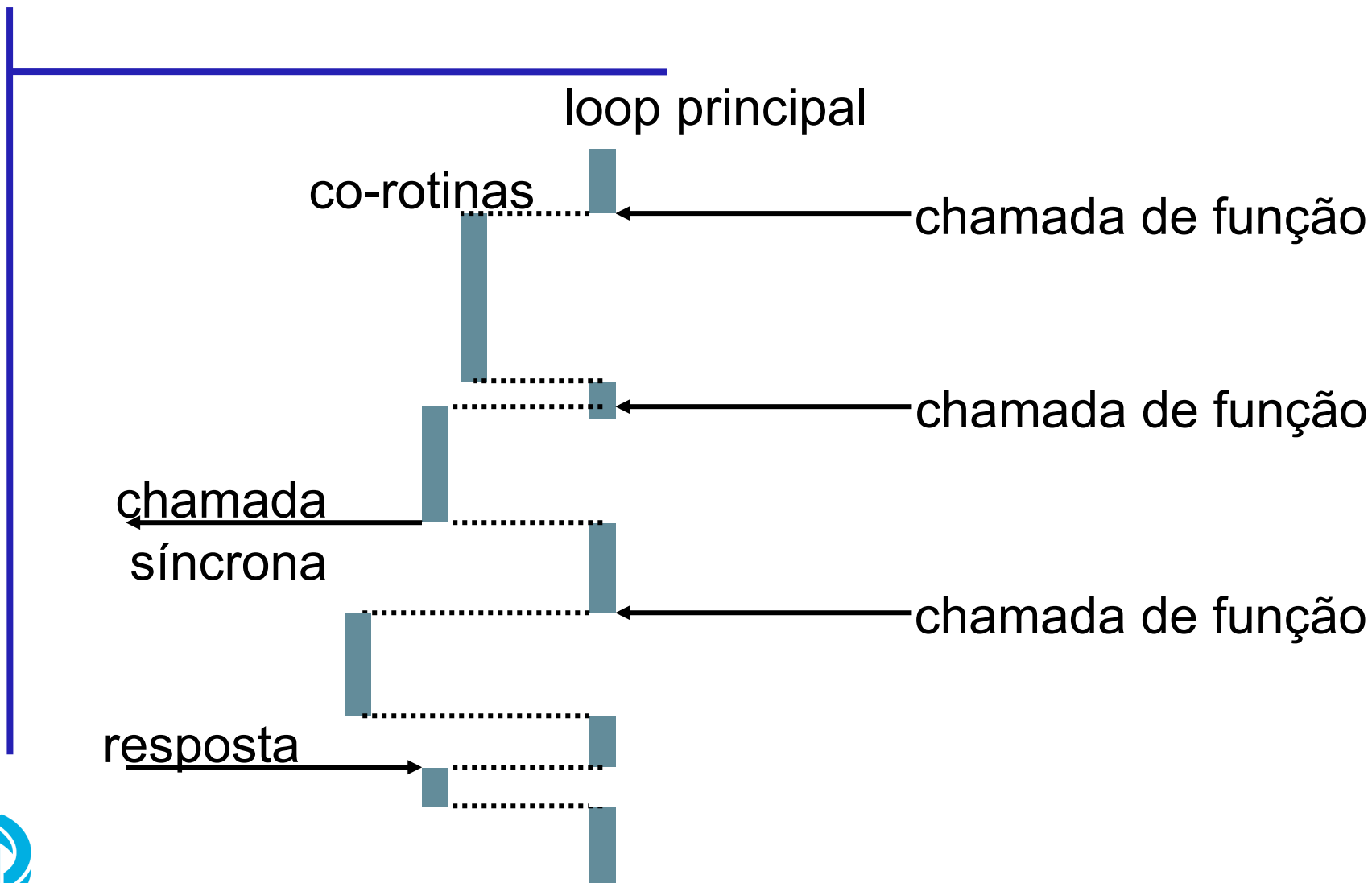


rpc e co-rotinas

- podemos executar cada função como uma co-rotina diferente
- loop de eventos fica na co-rotina principal
- escalonamento:
 - quando uma das co-rotinas faz uma chamada remota síncrona, queremos que isso implique em um yield
 - volta ao loop de eventos
 - disponível para receber novas chamadas ou respostas de chamadas anteriores



rpc e co-rotinas



construtor síncrono

```
function getavrg (servers)
  local acc = 0, local expected = #servers
  for i = 1, expected do
    -- create function values
    local get = rpc.async(servers[i], "getValue",
    avrg)
    -- Invoke the remote function
    acc = acc + get()
  end
  return acc/expected
end
```



chamada síncrona - implementação

```
function sync(proc, func)
  local f = function (...)
    -- Reference to the current coroutine
    local co = coroutine.running()
    -- Callback that will resume the execution
    local callback = function (...)
      coroutine.resume(co, ...)
    end
    -- Send the remote invocation
    async(proc, func, callback)(...)
    -- Suspend the execution before to return
    return coroutine.yield()
  end
end
return f
end
```



sincronização

- como adicionar facilidades de sincronização distribuída?
 - monitores
 - guardas



Monitores

```
local function _put(item)
  -- place item in buffer
end
local function _get()
  -- return first item in buffer
end
-- Creates a lock
local bufmon = monitor.create()
put = monitor.doWhenFree(bufmon, _put)
get = monitor.doWhenFree(bufmon, _get)
```



Sincronizadores

```
local on = false
local function check_on(request)
    return not on
end
local function trigger_on(request)
    on = true
end
local function trigger_off(request)
    on = false
end
```

...



Sincronizadores (cont)

```
local on = false
local function check_on(request) ... end
local function trigger_on(request) ... end
local function trigger_off(request) ... end
synchronizer.set_trigger("SharedResource", "turn_on",
    trigger_on)
synchronizer.set_trigger("SharedResource", "turn_off",
    trigger_off)
synchronizer.add_constraint("SharedResource",
    "turn_on", check_on)
```

