Pontifícia Universidade Católica do Rio de Janeiro  
INF 2545 – Sistemas Distribuídos Professora: Noemi 2010.1

**Trabalho 1**Tipos de Servidor

**Alunos:**  
Danilo Moret  
Thiago Manhente de C. Marques

# Experimentos propostos

Nossa proposta original era realizar todos os testes duas vezes, uma com o servidor rodando em ambiente Linux e outro em ambiente Windows. Porém, tivemos problemas para colocar o LuaPosix para funcionar no Windows, e optamos então por realizar ambos os experimentos em ambientes Linux (Ubuntu e Kubuntu)

Os ambientes em que o servidor rodará serão os seguintes:

Virtualbox Ubuntu 64, RAM 2GB sobre host Intel(R) Core(TM)2 Duo CPU P8700 @ 2.53GHz, RAM 8GB

Kubuntu sobre Intel® Core™ 2 Duo CPU T8100 @ 2.10GHz, RAM 2GB

As seguintes configurações de servidor serão utilizadas:

Servidor monoprocesso;

Servidor multiprocesso com alocação sob demanda;

Servidor multiprocesso com pré-alocação de 5 processos; e

Servidor multiprocesso com pré-alocação de 10 processos.

Para cada tipo de servidor, serão testados os seguintes cenários de demanda:

Demanda baixa: 1 e 5 processos clientes acessando o servidor simultaneamente.

Demanda média: 10 processos clientes acessando o servidor simultaneamente.

Demanda alta: 15 processos clientes acessando o servidor simultaneamente.

Para cada par de configurações servidor-demanda acima, testaremos o envio de arquivos com os seguintes tamanhos:

10KB, 100KB, 1MB, 10MB, 100MB.

Teremos assim, no total, 160 experimentos:

4 configurações de servidor \* 5 tamanhos de arquivos\* 4 cenários de demanda \* 2 ambientes.

Em cada teste, as seguintes propriedades serão medidas:

Tempo total para transferência dos arquivos.

Maior tempo de transferência dos arquivos.

Tempo médio de transferência dos arquivos.

Ocorrência de timeouts nos clientes.

Número médio de clientes atendidos por minuto pelo servidor (throughput).

# Resultados esperados

Esperamos que o servidor monoprocessado tenha um bom desempenho com a conexão de poucos clientes, visto que não possui sobrecarga de criação de processos e trocas de contexto. Esperamos, porém, que esse desempenho vá decaindo vertiginosamente conforme o número de clientes cresça, visto que ele só pode atender a um de cada vez.

Esperamos também que os servidores não apresentem grandes diferenças de desempenho para arquivos de tamanho bem reduzido, por ser extremamente rápida a resposta do servidor para o cliente.

# Resultados auferidos

Nas próximas páginas estão as tabelas com os tempos auferidos. Os resultados vem do processamento dos logs gerados pelo programa cliente.

Alguns dos testes propostos não puderam ser realizados devido a travamentos nas máquinas que rodavam o servidor. Os testes que não puderam ser realizados foram:

Servidor multiprocessos sob demanda transferindo arquivos de 100MB para 10 e 15 clientes (em ambos os ambientes de teste)

Servidor multiprocessos pré-alocados (5 e 10 processos) transferindo arquivos de 100MB para 10 e 15 clientes (no ambiente VirtualBox Ubuntu)

Tempos de transferência médios



Tempos de transferência máximos



Tempos de transferência totais



# Conclusões quanto aos resultados

## Avaliação Geral

Em geral, os resultados auferidos confirmaram as expectativas quanto ao desempenho dos servidores para poucos e muitos clientes e quanto à influência do tamanho do arquivo no desempenho do servidor.

Considerando arquivos com mais de 1MB, o servidor monoprocessado teve um melhor desempenho para pouca demanda, apresentando tempos médios de resposta menores que os demais tipos para 1 e 5 clientes. Teve, porém, desempenho inferior aos servidores pré-alocados para 10 e 15 clientes, apresentando tempos médios de resposta maiores.

Os resultados auferidos também comprovaram que, para transações em que a resposta do servidor é bastante rápida, como no caso de arquivos pequenos (até 1MB), a diferença entre os tipos de servidores é imperceptível. Nesses casos, uma implementação mais rebuscada não se justifica, podendo se optar tranquilamente por uma implementação simples de servidor monoprocesso sem nenhuma perda considerável de desempenho.

No caso específico de servidores de arquivos, a maior vantagem de se ter multiprocessos é poder sobrepor o tempo de leitura do arquivo com a execução dos outros processos. Para arquivos pequenos, como o tempo de leitura é bastante reduzido, essa vantagem não é significativa.

## Multiprocessos em computadores com um único processador

Os ambientes de teste possuíam ambos apenas um processador. Embora possuíssem mais de um núcleo, ambos os ambientes observaram queda de desempenho quando o número de processos aumentava além de determinados níveis. Cremos que isso se dê pela grande concorrência que se estabelece sobre o único processador, levando os vários processos a serem escalonados com muita rapidez e gerando uma carga extra de trocas de contexto, justificando assim a perda de desempenho.

Testes adicionais em máquinas com vários processadores seriam recomendados, porém não dispúnhamos de nenhum ambiente com tal característica.

## Criação de processos sob demanda

Ao observarmos a dificuldade que tivemos de executar os casos de teste com grande número de clientes para o servidor sob demanda, concluímos que a criação desses processos num ambiente de alta concorrência pode levar a estagnação do servidor, por gerar uma grande sobrecarga de trabalho para a criação dos processos.

Uma solução possível é misturar os conceitos de alocação sob demanda e pré-alocação, da seguinte maneira: Pré-aloca-se um determinado número de processos no servidor. Ao verificar que a demanda está crescendo e chegando próximo ao limite dos processos alocados, realiza-se uma nova pré-alocação de mais um determinado número de processos para atender a crescente demanda. Analogamente, observando-se a diminuição da demanda, poder-se-ia terminar um bloco de processos para liberar recursos no servidor (em especial, tempo de escalonamento dos processos no processador).

### Questão de processos-zumbis

Durante testes com o servidor multiprocesso sob demanda no ambiente Kubuntu, percebemos que os processos-filhos que terminavam de enviar os arquivos ao invés de serem encerrados passavam para o estado processo-zumbi.

Pesquisamos e vimos que isso pode ser causado em algumas versões do Unix/Linux quando o processo-pai ignora o sinal de término do processo-filho, e que poderia ser revertido usando a função signal() para configurar SIGCHLD para SIG\_IGN. Não fizemos esse tratamento por não descobrirmos um jeito de fazê-lo em Lua, visto que a biblioteca LuaPosix aparentemente não fornece suporte para essa função signal().

Nos testes que realizamos, cremos que a existência desses processos-zumbis não tenha prejudicado tanto o desempenho. Porém, em um ambiente de maior escala, essa questão é importante por abrir a possibilidade de a tabela de processos ser poluída com esses processos.

# Anexos - Código

## sd\_common.lua

--[[

| Pontifícia Universidade Católica do Rio de Janeiro

| INF2545 - Sistemas Distribuidos Prof.: Noemi

| Alunos: Danilo Moret

| Thiago M. C. marques

|

| Trabalho 1 - Tipos de servidores 2010.1

--]]

--[[

| File Description:

| This file holds common methods and constants to all servers and clients, ensuring they are using

| similar methods.

|

| Variables:

| filepath - Path to the file to be sent by the server. Default is "10MB.txt".

| chunk\_size - Maximum size of each chunk to be sent to the client. Default is 10KB.

--]]

local socket = require("socket")

require("posix")

-- File size constants.

local KB\_const = 1024 -- Constant to convert KB in bytes.

local MB\_const = 1024 \* KB\_const -- Constant to convert MB in bytes.

local size\_10KB = 10 \* KB\_const

local size\_100KB = 100 \* KB\_const

local size\_1MB = 1 \* MB\_const

local size\_10MB = 10 \* MB\_const

local size\_100MB = 100 \* MB\_const

-- Variables initialization

local filepath = filepath or "10MB.txt" -- Path to file to be sent to clients.

local serv\_well\_known\_port = serv\_well\_known\_port or 1111

-- Initializing instrumentation counters

local instrumentation\_steps = 5

local \_debug = \_debug or false

-- Instrumentation: Lists initial parameters.

function list\_initial\_param(arg)

if arg then

print("-- Initial parameters --")

for key, val in ipairs(arg) do

print("Arg " .. key .. ": " .. val)

end

print("-- End of Initial parameters --")

else

print("-- No initial parameters passed (arg = nil)")

end

end

-- Print a log message in the standard output. Format is: "[time] name: message". Returns the time.

function time\_log(message, name, end\_time)

end\_time = end\_time or socket.gettime()

local header = "[" .. end\_time .. "] "

if name then

header = header .. name .. ": "

end

if \_debug then print(header .. message) end

return end\_time

end

function elapsed\_time\_log(message, name, start\_time, end\_time)

assert(start\_time, " ! Error - Start time must not be nil.")

end\_time = end\_time or socket.gettime()

local elapsed\_time = end\_time - start\_time

message = message .. " (elapsed time: " .. elapsed\_time .. ")"

return time\_log(message, name, end\_time)

end

-- Getting server up on well known port.

function server\_up()

local server, server\_error = socket.bind("\*", serv\_well\_known\_port)

assert(server, " ! Error - Could not create server. " .. (server\_error or ""))

local ip, port = server:getsockname()

time\_log("Server listening on port " .. port)

return server

end

-- File sending to client.

function send\_file(client)

local \_file = io.open(filepath, "r")

if \_file then

client:send(\_file:read("\*all"))

\_file:close()

if \_debug then time\_log("Transferred " .. filepath .. " file.") end

else

time\_log("ERROR! File " .. filepath .. " could not be opened. Sending aborted.")

end

end

-- File receiveng from server.

function client\_run(name, serv\_addr, serv\_port)

-- Parameter's default values initialization.

serv\_addr = serv\_addr or "localhost"

serv\_port = serv\_port or serv\_well\_known\_port

local client = socket.tcp()

-- Start connection.

client:settimeout(30)

local connection\_req\_time = time\_log("Starting connection.", name)

client:connect(serv\_addr, serv\_port)

local connect\_sucess\_time = elapsed\_time\_log("Connection stabilished.", name, connection\_req\_time)

if not client then

local error\_msg = "Could not connect to server at address: " .. serv\_addr .. " port: " .. port

return nil, error\_msg

end

-- Start file receival.

client:settimeout(10\*60) -- 10 min.

local receival\_beg\_time = time\_log("Starting file receival.", name)

local sample, err = client:receive("\*a")

client:close()

local receival\_end\_time = elapsed\_time\_log("File receival complete.", name, receival\_beg\_time)

local total\_time = elapsed\_time\_log("Total time taken by client.", name, connection\_req\_time, receival\_end\_time)

return sample, err

end

-- Receive files loop

function receive\_files(files, name, serv\_addr, serv\_port)

local files\_per\_step = files / instrumentation\_steps

local total\_time = 0

local higher\_time = 0

for j=1, instrumentation\_steps do

-- Using socket.gettime to obtain better precision, i.e., not in seconds only

-- local start\_time = os.time()

local start\_time = socket.gettime()

for i=1, files\_per\_step do

-- Execute check step.

local sample, err = client\_run(name, serv\_addr, serv\_port)

if not sample then

print(" ! Error - File not received. " .. (err or ""))

end

end

local end\_time = socket.gettime()

local elapsed\_time = end\_time - start\_time

if elapsed\_time > higher\_time then

higher\_time = elapsed\_time

end

total\_time = total\_time + elapsed\_time

-- When using socket.gettime, the method os.difftime is no longer usueful, so we're subtracting directly. It might not work the same way on differente OS's

-- print("Sent " .. files\_per\_step .. " files in " .. os.difftime(os.time(), start\_time) .. "s")

print(name .. " received " .. files\_per\_step .. " files in " .. elapsed\_time .. "s")

end

print(name .. "| Files transfered: " .. files .. "\t Total time taken: " .. total\_time)

print("Higher time taken: " .. higher\_time .. "\t Average time taken: " .. total\_time/files)

end

## multi\_client.lua

--[[

| Pontifícia Universidade Católica do Rio de Janeiro

| INF2545 - Sistemas Distribuidos Prof.: Noemi

| Alunos: Danilo Moret

| Thiago M. C. marques

|

| Trabalho 1 - Tipos de servidores 2010.1

--]]

require("posix")

require("sd\_common")

if \_debug then

list\_initial\_param(arg)

end

-- Initializing instrumentation counters

local files = files or 20

local name = name or "mc"

local simultaneous\_clients = simultaneous\_clients or 4

local serv\_addr = serv\_addr or "localhost"

for cli=1, simultaneous\_clients do

pid = posix.fork()

if pid ~= 0 then -- Child process

local client\_name = "multi\_cli" .. name .. cli

receive\_files(files, client\_name, serv\_addr, serv\_well\_known\_port)

os.exit()

end

end

## file\_server\_monoproc.lua

--[[

| Pontifícia Universidade Católica do Rio de Janeiro

| INF2545 - Sistemas Distribuidos Prof.: Noemi

| Alunos: Danilo Moret

| Thiago M. C. marques

|

| Trabalho 1 - Tipos de servidores 2010.1

--]]

--[[

| File Description:

| Sets a monoprocess server that sends a specified file to the client.

--]]

local socket = require("socket")

require("sd\_common")

if \_debug then

list\_initial\_param(arg)

end

-- Getting the server up

local server = server\_up()

while true do

-- Client connection recieval.

local client = server:accept()

send\_file(client)

client:close()

end

## file\_server\_on\_demand.lua

--[[

| Pontifícia Universidade Católica do Rio de Janeiro

| INF2545 - Sistemas Distribuidos Prof.: Noemi

| Alunos: Danilo Moret

| Thiago M. C. marques

|

| Trabalho 1 - Tipos de servidores 2010.1

--]]

--[[

| File Description:

| Sets a on-demand multiprocess server that sends a specified file to the client.

--]]

local socket = require("socket")

require("posix")

require("sd\_common")

if \_debug then

list\_initial\_param(arg)

end

-- Getting the server up

local server = server\_up()

while true do

-- Client connection recieval.

local client = server:accept()

local pid = posix.fork()

if pid == 0 then -- Child process, pid stores the child's PID.

server:close() -- Closes socket that the child is not using.

send\_file(client)

client:close()

os.exit() -- Child ends execution

else -- Father process, pid ~= 0

client:close() -- Closes socket that the father is not using.

end

## end

## file\_server\_pre\_allocated.lua

--[[

| Pontifícia Universidade Católica do Rio de Janeiro

| INF2545 - Sistemas Distribuidos Prof.: Noemi

| Alunos: Danilo Moret

| Thiago M. C. marques

|

| Trabalho 1 - Tipos de servidores 2010.1

--]]

--[[

| File Description:

| Sets a pre-allocated multiprocess server that sends a specified file to

| the client.

|

| Variables:

| chunk\_size - Maximum size of each chunk to be sent to the client. Default is 10KB.

| num\_proc - Number of processes to be pre-allocated. Default is 5.

--]]

local socket = require("socket")

require("posix")

require("sd\_common")

if \_debug then

list\_initial\_param(arg)

end

-- Total processes

local num\_proc = num\_proc or 5

-- Getting the server up

local server = server\_up()

local pid, proc\_num

-- Setting que process pool.

for i = 1, num\_proc - 1 do

proc\_num = i

pid = posix.fork()

if pid == 0 then -- Child process.

break

end

end

-- Listening to client requests.

while true do

-- Client connection recieval.

local client = server:accept()

send\_file(client)

client:close()

end