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

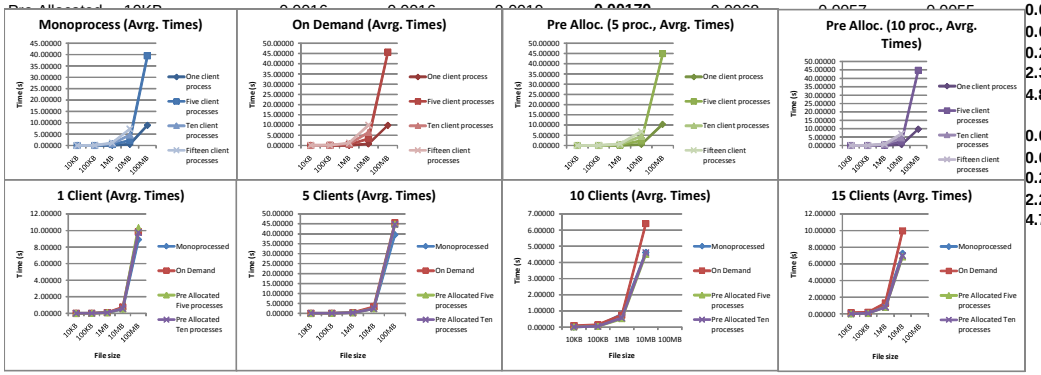
Nas próximas páginas estão as tabelas com os tempos auferidos. Os resultados vem do processamento dos logs gerados pelo programa cliente.

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)

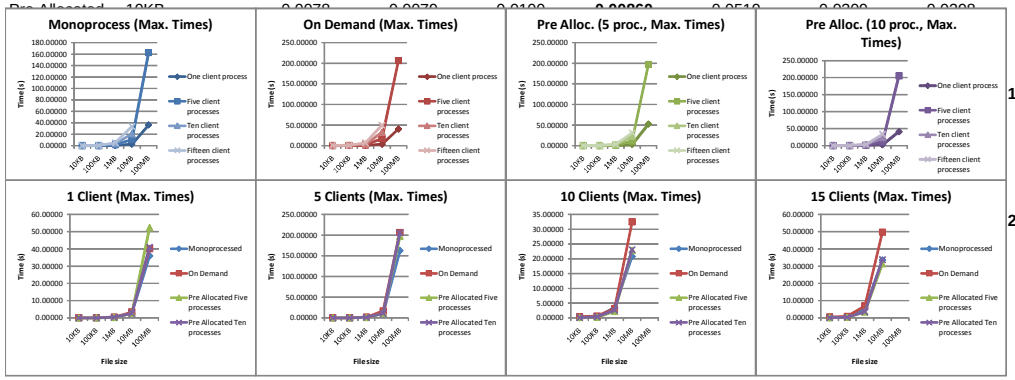
Tempos de transferência médios

		One client process				Five client processes				Ten client processes				Fifteen client processes			
		Run 1	Run 2	Run 3	**Average**	Run 1	Run 2	Run 3	**Average**	Run 1	Run 2	Run 3	**Average**	Run 1	Run 2	Run 3	**Average**
Monoprocessed	10KB	0.0014	0.0015	0.0021	**0.00169**	0.0060	0.0066	0.0180	**0.01019**	0.0117	0.0110	0.0112	**0.01132**	0.0171	0.0164	0.0187	**0.01737**
	100KB	0.0053	0.0052	0.0053	**0.00529**	0.0283	0.0234	0.0234	**0.02503**	0.0521	0.0496	0.0483	**0.05002**	0.0760	0.0787	0.0773	**0.07733**
	1MB	0.0968	0.0801	0.0816	**0.08616**	0.4224	0.4036	0.3905	**0.40554**	0.8304	0.7927	0.7906	**0.80457**	1.1618	1.1841	1.2019	**1.18260**
	10MB	0.5408	0.5076	0.5263	**0.52490**	2.3420	2.3269	2.2600	**2.30963**	4.7239	4.5808	4.5410	**4.61524**	7.0976	7.4385	7.2650	**7.26702**
	100MB	8.9372	8.8908	8.8247	**8.88426**	39.9896	39.1601	39.3306	**39.49345**								
On Demand	10KB	0.0111	0.0105	0.0107	**0.01076**	0.0489	0.0471	0.0465	**0.04752**	0.0946	0.0969	0.0987	**0.09672**	0.1391	0.1447	0.1363	**0.14002**
	100KB	0.0205	0.0196	0.0198	**0.01999**	0.0824	0.0800	0.0803	**0.08089**	0.1381	0.1478	0.1487	**0.14485**	0.2142	0.2119	0.2044	**0.21019**
	1MB	0.1035	0.1013	0.1010	**0.10197**	0.4270	0.4220	0.4073	**0.41876**	0.7381	0.7435	0.7603	**0.74731**	1.2385	1.4443	1.2505	**1.31107**
	10MB	0.7685	0.7627	0.7716	**0.76762**	3.4644	3.3438	3.2827	**3.36363**	6.5403	6.4307	6.2528	**6.40795**	9.6603	10.2973	9.9281	**9.96190**
	100MB	9.7782	9.8243	9.9104	**9.83764**	46.5984	45.2701	44.7441	**45.53755**								
Pre Allocated	10KB									0.0154	0.0144	0.0133	**0.01436**	0.0199	0.0205	0.0187	**0.01972**
	100KB									0.0485	0.0492	0.0470	**0.04823**	0.0706	0.0865	0.0754	**0.07748**
	1MB									0.5225	0.5441	0.4935	**0.52004**	0.7909	0.7958	0.8052	**0.79727**
	10MB									4.5642	4.6499	4.3089	**4.50769**	6.7724	7.0252	6.7404	**6.84601**
	100MB																
Pre Allocated Five processes	10KB									0.0117	0.0132	0.0118	**0.01222**	0.0195	0.0183	0.0173	**0.01835**
	100KB									0.0560	0.0657	0.0478	**0.05646**	0.0725	0.0747	0.0805	**0.07591**
	1MB									0.6139	0.5508	0.5469	**0.57053**	0.7518	0.7877	0.8489	**0.79610**
	10MB									4.6694	4.5750	4.5857	**4.61003**	7.6153	6.8482	6.5544	**7.00601**
	100MB																



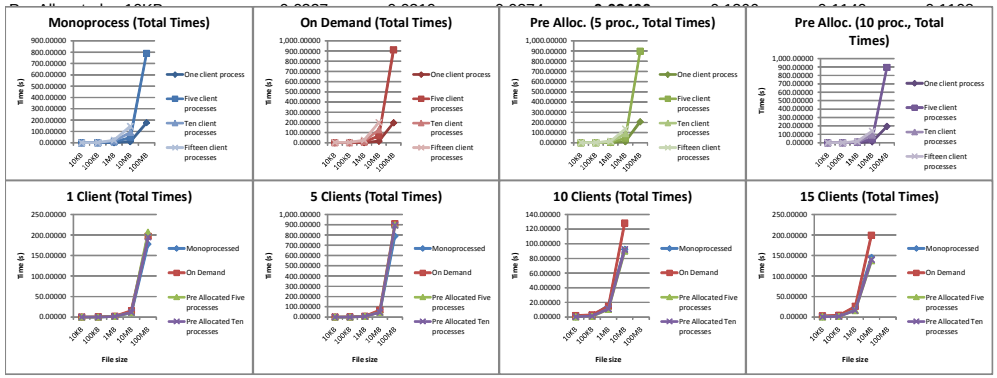
Tempos de transferência máximos

		One client process				Five client processes				Ten client processes				Fifteen client processes			
		Run 1	Run 2	Run 3	**Average**	Run 1	Run 2	Run 3	**Average**	Run 1	Run 2	Run 3	**Average**	Run 1	Run 2	Run 3	**Average**
Monoprocessed	10KB	0.0073	0.0072	0.0120	**0.00881**	0.0353	0.0438	0.2666	**0.11523**	0.0660	0.0638	0.0637	**0.06448**	0.1086	0.1134	0.1148	**0.11224**
	100KB	0.0237	0.0240	0.0237	**0.02383**	0.1380	0.1060	0.1041	**0.11601**	0.2519	0.2439	0.2162	**0.23735**	0.3770	0.3876	0.3896	**0.38476**
	1MB	0.5728	0.3304	0.3327	**0.41198**	1.8747	1.7867	1.6308	**1.76404**	4.0690	3.3075	3.3108	**3.56244**	4.8274	4.9530	5.1045	**4.96161**
	10MB	2.7737	2.4599	2.4499	**2.56119**	10.8001	10.2703	9.8035	**10.29133**	22.9954	19.6855	19.7979	**20.82628**	30.3469	36.3157	34.1701	**33.61092**
	100MB	36.3952	36.0153	35.7721	**36.06085**	164.5341	162.0253	161.6153	**162.72490**								
On Demand	10KB	0.0480	0.0446	0.0445	**0.04569**	0.2107	0.2016	0.2003	**0.20418**	0.4001	0.4288	0.4147	**0.41451**	0.5911	0.6145	0.5700	**0.59185**
	100KB	0.0893	0.0878	0.0915	**0.08954**	0.4137	0.3603	0.3525	**0.37549**	0.5970	0.6430	0.6566	**0.63220**	0.9197	0.9303	0.8886	**0.91286**
	1MB	0.4634	0.4214	0.4227	**0.43586**	1.8653	1.8447	1.8551	**1.85504**	3.1129	3.1053	3.3290	**3.18240**	6.9738	8.0030	5.7750	**6.91728**
	10MB	3.4592	3.4055	3.4219	**3.42887**	18.3127	16.5479	15.5214	**16.79402**	33.6303	32.7218	31.1995	**32.51720**	46.2348	47.8871	55.2949	**49.80561**
	100MB	39.9421	39.8973	40.8026	**40.21399**	206.9945	208.1982	204.4793	**206.55737**								
Pre Allocated	10KB								**0.03722**	0.1263	0.1182	0.0886	**0.11104**	0.1167	0.1550	0.1369	**0.13618**
	100KB								**0.13085**	0.2525	0.2581	0.2279	**0.24618**	0.3514	0.4071	0.3605	**0.37299**
	1MB								**1.15387**	2.1932	2.3420	2.0661	**2.20041**	3.3252	3.2887	3.5163	**3.37674**
	10MB								**11.28902**	25.3070	22.2781	21.0727	**22.88591**	31.8221	31.1127	31.0228	**31.31923**
	100MB								**197.50087**								
Pre Allocated (10 processes)	10KB								**0.03469**	0.0710	0.0752	0.0666	**0.07093**	0.1176	0.1113	0.1042	**0.11104**
	100KB								**0.14407**	0.3116	0.5515	0.2443	**0.36912**	0.3704	0.3583	0.4586	**0.39578**
	1MB								**1.31460**	2.7777	2.4878	2.3611	**2.54221**	3.3741	3.3618	3.8807	**3.53889**
	10MB								**11.21063**	23.4844	22.5504	23.3291	**23.12132**	34.7861	34.9307	31.7953	**33.83739**
	100MB								**205.52975**								



Tempos de transferência totais

		One client process				Five client processes				Ten client processes				Fifteen client processes			
		Run 1	Run 2	Run 3	**Average**	Run 1	Run 2	Run 3	**Average**	Run 1	Run 2	Run 3	**Average**	Run 1	Run 2	Run 3	**Average**
Monoprocessed	10KB	0.0286	0.0310	0.0421	**0.03389**	0.1191	0.1312	0.3609	**0.20374**	0.2347	0.2199	0.2249	**0.22649**	0.3413	0.3271	0.3735	**0.34732**
	100KB	0.1059	0.1050	0.1066	**0.10580**	0.5659	0.4685	0.4672	**0.50054**	1.0430	0.9917	0.9668	**1.00046**	1.5199	1.5736	1.5463	**1.54656**
	1MB	1.9355	1.6025	1.6316	**1.72321**	8.4486	8.0728	7.8109	**8.11077**	16.6089	15.8530	15.8123	**16.09139**	23.2351	23.6830	24.0377	**23.65192**
	10MB	10.8159	10.1516	10.5263	**10.49794**	46.8396	46.5377	45.2005	**46.19260**	94.4781	91.6168	90.8195	**92.30478**	141.9523	148.7699	145.2991	**145.34042**
	100MB	178.7447	177.8163	176.4944	**177.68514**	799.7925	783.2020	786.6127	**789.86907**								
On Demand	10KB	0.2219	0.2091	0.2146	**0.21517**	0.9777	0.9428	0.9306	**0.95035**	1.8911	1.9371	1.9748	**1.93433**	2.7819	2.8935	2.7257	**2.80034**
	100KB	0.4100	0.3928	0.3968	**0.39987**	1.6480	1.6005	1.6051	**1.61786**	2.7617	2.9561	2.9733	**2.89703**	4.2838	4.2387	4.0890	**4.20382**
	1MB	2.0704	2.0268	2.0210	**2.03939**	8.5400	8.4392	8.1466	**8.37525**	14.7627	14.8704	15.2055	**14.94623**	24.7696	28.8855	25.0093	**26.22146**
	10MB	15.3709	15.2548	15.4314	**15.35237**	69.2875	66.8755	65.6545	**67.27253**	130.8068	128.6143	125.0558	**128.15898**	193.2053	205.9469	198.5621	**199.23809**
	100MB	195.5646	196.4857	198.2080	**196.75272**	931.9677	905.4027	894.8827	**910.75101**								



0.12045	0.3085	0.2882	0.2651	**0.28729**	0.3986	0.4103	0.3742	**0.39437**
0.53469	0.9707	0.9839	0.9395	**0.96469**	1.4117	1.7298	1.5073	**1.54963**
5.33911	10.4500	10.8828	9.8695	**10.40077**	15.8170	15.9158	16.1032	**15.94532**
46.38691	91.2849	92.9987	86.1775	**90.15373**	135.4478	140.5050	134.8078	**136.92019**
897.97978								
0.12258	0.2336	0.2639	0.2354	**0.24431**	0.3892	0.3652	0.3466	**0.36701**
0.57090	1.1194	1.3131	0.9553	**1.12927**	1.4501	1.4940	1.6104	**1.51813**
5.93020	12.2776	11.0161	10.9381	**11.41061**	15.0353	15.7537	16.9772	**15.92205**
45.66094	93.3889	91.4990	91.7137	**92.20053**	152.3068	136.9648	131.0890	**140.12018**
895.42207								

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
```