**Nova Southeastern University**

**College of Engineering and Computing**

**CISC 665 Distributed Systems**

**Winter 2017**

**(January 9– May 7, 2017)**

**Assignment #3**

Due Date: May 1, 2017

Instructor: Dr. Gregory Simco

Distributed Systems

Assignment Three

Final Paper

Bryce Tucker

Student #: N01530887

Table of Contents

[Chapter 1: Introduction 5](#_Toc478636008)

[Problem Statement 5](#_Toc478636009)

[Goal Statement 5](#_Toc478636010)

[Server 5](#_Toc478636011)

[Data store 7](#_Toc478636012)

[Client 7](#_Toc478636013)

[Chapter 2: Review of Literature 8](#_Toc478636014)

[Purpose 8](#_Toc478636015)

[Cache architecture 8](#_Toc478636016)

[Cache Replacement Strategy 9](#_Toc478636017)

[Parallel document retrieval 10](#_Toc478636018)

[Summary 11](#_Toc478636019)

[Chapter 3: Approach 12](#_Toc478636020)

[System Implementation 12](#_Toc478636021)

[Client 15](#_Toc478636022)

[Client – server communication protocol 18](#_Toc478636023)

[Server 18](#_Toc478636024)

[Data Layer 23](#_Toc478636025)

[References: 24](#_Toc478636026)

# Chapter 1: Introduction

This study provides an exploration of the same line of design reasoning as that described in the paper by Nelson [2006] *Concurrent Caching*, the effort is needed to examine an alternative implementation to verify similar results can be obtained. This study applies the concept of concurrent cache to an example application and explores the benefits and issues involved with implementing the architecture using the Java programming language.

## Problem Statement

The problem for this assignment is to explore and evaluate an implementation of a document vending application that uses a caching method of separate processes for each file being served to clients. The primary impetus for the prototype developed comes from Nelson [2006] where the idea of using processes to provide caching for individual datum is explored. The concept was then applied to the document vending scenario to create a testable prototype.

The requirements for the document vending application included accounting for costs and the ability to upload files to be available for distribution. Due to time constraints, the requirements were pared down to the basic operations of transferring the requested files to clients. A significant amount of effort was put into developing a test harness that managed the testing of the prototype using multiple clients in separate processes performing file requests.

## Goal Statement

The goal is to examine the effort involved in the design and implementation of a file server that uses the model of having separate processes provide caching of individual files. The examination will include measurements of performance differences between an implementation of the file server that does not use any caching and the prototype that uses the caching method under examination. This comparison will provide some insight into the cost and benefits of the caching method.

The measurements of performance of the document server will be the duration of time in milliseconds between when the client first requests the file and when the client fully receives the file.

# Chapter 2: Review of Literature

## Purpose

The review of literature pertaining to the implementation of a document vending application, in specific the implementation of a distributed file server with a cache was conducted. The purpose of the review was to establish a foundation of the current state of investigation into the type of implementation pertinent to the stated requirements of this assignment. From this literature review on document vending applications and cache methodologies, the focus became the paper by Nelson [2006] *Concurrent Caching*.

## Cache architecture

The paper by Nelson [2006] describes a cache implementation that uses separate processes to contain cached data. The paper describes advantages that include:

* Each datum can respond actively, independently and in parallel to requests
* The total size of cached data may exceed the memory of a single compute node
* An idle datum may be easily removed by terminating its process
* Stale data may be replaced in parallel when multiple changes to the data store occur
* Replacement policies may be implemented independently from the cache itself
* A cached datum may dynamically migrate to the CPU that best responds to requests

From this paper, the implementation of the solution for this given assignment includes the use of processes to perform the caching requirement. By following the described implementation, the cache manager will spawn processes to cache documents instead of maintaining the documents in a cache itself. When a document request is made, the cache manager will examine a maintained list to determine if the requested document is cached in a process, the IP address/port number will then be provided so the client and the given process can interact directly for the client to download the document.  
Cache replacements will be performed by terminating the process and spawning a replacement process to contain a given document. The cache manager will control the number of cache processes based on a configuration value to meet the requirement that the cache not be infinite. The cache manager will control the replacement of cached documents by tracking the number of accesses the file has received, when a cache process needs to be terminated to allow a new cache process to launch, the cache of the file with the fewest accesses will be terminated.

The implementation of a prototype as described in Nelson [2006] is accomplished using Erlang. The implementation for this effort will be in Java.

## Summary

The review of literature applicable to this assignment of implementing a document vending application, guided the focusing of the problem statement and goals to be on the testing of an implementation of the use of processes to implement a caching method where each datum is cached in a separate process.

# Chapter 3: Methodology

## Overview

The first part of the approach was to establish a test harness that would control a stable testing environment, with repeatable inputs to the system under test. The test harness launches the Document Server (the implementation of the file server) and then creates processes that are clients of the Document Server and request files from the server. The test harness reads a scenario file that lists out the files being requested by the clients in order. This allows the tests to be consistent in that the same files are requested each time, though it does not guarantee that the messaging will be consistent and that the files will be requested in the same order each time. The client processes may not establish their communications with the document server in a consistent order as the system the tests are running on may introduce delays.

The test harness also implements a logging function that receives log messages via IP sockets from each client process. Each client process will report events with timestamps so the operations of the system can be measured.

The test harness launches both the client processes and the document server process. The document server process launches a process to provide a data layer. The data layer accesses the files that will be served to the clients. The document server establishes a server that listens for connection requests from the clients. When a connection is established, a separate thread is launched to handle the request. Each client performs a connection and request to simulate obtaining a list of available files from the document server, there is then a pause of a random number of milliseconds before the client will request the desired file.

In the case of the prototype that does not have a cache, the file request to the document server results in the port number of the data layer server port that the data layer establishes when its process begins. The client then closes the connection to the document server and opens a connection with the data layer to request the file. The data layer then retrieves the file from disk and transfers the file to the client. Once the client receives the file, the client reports the event to the log server and terminates.

In the case of the prototype with the cache implementation, when the client requests the file from the document server, the list of files is read through to determine if the file has been cached in a cache process. If the file is in a cache, then the IP address and port number of the cache process is sent to the client. If the file is not in a cache, then the document server determines if a cache process needs to be terminated based on the number of cache processes currently operating. If there is space available for a new cache to begin, then a new cache process is launched with the filename of the file to cache. The document server must wait for the cache process to start and send back via an IP connection the port number that cache process has been assigned by the system. The document server can then send the port number to the client so the client can communicate with the cache process to receive the desired file.

## System Implementation

This prototype implementation was accomplished in the Java programming language without using third party libraries. Each part of the system communicated using TCP sockets available in Java, processes were launched using the ProcessBuilder class and several parts of the system implemented multiple threads of execution to handle TCP communications.

The overriding part of the prototype system is the test harness that is executed first and that causes the document server prototype to execute and launches client processes to interact with the document server. The entirety of the system is executed on the same machine, therefore there were certain limitations in the execution of the testing due to resource availability. More on that in the results section.

The test harness executes and reads from a scenario file that consists of a certain number of lines where each line contains a filename (that corresponds with a file in the document server) and a delay period. Both values are randomly selected using a scenario builder application. The list of available files and the scope of the randomized delay period can be set in the code for the scenario builder. The scenario builder then executes and produces a scenario file. This file can then be used to stimulate the testing of the prototype repeatedly and can be used to stimulate different prototype implementations. This consistency of stimulus can help to limit the differences between tests.

### Clients

Clients were implemented to be executed as separate processes and log into the document server via a TCP socket using a hard-coded IP address and port number (The IP address used by all the system components for their TCP sockets was ‘localhost’). When the test harness executes the client process, two values are included as arguments: filename and delay period. The filename is the name of the file the client will download from the document server. The delay period is a value used to simulate a user of the client taking time to decide on a file to download.

### Document Server

The document server is the main execution component of the prototype. For the purposes of testing, two prototypes were implemented, one prototype implements cache processes and the other does not, instead allowing direct communication from clients to the data layer to obtain the desired files. This difference allows for a comparison between the amount of time taken for the clients to receive files with or without the prototype cache system.

In both prototypes, the data layer is implemented in the same way and is executed as a separate process from the document server. The document server maintains a list of files stored on disk that the data layer retrieves and serves to clients. In the case of the prototype that does not do caching, the document server always provides the IP address and port number of the data layer to clients when they request a file from the document server. In the case of the prototype with caching and for the purposes of testing, the list of files is hard-coded into the document server, but the server creates a dynamic list in the form of a linked list from the hard-coded list of file names. The dynamic list maintains the data of which files are currently in a cache process and the number of times that filename is requested by clients.

In both cases, the timing of the request for a file by a client starts when the client sends a filename to the document server. The document server and the data layer both have hard coded port numbers to facilitate communication.

### Data Layer

The data layer is a separate application that operates to store the documents and provide those documents upon request. The data layer uses a process with multiple threads to handle the communication. The communication is conducted with TCP Sockets in Java.

In the case of the prototype with no caching, the clients are given the data layer’s IP address and port number and will establish communication with the data layer directly. In the case of the prototype that does have caching, the cache processes are launched by the document server and those processes request the desired file from the data layer. The data layer then transmits the file to the cache process where the file data is kept in active memory.

In both cases of requests for files, the data layer retrieves the file from disk and then transmits the file to the requester. The TCP transmission size available on the Java socket using the writeUTF method has a maximum size limit of around 65K bytes. Therefore, the files must be broken up before transmission. The requesting process will send a signal that a file is being requests, the data layer responds with a request for the filename, the requesting process sends the filename, then the data layer will get the file from disk, analyze it and break it into parts and will send the number of parts and the total file size to the requesting process. When the requesting process receives the file size and number of parts being transmitted, it prepares to receive the file.

In the case of the prototype without caching, the client is the one receiving the file and will prepare to receive and reconstruct the file into a string/text file. Once the file is received completely and reconstructed, the client logs the fact that the file has been received. This is the point when the timing of the request is finished in both prototypes. In the case of the prototype with caching, the cache process receives the information about the file being transmitted from the data layer and prepares to receive the file parts. The cache process does not reconstruct the file, instead the parts are kept separated in an array so when a request from a client is received, the data can immediately be transferred. The client operates the same in either prototype case and receives and reconstructs the file.

## Cache System

The cache implementation using processes to cache files being downloaded is the predominant area of investigation for this study. The cache is implemented as separate processes that are launched by the document server when the client requests a file. The document server manages the cache by maintaining a list of available files and if the file is in a cache, then the file information includes the IP address and port number of the cache process for that file. The number of cache processes the system can have active at one time is set in the code to a certain value and the document server also maintains a count of how many cache processes are active at any time.

When a file is requested, if the file is in cache, then the number of cache processes is checked against the maximum allowed processes. If there is room for more processes, then a new process is launched for a cache and the requested filename is sent as an argument to the process. If there is not room for another cache, then an existing cache must be terminated first. The cache to terminate is determined by traversing the linked list of files and identifying the file that currently has a cache process and has the lowest number of requests for the file. This cache replacement policy is unsophisticated but is used to accomplish the basic operation of the caching system.

Cache processes that are identified to be terminated are sent a termination signal via their TCP socket from the document server. The document server must then use the waitFor method on the process variable to receive the signal that the process has terminated. The whole operation of reading and changing the list of files, starting and terminating cache processes and updating the file list with access count values and the TCP address and port number of a new cache process is conducted during a lock of a MUTEX in the document server.

The benefit of simply terminating a process to clear out a cache is accomplished, but there is potential for a bottleneck that could last for a significant amount of time if the cache being terminated has already received multiple requests for the file and must complete those transmissions before terminating. The locking is necessary because of the potential that file transmissions could be occurring when changes to the cache process list is taking place. Therefore; the described benefit of concurrent starts and terminations of cache processes has not been achieved.

## Test Harness

The test harness is used to perform tests of the prototypes and apply the same input to both prototypes and for each test. The test harness reads a scenario file to determine what clients to start and when. Due to limitations of the workstation being used and the Java virtual machine settings, the tests of the prototypes were limited to about 80 clients. Greater than that number of clients in the scenario resulted in the JVM running out of memory or not being able to perform garbage collection.

To address that issue, a randomization of the starting times of the client processes was implemented to keep the test harness from trying to start all the client processes at the same time. With this in place, the simulation could hold 150 clients being executed by the test harness. Only around 20 client processes were active at the same time. As each test is performed, the activities of the client are logged and are written to a file by the test harness.

### Test procedures

Testing was conducted of the same scenario of 150 clients on the prototype with no cache five times. The same scenario was also conducted on the prototype with caching where the maximum number of cache processes was six, then 12, then 20 each test conducted five times. The total number of files in the system for testing purposes is 30, so the ratio of cached processes to total number of files got quite high.

After each test, the log file is analyzed by a separate program and converted from all the log entries of the client processes to a list of clients, the filename they requested, and the duration of time between first requesting the document from the document server and receiving the document.

# Chapter 4: Results

## Data analysis:

## Findings:

|  |  |  |
| --- | --- | --- |
| Client 0 | file1m5.txt | 321 |
| Client 1 | file1m5.txt | 39 |
| Client 5 | file1m5.txt | 51 |
| Client 3 | file1m5.txt | 35 |
| Client 4 | file1m5.txt | 48 |
| Client 2 | file1m5.txt | 39 |
| Client 6 | file1m5.txt | 28 |
| Client 8 | file1m5.txt | 38 |
| Client 7 | file1m5.txt | 48 |
| Client 9 | file1m5.txt | 54 |
|  |  | 701 |

|  |  |  |
| --- | --- | --- |
| Client 0 | file1m5.txt | 120 |
| Client 2 | file1m5.txt | 198 |
| Client 1 | file1m5.txt | 99 |
| Client 4 | file1m5.txt | 92 |
| Client 3 | file1m5.txt | 115 |
| Client 6 | file1m5.txt | 102 |
| Client 7 | file1m5.txt | 74 |
| Client 8 | file1m5.txt | 94 |
| Client 5 | file1m5.txt | 75 |
| Client 9 | file1m5.txt | 93 |
|  |  | 1062 |

## Summary of results:

# Conclusions

## Conclusions:

## Implications:

## Recommendations:

# References:

Koo, S. G. M., Rosenberg, C., Xu, D. (2003). Analysis of Parallel Downloading for Large File Distribution. *Proceedings of the The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems*, 128-135. doi: <https://doi.org/10.1109/FTDCS.2003.1204324>

Janakiraman, R., Xu, L. (2004). Efficient and Flexible Parallel Retrieval Using Priority Encoded Transmission. *NOSSDAV ’04 Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, 48-53. doi: <https://doi.org/10.1145/1005847.1005859>

Podlipnig, S., Böszörmenyi, L. (2003). A Survey of Web Cache Replacement Strategies. *ACM Computing Surveys Volume 35 Issue 4*, 374-398. doi: <https://doi.org/10.1145/954339.954341>

Rodriguez, P., Spanner, C., Biersack, E. W. (2001). Analysis of Web Caching Architectures: Hierarchical and Distributed Caching. *IEEE/ACM Transactions on Networking (TON) Volume 9 Issue 4*, 404-418. doi: <https://doi.org/10.1109/90.944339>

Nelson, J. (2006). Concurrent Caching. *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, 32-38. doi: <https://doi.org/10.1145/1159789.1159797>