MuStServ: A Lightweight Multimedia Streaming Server

By

Ahmad Suhaily Sulaiman

Dissertation submitted in partial fulfillment of
the requirements for the
Bachelor of Information Technology (Hons)
(Information and Communication Technology)

JAN 2006

Universiti Teknologi PETRONAS
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

MuStServ: A Lightweight Multimedia Streaming Server

By

Ahmad Suhaily Sulaiman

Dissertation Submitted to the Information Technology Programme
Universiti Teknologi PETRONAS
In partial fulfillment of the requirements for the
Bachelor of Information Technology (Hons)
(Information and Communication Technology)

Approved By,

(Mr. Low Tan Jung)

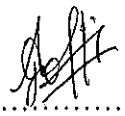
UNIVERSITI TEKNOLOGI PETRONAS

TRONOH, PERAK

Jan, 2006

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in the project, that the originality work is my own expect as specified in the references and acknowledgements and that the originality work contain herein have not been undertaken or done by unspecified sources or persons.



.....

(Ahmad Suhaily Sulaiman)

ABSTRACT

Media streaming is no longer a new term, more so after broadband internet connection became more of a necessity than luxury in most homes. The objective of this project is to come up with a streaming solution for multimedia files that utilizes as little system resources as possible and is not platform-dependent. To tackle this project, I followed a series of steps; namely research, designing the system architecture, develop the system, and test it. Python, JAVA, and PERL were analyzed to determine which programming language is the most suitable to be used. Those steps were then succeeded by choosing what transfer mode to be employed. As a result of the analysis, Python was decided the most suitable programming language to be used because it is easy to learn and fast to develop. HTTP streaming was the transfer mode of choice simply because it is easier to implement compared to UDP. Tests showed that MuStServ consumes significantly lower memory compared to Apache, the most widely used and easiest way to implement HTTP streamer. Thus, it can safely be concluded that this project has achieved its goal. The usage of efficient programming language, good architecture and also sheer simplicity in its design are the secrets why MuStServ is so lightweight.

Table of Contents

ABSTRACT 1
List of Tables 3
List of Figures 3
List of Abbreviations 3
1 INTRODUCTION 4
1.1 Background 4
1.2 Problem Statement 5
1.3 Objectives and Scope of Study 6
2 LITERATURE REVIEW AND / OR THEORY 7
2.1 HTTP Streaming 7
2.2 Server Load Balancing 9
2.2.1 Why is load balancing of servers needed? 9
2.2.2 About load balancing mechanism - IP Spraying 9
2.2.3 Types of load balancing 10
2.2.4 Methods of load balancing 11
3 METHODOLOGIES 13
3.1 Research 13
3.2 Design 15
3.3 Develop 15
3.4 Testing 15
4 RESULTS AND DISCUSSION 16
4.1 Java versus Python 16
4.2 Perl versus Python 18
4.3 Python Advantages 20
4.4 Apache versus MuStServ 22
4.5 MuStServ's System Architecture 27
4.5.1 Home Use Configuration 28
4.5.2 Large Scale Implementation 30
4.6 MuStServ's Simplified Flowcharts 31
5 CONCLUSIONS 32
6 REFERENCES 33

List of Tables

Table 4.1 JAVA vs. Python at First Glance 16
Table 4.2 Verbosity of Java vs. Conciseness of Python 18

List of Figures

Figure 3.1 VideoLAN Media Player User Interface 14
Figure 3.2 Netlimiter User Interface 14
Figure 4.1 Static Typing 17
Figure 4.2 Dynamic Typing 18
Figure 4.3 Sample Perl Code 19
Figure 4.4 Sample Python Code 20
Figure 4.5 Market Share for Top Servers across All Domains August 1995 - May 2006 23
Figure 4.6 Apache's CPU and Memory Usage with No Client (Idle) 24
Figure 4.7 MuStServ's CPU and Memory Usage with No Client (Idle) 25
Figure 4.8 MuStServ's Resource Usage with 10 Clients Watching DVD Quality Movies 26
Figure 4.9 MuStServ in Simple Home Network Configuration 28
Figure 4.10 MuStServ in Big Scale Network Configuration 30
Figure 4.11 Flowchart for Webserver Module 31
Figure 4.12 Flowchart for Streamer Module 31

List of Abbreviations

1. **MP3** – MPEG Layer 3 audio file format
2. **FTP** – File Transfer Protocol
3. **HTTP** – Hyper Text Transfer Protocol
4. **DNS** – Domain Name Server
5. **VLC** – VideoLAN Media Player
6. **CPU** – Central Processing Unit
7. **OS** – Operating System
8. **UDP** – User Datagram Protocol
9. **MBONE** – Multicast Backbone

1 INTRODUCTION

1.1 Background

There are two main ways to deliver multimedia files over the Internet: downloads or streaming media. In downloads, an audio file is stored on the user's computer. Compressed formats like MP3 are the most popular form of audio downloads, however take note that any type of audio file can be hosted through a Web or FTP site. Streaming audio is not stored, but only played on demand. It is a continuous broadcast that works through three software packages: the encoder, the server and the player. The encoder converts audio content into a streaming format, the server (repeater) makes it available over the Internet and the player retrieves the content. There are two types of well-known streaming media content; namely live feed, and on-demand stream. For a live broadcast, the encoder and streamer work together in real-time. An audio feed runs to the sound card of a computer running the encoder software at the broadcast location and the stream is uploaded to the repeater server. Since that requires a large amount of computing resources, the repeater must be a dedicated server. On-demand stream on the other hand normally doesn't need on-the-fly re-encoding. The client simply sends a request to the server, which in turn will prepare the content to be delivered. When everything is ready, a signal is then sent to let the client know that the content is ready to be pulled from the server.

The use of multimedia elements such as audio and video in a network environment has become something common to the users. Various formats of those files are being shared and streamed over the net almost every second. Up until now, there has been a lot of effort being put to organize a mean for a controllable environment for multimedia content streaming. This enables user to control and play audio and video files without having to download the whole file as this process allows data to be played incrementally, as the data fragments are received by the user, rather than waiting until the entire clip has been downloaded before playing it.

Audio and video files are being played in a continuous stream over the Internet. Streaming web files requires the use of plug-ins, which automatically decompress the files and play them in real time. Users can begin playing the file before the file's data has been completely transmitted. One way of streaming these files is to put them on a web server and let the client access and play the files in real-time. The client machine will save these files in the buffer and then run it from there. Normally, setting up a streaming server requires you to setup separate servers for source and the actual streamer. All of these servers require quite an amount of resources in terms of memory and CPU time. In meeting this demand, I have decided to propose to develop a hybrid web-based application to serve these files and let clients have full control of what they want to listen or watch. It allows clients to access their multimedia collection from any networked computer. It will stream their files via HTTP to any media player that supports playing off a remote connection (e.g. VLC, Winamp, FreeAmp, Sonique, XMMS). The small resource requirement of this program will enable even the most out-dated PCs to become a very powerful streaming server.

1.2 Problem Statement

As discussed above, setting up an audio streaming server includes setting up a source server-cum-encoder, and a streamer (repeater). Re-encoding audio files on-the-fly is by no means an easy task. This process utilizes very high memory and CPU time, which is why a dedicated server is normally needed for this task. The previous statement is also true for the streamer. However, it is not a norm for home users to have a spare Sun® Fire E25K™ Server lying around the house that can be used as their media server for the whole household. More often than not, these spare machines (if they are lucky enough to have one) are low-specs computers with maybe a Pentium III™ or a Pentium II™ processor and 128MB of RAM or less.

Many of the readily-available streaming solutions today are Operating Systems-dependent. Take Unreal Media Servers for example, they only run on Microsoft Windows® Operating System. The same goes to Peerfish.

By running a streaming server on Windows is compounding the problem of resources requirement because the OS itself is quite resource-hungry. For instance, it is rather unacceptable to run a Windows 2000 OS on a Pentium II™ machine with only 64MB of RAM, let alone running a streaming server on top of it. In contrast to that, if the OS used is either Linux or BSD, this should not be a problem. Not only these UNIX-descendents are generally considered to be more stable and secure, but if installed without GUI will obviously use much less resources than Microsoft Windows, which makes it a perfect platform to run a streaming server on a home network.

1.3 Objectives and Scope of Study

The objective of this project is to provide an elegant and flexible yet simple and extremely lightweight media streaming solution to cater for the needs of home users and commercial clients alike. For a home user, a lightweight streaming server means they can put their otherwise obsolete old computers to good use, rather than letting it collect dust in the store-room. As for businesses, this means they can support more clients with the same hardware they already have, which may very likely be translated to more profit.

However, given that only a limited amount of time is available to research and develop this project, more focus and effort will be put into making sure the server-side and connections modules work flawlessly. It is essential to ensure these modules are stable enough and as lightweight as possible to stand up to daily use and abuse. Basic security features will be included along with basic web interface for clients.

2 LITERATURE REVIEW AND / OR THEORY

Efficient multimedia delivery over public networks is a challenge for modern technology. Insufficient bandwidth, network latency, paranoid firewall restrictions and many more obstacles make it very difficult to transfer streaming content to end users, especially in real-time. Existing Media servers, such as Microsoft and Real Networks ones, only partially cover growing demand for streaming quality [1]. Sure, the usage of Multicasting would most probably overcome the insufficient bandwidth limitation and look very appealing at first, but two interesting points should be taken into consideration. First, for a home user bandwidth limitation is not a problem. Nowadays most wired home networks have 100Mbps bandwidth ready to be exploited. For wireless networks, they have at least 11Mbps, which in reality is actually already abundant. Secondly, without utilizing the likes of MBONE, UDP packets can not be routed through public data networks.

By definition, HTTP Streaming is a slightly more advanced method compared to HTTP downloading by *embedding* the file in a web page using special HTML code [2]. Delivering video files this way is known as HTTP streaming or HTTP delivery. HTTP means Hyper Text Transfer Protocol, and is the same protocol used to deliver web pages. For this reason it is easy to set up and use on almost any webserver, without requiring additional software or special hosting plans. The sheer simplicity and ease of implementation for this transfer mode along with very low cost incurred makes it very desirable to be chosen as the transfer mode for this project.

2.1 HTTP Streaming

This is the simplest and cheapest way to stream video from a website. Small to medium-sized websites are more likely to use this method than the more expensive streaming servers.

For this method you don't need any special type of website or host — just a host server which recognizes common video file types (most standard hosting accounts do this). You also need to know how to upload files and how to create hyperlinks.

There are some limitations to bear in mind regarding HTTP streaming:

* HTTP streaming is a good option for websites with modest traffic, i.e. less than about a dozen people viewing at the same time. For heavier traffic a more serious streaming solution should be considered.

* You can't stream live video, since the HTTP method only works with complete files stored on the server.

* You can't automatically detect the end user's connection speed using HTTP. If you want to create different versions for different speeds, you need to create a separate file for each speed.

* HTTP streaming is not as efficient as other methods and will incur a heavier server load.

These things won't bother most website producers — it's normally only when you get into heavy traffic that you should be worried about them. [2]

2.2 Server Load Balancing

Load balancing improves network performance by distributing traffic efficiently so that individual servers are not overwhelmed by sudden fluctuations in activity.

2.2.1 Why is load balancing of servers needed?

If there is only one web server responding to all the incoming HTTP requests for a website, the capacity of the web server may not be able to handle high volumes of incoming traffic once the website becomes popular. The website's pages will load slowly as some of the users will have to wait until the web server is free to process their requests. The increase in traffic and connections to your website can lead to a point where upgrading the server hardware will no longer be cost effective.

In order to achieve web server scalability, more servers need to be added to distribute the load among the group of servers, which is also known as a server cluster. The load distribution among these servers is known as load balancing. Load balancing applies to all types of servers (application server, database server), however, we will be devoting this section for load balancing of web servers (HTTP server) only.

2.2.2 About load balancing mechanism - IP Spraying

When multiple web servers are present in a server group, the HTTP traffic needs to be evenly distributed among the servers. In the process, these servers must appear as one web server to the web client, for example an internet browser. The load balancing mechanism used for spreading HTTP requests is known as IP Spraying. The equipment used for IP spraying is also called the 'load dispatcher' or 'network dispatcher' or simply, the 'load balancer'. In this case, the IP sprayer intercepts each HTTP request, and redirects them to a server in the server cluster. Depending on the type of sprayer involved, the architecture can provide scalability, load balancing and failover requirements.

2.2.3 Types of load balancing

Load balancing of servers by an IP sprayer can be implemented in different ways. These methods of load balancing can be set up in the load balancer based on available load balancing types. There are various algorithms used to distribute the load among the available servers.

Random Allocation

In a random allocation, the HTTP requests are assigned to any server picked randomly among the group of servers. In such a case, one of the servers may be assigned many more requests to process, while the other servers are sitting idle. However, on average, each server gets its share of the load due to the random selection.

Pros: Simple to implement.

Cons: Can lead to overloading of one server while under-utilization of others.

Round-Robin Allocation

In a round-robin algorithm, the IP sprayer assigns the requests to a list of the servers on a rotating basis. The first request is allocated to a server picked randomly from the group, so that if more than one IP sprayer is involved, not all the first requests go to the same server. For the subsequent requests, the IP sprayer follows the circular order to redirect the request. Once a server is assigned a request, the server is moved to the end of the list. This keeps the servers equally assigned.

Pros: Better than random allocation because the requests are equally divided among the available servers in an orderly fashion.

Cons: Round robin algorithm is not enough for load balancing based on processing overhead required and if the server specifications are not identical to each other in the server group.

Weighted Round-Robin Allocation

Weighted Round-Robin is an advanced version of the round-robin that eliminates the deficiencies of the plain round robin algorithm. In case of a weighted round-robin, one can assign a weight to each server in the group so that if one server is capable of handling twice as much load as the other, the powerful server gets a weight of 2. In such cases, the IP sprayer will assign two requests to the powerful server for each request assigned to the weaker one.

Pros: Takes care of the capacity of the servers in the group.

Cons: Does not consider the advanced load balancing requirements such as processing times for each individual request.

The configuration of a load balancing software or hardware should be decided on the particular requirement. For example, if the website wants to load balance servers for static HTML pages or light database driven dynamic web pages, round robin will be sufficient. However, if some of the requests take longer than the others to process, then advanced load balancing algorithms are used. The load balancer should be able to provide intelligent monitoring to distribute the load, directing them to the servers that are capable of handling them better than the others in the cluster of server.

2.2.4 Methods of load balancing

There are various ways in which load balancing can be achieved. The deciding factors for choosing one over the other depends on the requirement, available features, complexity of implementation, and cost. For example, using hardware load balancing equipment is very costly compared to the software version.

Round Robin DNS Load Balancing

The in-built round-robin feature of BIND of a DNS server can be used to load balance multiple web servers. It is one of the early adopted load balancing techniques to cycle through the IP addresses corresponding to a group of servers in a cluster.

Pros: Very simple, inexpensive and easy to implement.

Cons: The DNS server does not have any knowledge of the server availability and will continue to point to an unavailable server. It can only differentiate by IP address, but not by server port. The IP address can also be cached by other nameservers and requests may not be sent to the load balancing DNS server.

Hardware Load Balancing

Hardware load balancers can route TCP/IP packets to various servers in a cluster. These types of load balancers are often found to provide a robust topology with high availability, but come with a much higher cost.

Pros: Uses circuit level network gateway to route traffic.

Cons: Higher costs compared to software versions.

Software Load Balancing

Most commonly used load balancers are software based, and often comes as an integrated component of expensive web server and application server software packages.

Pros: Cheaper than hardware load balancers. More configurable based on requirements. Can incorporate intelligent routing based on multiple input parameters.

Cons: Need to provide additional hardware to isolate the load balancer.

3 METHODOLOGIES

To tackle this project, I followed a series of steps; namely research, designing the system architecture, develop the system, and test it.

3.1 Research

At this stage, I started with identifying the programming language to be used. Since one of the objectives of this project is to provide a cross-platform streaming solution, so obviously a programming language that is supported on multiple platforms must be used. For this, three candidates have been chosen; Java, Perl, and Python. All three programming languages were studied in detail and compared side by side.

The aforementioned step was then followed by analyzing which protocol is suitable to be used for the media transfer. For this step, Videolan Media Player and Netlimiter were used extensively to test the performance of each protocol. However, the most important trait that I was looking for was ease of implementation. This is so because my focus is to efficiently deliver the media files using as little memory and processing power as possible.

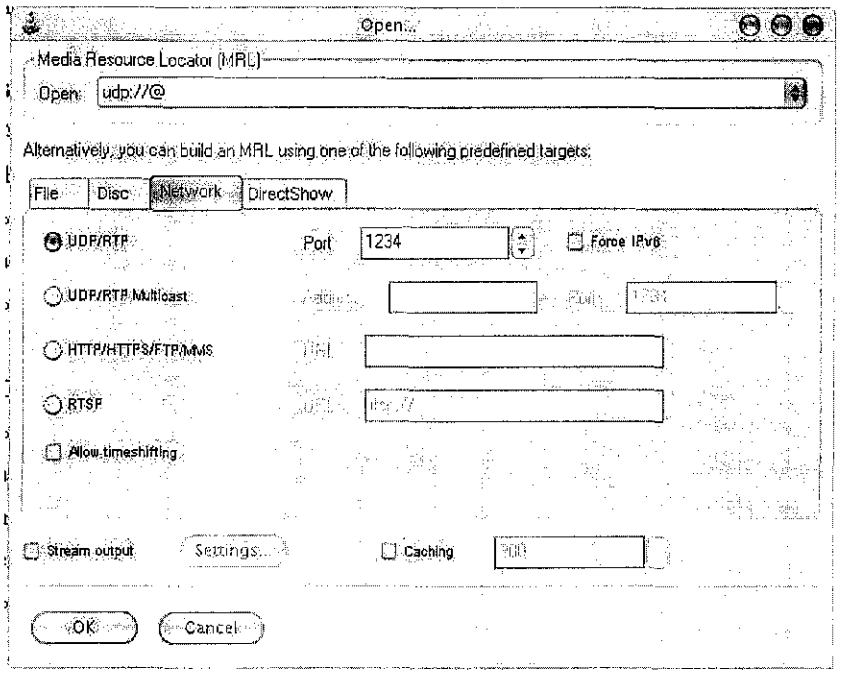


Figure 3.1 **VideoLAN Media Player User Interface**

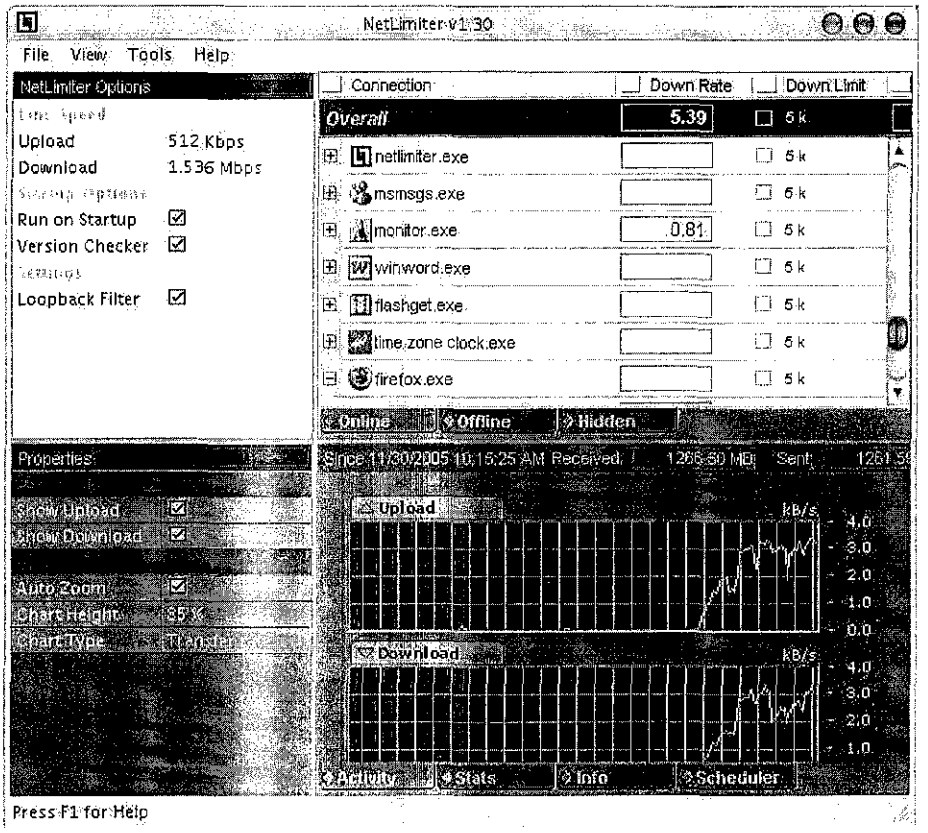


Figure 3.2 **Netlimiter User Interface**

Next, a study of currently available products was done to get an idea of how they work and what techniques can be integrated into this project. Three well-known media streaming products were analyzed, namely Nullsoft Shoutcast, Icecast, and Unreal Media Server. By examining the internal workings of these software, I get to know the best architecture to be used in order to make MuStServ as flexible as possible.

The final part I needed to find out is a media client that best suits MuStServ. Not all clients are the same. Some clients are more suited to high bandwidth media streaming, while others are tuned for lightweight-ness. Again, in choosing the best player, I had to look back at the objectives outlined earlier and made a choice based on these objectives.

3.2 Design

My design was based on the findings of the research phase, with the defining features of MuStServ well integrated into it. Characteristics like simple, scalable, and efficient are among the qualities that set MuStServ apart from any other multimedia streaming solutions.

3.3 Develop

In the development phase, small prototypes were constructed to validate the viability of MuStServ's design. Once validated, the entire system was hand-coded. Module by module, MuStServ started to materialize. As the modules were developed, constant testing was done to ensure each module is optimized for performance. This is important, since the ultimate goal of this project is to produce a system that utilizes as little memory and as low CPU usage as possible.

3.4 Testing

With all the modules ready, the system was then tested as a whole. Further configuration tweaking was done to achieve the objectives of this project. We shall all see in the Results and Discussion chapter how MuStServ fares to Apache, a very popular webserver daemon when put side by side.

4 RESULTS AND DISCUSSION

4.1 Java versus Python

Table 4.1 JAVA vs. Python at First Glance

JAVA	PYTHON
▪ Objects are better than functions ▪ Libraries should be written in JAVA for portability ▪ Fast execution requires hand-coded optimization ▪ Language controlled by one company behaving as a single entity	▪ Objects are as useful as functions as well as classes ▪ Strives to work well with libraries written in different languages ▪ Fast execution requires good algorithm ▪ Language controlled by community

Python and Java make for particularly interesting contrasts and combinations. Both are Object Oriented language, but however, as I researched more on both languages, I noticed one very prominent dissimilarity; Java's attitude has many absolutes: objects are better than functions, all differences between platforms should be abstracted away, libraries should always be written in Java for portability, fast execution requires hand-coded optimization, and it is best that one company has absolute control over the language so it doesn't become fragmented. Python's attitude is quite different: functions are just as useful as classes, cross-platform compatibility is important but not all platforms are alike, bridging to libraries written in other languages is important, fast execution requires good algorithms, and the language should be controlled by culture rather than licensing. I think Python's advantages over Java are easy to understand once you see the difference in attitude.

Java is a statically-typed language. In a statically typed language, every variable name is bound both to a type (at compile time, by means of a data declaration) and to an object.

The binding to an object is optional; if a name is not bound to an object, the name is said to be null. Once a variable name has been bound to a type (that is, declared) it can be bound (via an assignment statement) only to objects of that type; it cannot ever be bound to an object of a different type. An attempt to bind the name to an object of the wrong type will raise a type exception. In Java, all variable names (along with their types) must be explicitly declared. Attempting to assign an object of the wrong type to a variable name triggers a type exception.

Java container objects (e.g. Vector and ArrayList) hold objects of the generic type Object, but cannot hold primitives such as int. To store an int in a Vector, you must first convert the int to an Integer. When you retrieve an object from a container, it doesn't remember its type, and must be explicitly cast to the desired type.

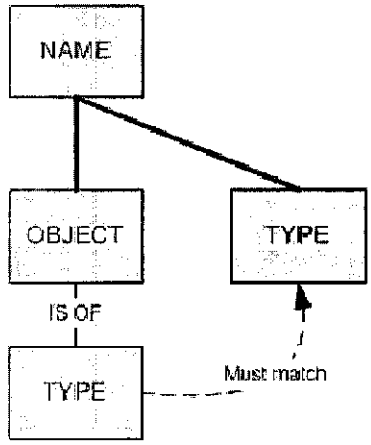


Figure 4.1 **Static Typing**

Python, on the other hand is a dynamically typed language. In a dynamically typed language, every variable name is (unless it is null) bound only to an object. Names are bound to objects at execution time by means of assignment statements, and it is possible to bind a name to objects of different types during the execution of the program. In Python, you never declare anything. An assignment statement binds a name to an object, and the object can be of any type. If a name is assigned to an object of one type, it may later be assigned to an object of a different type.

Python container objects (e.g. lists and dictionaries) can hold objects of any type, including numbers and lists. When you retrieve an object from a container, it remembers its type, so no casting is required.

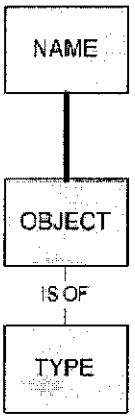


Figure 4.2 Dynamic Typing

Java is verbose, while Python is concise. **Figure 4.3** shows a classic example of the "Hello, world" program written in both Java and Python. As we can see, it is clear that Java is unnecessarily verbose. It requires the use of too many words which makes a simple task a lot more complicated than it really is. For the same reason, Python codes are generally and almost always faster than Java codes [3].

Table 4.2 **Verbosity of Java vs. Conciseness of Python**

Java	Python

`public class HelloWorld`
`{`
`  public static void main (String[] args)`
`  {`
`  System.out.println("Hello, world!");`
`  }`
`}`
	`print "Hello, world!"`

4.2 Perl versus Python

Perl and Python are scripting languages. While being very useful, Perl's framework design is not as good as Python's. For example, one of the first things I discovered I didn't like was the syntax. It's very complex and there are lots of operators and special syntaxes. This means that you get short, but complex code with many syntax errors that

will take some time to sort out. It also means that reading someone else's code is difficult. You can't easily understand someone else's scripts and adapt them to your own needs, nor can you easily take over the maintenance of someone else's program.

```
#This function produces a Soundex value from a name or word

sub soundex
{
  local (@s, $f, $fc, $_) = @_;

  push @s, '' unless @s;     # handle no args as a single empty string

  foreach (@s)
  {
    tr/a-z/A-Z/;
    tr/A-Z//cd;

    if ($_ eq '')
    {
      $_ = $soundex_nocode;
    }
    else
    {
      ($f) = /^(.)/;
      tr/AEHIOUWYBFPVCGJKQSXZDTLMNR/00000000111122222222334556/;
      ($fc) = /^(.)/;
      s/^$fc+//;
      tr///cs;
      tr/0//d;
      $_ = $f . $_ . '000';
      s/^(.{4}).*/$1/;
    }
  }

  wantarray ? @s : shift @s;
}
}
}
```

Figure 4.3 **Sample Perl Code**

Python codes, on the other hand, are highly readable. It has a simple visual layout, uses English keywords frequently where other languages use punctuation, and has notably fewer syntactic constructions than many structured languages such as C, Perl, or Pascal. For instance, Python has only two structured loop forms:

1. for item in iterator:, which loops over elements of a list or iterator
2. while expression:, which loops as long as a boolean expression is true.

It thus forgoes the more complex, C-style for (initialize; end condition; increment) syntax (common in many popular languages) And it does not have any of the common

alternative loop syntaxes such as do...while, repeat until, etc. though of course equivalents can be expressed. Likewise, it has only if...elif...else for branching - no switch or labeled goto [4]. Syntactical significance of indentation also makes a lot of sense to help make Python codes more readable.

```
#This function produces a Soundex value from a name or word

def soundex(string):
    """Returns the Soundex code of the string. Characters not A-Z skipped."""
    string=lower(string)

    if not is_letter(string[0]): string=string[1:]
    last=no_tbl[ord(string[0])-97]
    res =upper(string[0]) # This is where the result will end up

    for char in string[1:]:
          if is_letter(char):
              new=no_tbl[ord(char)-97]
              if (new!="0" and new!=last):
                    res=res+new
              last=new

    if len(res)<4:
          return res+"0"*(4-len(res))
    else:
          return res[:4]
```

Figure 4.4 **Sample Python Code**

4.3 Python Advantages

Python programs are far quicker to develop than other high-level languages. Because of the elegance and simplicity of the language, Python programs tend to be 3-5 times shorter than their equivalent in Java, and 5-10 times shorter than C++ equivalents [3]. In addition, being an interpreted language eliminates the lengthy edit-compile-debug cycle of other high-level languages. Since programmer costs are the most significant part of any programming project, this advantage alone provides a good reason to consider using Python.

Python programs are easier to maintain. The simple, clean syntax not only allows the original developers to remember what they did, it also allows other developers to understand and change programs. This allows for much lower maintenance costs for Python programs.

Python is an extremely versatile language. It can be used for the simplest scripting applications, as well as for the development of complex websites, and all the way up to the construction of complex distributed applications. This versatility allows organizations to reduce training and software tool costs.

Python's object-oriented paradigm is the most powerful and easy to use of any commercial programming language. Like Smalltalk, Python has dynamic typing and binding, and everything in Python is an object. Object behavior can be adapted at run-time. This allows Python programmers to easily use the full, flexible power of object-orientation, as opposed to Java or C++. However, Python's syntax is more like other standard programming languages, so it is much easier for programmers to learn than Smalltalk. Again powerful simplicity allows for rapid development of more complex applications.

Python is available on an incredibly wide range of hardware and software platforms. This includes the usual suspects: Sun, Intel, IBM, Microsoft Windows variants, Macintosh OS variants and all UNIX flavors. However, Python has also found its way into a wide range of less well-known platforms, including PDAs and set-top boxes. Python is probably the only language where one can truly say, write once and deploy everywhere.

Python plays well with other languages. Python programs can be extended using C, C++, or Java, or can be embedded in programs written in these languages. Jython is an implementation of Python written in 100% Pure Java, which allows you to run Python on any Java platform. There is also a Python extension that supports Microsoft's .NET Framework™ Common Language Runtime (CLR). Hence Python code can be used with any web services variant offered by competing vendors. Moreover, in those parts of applications where speed is of the essence, code can be written in the most efficient programming language available, while still retaining the rapid application development

advantages of Python. Finally, Python can be used as a wrapper for legacy applications written in other languages.

Python plays well with programming standards. Many high-quality Python extensions are available which support almost all Internet standards, CORBA, COM, SOAP, XML, XML-RPC and so on.

Python provides the option for integration with low-level APIs for both the Windows and UNIX platforms, allowing applications to be highly platform compatible. Java, by contrast, only gives the option for developing to the lowest common denominator across platforms. C# only gives the option to be Microsoft compatible.

4.4 Apache versus MuStServ

Apache is a well known and widely used webserver. One of Apache's very nice features that relates to MuStServ is its ability to do HTTP Streaming of media files. HTTP Streaming allows clients to play the files before it has even finished downloading. While this is similar to what MuStServ hopes to achieve, it presents one problem. Apache handles the HTTP streams like any other file downloads. It allows the file to be transmitted as fast as the bandwidth would permit. This explains why a streaming solution with Apache webserver as its front-end should not be used in big scale implementation as the server's bandwidth usage will be maxed out with just a few clients connected.

I have decided to make MuStServ handle this obstacle proactively. When a client requests for a file, MuStServ first analyses the file to determine its maximum bit-rate (if the file uses variable bit rate encoding). It then uses this value as the maximum speed a client can download the file from the server. This effectively helps to manage the server's bandwidth usage and in effect, more clients can be supported.

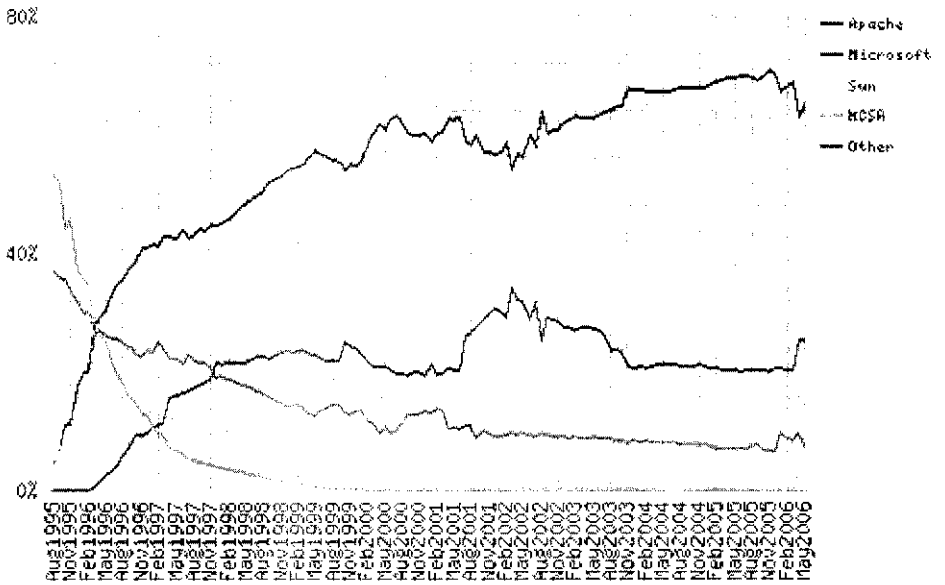


Figure 4.5 **Market Share for Top Servers across All Domains August 1995 - May 2006**

Apache is not a bad webserver at all. According to Netcraft and as we can see in **Figure 4.5,** Apache is in fact the most widely used webserver on the Internet [10]. It carries a long list of useful features which includes implementation of the latest protocols and extensible modules which means it can support PERL, PHP, SSL and many more via the use of mod_perl, mod_php, and mod_ssl respectively. While all these bells and whistles make Apache an excellent webserver, it also means Apache is not as lightweight as is needed to achieve the objectives of this project.

By simply making use of Python's readily available BaseHTTPServer and SocketServer modules, MuStServ has managed to be lean and includes only features that are really essential for a streaming server. This was purposely done to help keep the CPU and memory usage as low as possible. As we can see in **Figures 4.6 and 4.7** below, MuStServ's CPU and memory usage is a lot lower than Apache's and thus has achieved its goal.

Apache spawns two instances at any one time. In this example, Apache uses 4192K + 4984K = 9176K of RAM while MuStServ only uses 184K of RAM when idle. With ten clients connected, all watching DVD quality movies compressed using DivX codec, only 2516K of RAM was used (refer **Figure 4.8**). Of course, this number may vary according to the codec and rip quality of the movie. It also depends on the browsing activity done at that time by the clients.

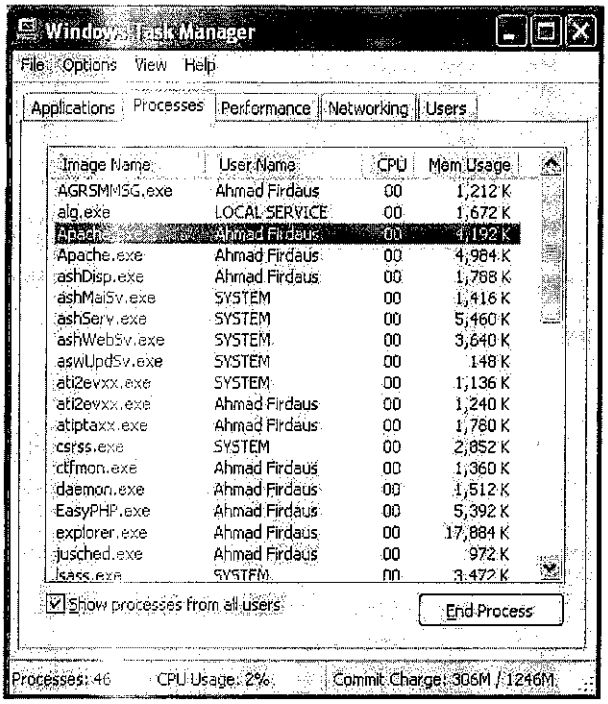


Figure 4.6 **Apache's CPU and Memory Usage with No Client (Idle)**

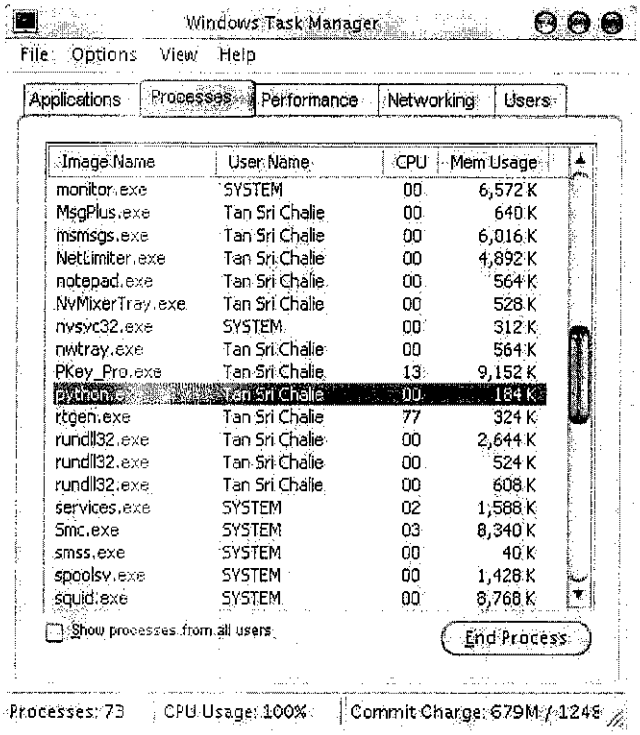


Figure 4.7 **MuStServ's CPU and Memory Usage with No Client (Idle)**

MuStServ's ability to run while utilizing as little system resources as possible relies heavily on its two main ingredients, which are BaseHTTPServer and SocketServer. BaseHTTPServer defines two classes for implementing HTTP servers (Web servers). Usually, this module isn't used directly, but is used as a basis for building functioning Web servers. The first class, `HTTPServer`, is a `SocketServer.TCPServer` subclass. It creates and listens at the HTTP socket, dispatching the requests to a handler. Code to create and run the server looks like this:

```
def run(server_class=BaseHTTPServer.HTTPServer,
        handler_class=BaseHTTPServer.BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

HTTPServer class builds on the TCPServer class by storing the server address as instance variables named server_name and server_port. The server is accessible by the handler, typically through the handler's server instance variable.

The second class, known as BaseHTTPRequestHandler is used to handle the HTTP requests that arrive at the server. By itself, it cannot respond to any actual HTTP requests; it must be subclassed to handle each request method (e.g. GET or POST). BaseHTTPRequestHandler provides a number of class and instance variables, and methods for use by subclasses.

The handler will parse the request and the headers, then call a method specific to the request type. The method name is constructed from the request. For example, for the request method "STREAM", the do_STREAM() method will be called with no arguments. All of the relevant information is stored in instance variables of the handler. Subclasses should not need to override or extend the __init__() method.

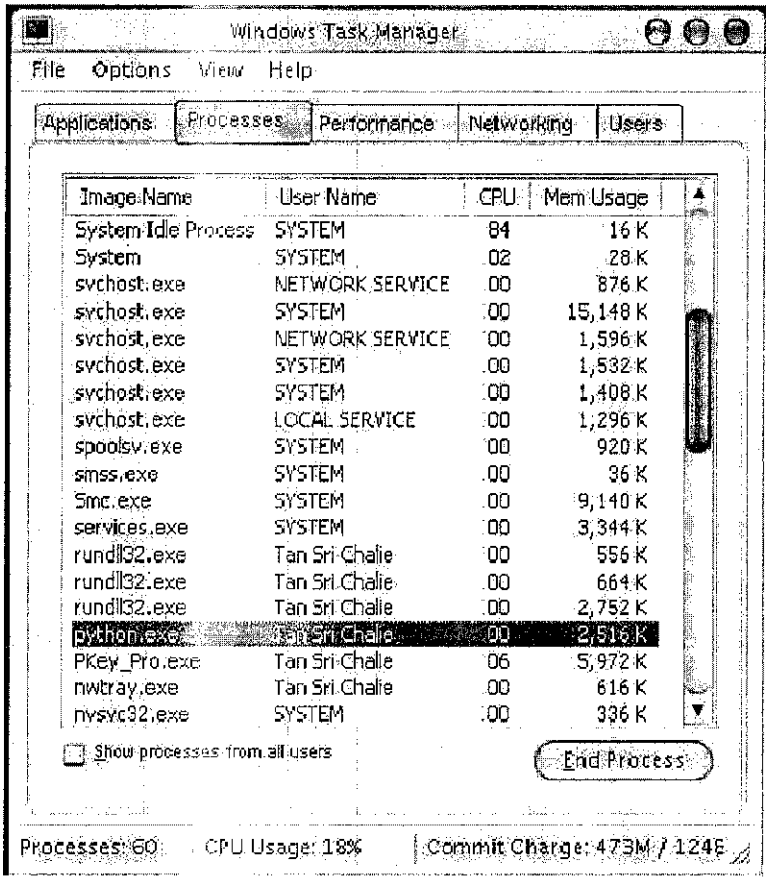


Figure 4.8 **MuStServ's Resource Usage with 10 Clients Watching DVD Quality Movies**

The SocketServer module simplifies the task of writing network servers. There are four basic server classes: TCPServer uses the Internet TCP protocol, which provides for continuous streams of data between the client and server. UDPServer uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The more infrequently used UnixStreamServer and UnixDatagramServer classes are similar, but use UNIX domain sockets; they're not available on non-Unix platforms. For the sake of portability across multiple platforms, the last two classes were not even considered to be used.

These four classes process requests synchronously; each request must be completed before the next request can be started. This isn't suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process. The solution is to create a separate process or thread to handle each request; the ForkingMixIn and ThreadingMixIn mix-in classes can be used to support asynchronous behavior.

4.5 MuStServ's System Architecture

MuStServ was designed to be flexible and scalable. With slight change in network configuration, MuStServ can be optimized for either home use or big-scale commercial implementation with many users. Theoretically, the limit is the server's bandwidth.

4.5.1 Home Use Configuration

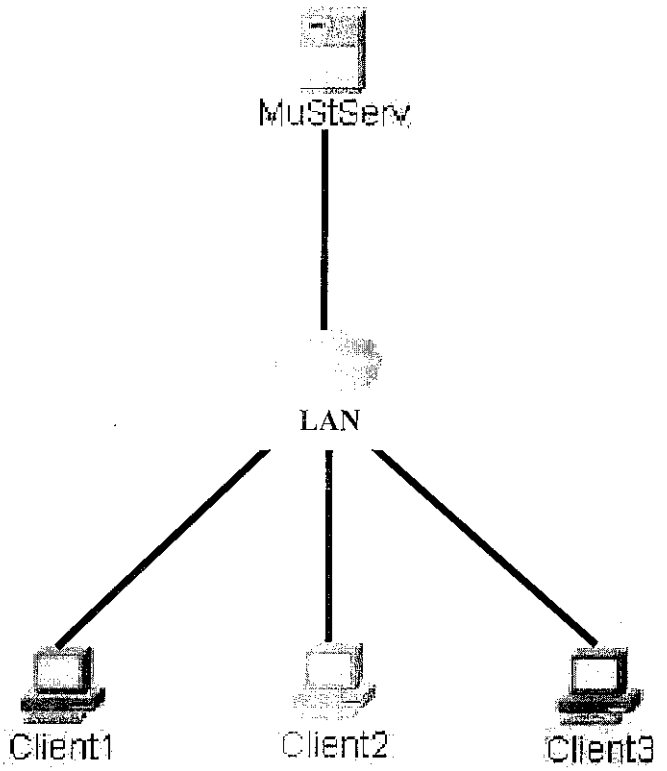


Figure 4.9 **MuStServ in Simple Home Network Configuration**

In a simple network configuration used in most home network, MuStServ fits in perfectly. One PC could be used to host MuStServ and the media files. The clients then connect to the Local Area Network using a switch or hub and then request the media files directly from MuStServ. Obviously, if the computers in the household are to be connected wirelessly, then wireless access point(s) should be used appropriately. Theoretically, 802.11b's 11Mbps connection speed should be able to cater the bandwidth needed for normal home streaming usage. However, it is recommended that the machine hosting MuStServ to be connected to the wireless access point using wired connection so that the bottleneck will not be on the relatively slow 11Mbps connection between MuStServ and the access point.

For more advanced home users, a dedicated file server could be used to store all the media files. In this configuration, MuStServ will function as proxy to the file server. If the user decides to add more file servers later on, only minimal change need to be done in MuStServ's text-based configuration file. The clients will then not even be aware that the media files are hosted on different machines.

For users who have a Mediacenter PC or Home Theater PC (HTPC) in their living room, storage size is a common problem that limits the amount of movies and TV series they can watch on their expensive home theater setup. MuStServ could very well solve this problem by serving as gateway to their large multimedia files collection stored on other machine(s) with bigger storage space.

4.5.2 Large Scale Implementation

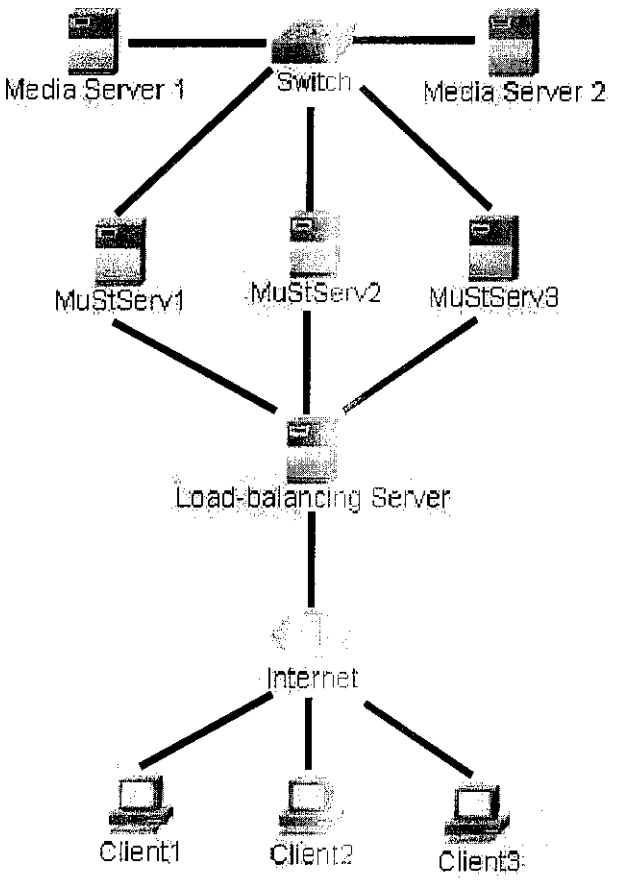


Figure 4.10 **MuStServ in Big Scale Network Configuration**

For large scale implementation, the usage of load balancing technique is important. The load-balancing server determines new requests will be handled by which MuStServ server. This will effectively spread the load among available MuStServ servers and as a result, no one server will be overloaded. One of the load-balancing techniques that can be used for this purpose is by using the round-robin dns. This can be done by simply entering multiple IP in the A NAME entry of the DNS.

MuStServ recognizes mapped network drives. With that said, the easiest way to implement the media servers is by simply using the readily available folder

sharing feature in Windows. However, if the media server is to be run on Linux, then the same objective can be achieved by using the Samba daemon which should be included by default in almost all distributions, including the most basic minimalist ones.

4.6 MuStServ's Simplified Flowcharts

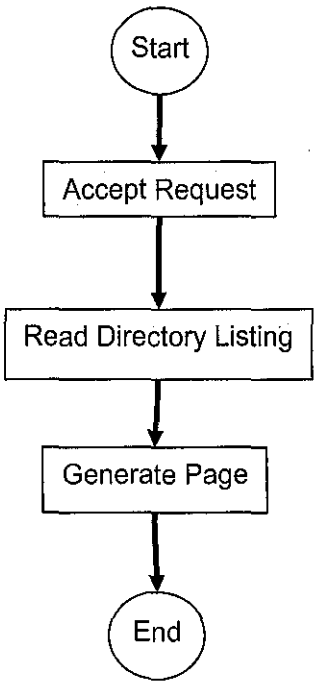


Figure 4.11 **Flowchart for Webserver Module**

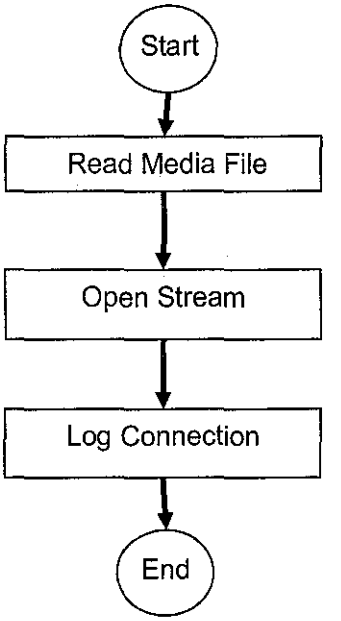


Figure 4.12 **Flowchart for Streamer Module**

5 CONCLUSIONS

Based on the proofs in **4.4**, I believe I can safely say that the objectives of this project have been achieved. The usage of efficient programming language, good architecture and also sheer simplicity in its design are the secrets why MuStServ is so lightweight.

Python truly shined as a real Rapid Application Development programming language in this project. It is very flexible, and easy to learn. Simple design reduces room for mistakes and thus contributes to the system's stability, a trait that is a must for every daemon.

As of now, MuStServ is far from perfect. There are a lot of enhancements that can be made to this project. For starters, a searching function would be a wonderful addition to this application. Users will then be able to quickly locate the files they want. This feature is especially helpful when a lot of files are being served by the server. Other recommendation would include prettier/more user friendly interface, better security feature as well as web-based configuration for administrators.

6 REFERENCES

Reference style is based of American Psychological Association's (APA) Style of Citations.

[1] Unreal Streaming Technologies Official Website. 20 Nov 2005
< http://www.umediaserver.net/overview.html>.

[2] Wavelength Media Website. 20 Nov 2005
< http://www.mediacollege.com/video/streaming/overview.html>

[3] Lefkowitz, Glyph (2000). *A subjective analysis of two high-level, object-oriented languages: Comparing Python to Java*

[4] Wikipedia entry of Python Programming Language. 28 Sept 2005
< http://en.wikipedia.org/wiki/Python_programming_language>

[5] Lutz, Mark (1996). *Programming Python,* CA: O'Reilly & Associates, Inc.

[6] Holden, Steve (2002). *Python Web Programming,* Sams Publishing.

[7] Beazley, D. M. and Rossum, G. V. (2001). *Python Essential Reference*, Sams Publishing.

[8] Martelli, Alex. (2003). *Python in a Nutshell,* O'Reilly & Associates, Inc.

[9] Dawson, Michael (2003). *Python Programming for the Absolute Beginner,* Thomson Course Technology.

[10] Netcraft Web Server Survey Archives. 09 May 2006
<http://www.netcraft.com/survey/>