REAL-TIME MULTI-NODE DATA TRANSMISSION: DESIGNING A LOW LATENCY NETWORK TV

OGUNTADE TEMITOPE .P.

(2006 / 24378EE)

DEPARTMENT OF ELECTRICAL/COMPUTER ENGINEERING, FEDERAL UNIVERSITY OF TECHNOLOGY, MINNA

OCTOBER, 2010

DEDICATION

This project work is dedicated to God Almighty, my beloved parents, and my supportive relatives.

Declaration

I Oguntade Philip Temitope declares that this work was done by me and has never been presented elsewhere for the award of a degree. I also hereby relinquish the copyright to the Federal University of Technology, Minna.

Oguntade Philip Temitope

(Name of student)

15-11-2010

(Signature and Date)

ENGR A.G. RAJI

(Name of H.O.D)

(Jan. 11, 2011)

(Signature and Date)

Mr. Tola Omokhafe

(Name of Supervisor)

15-11-2010

(Signature and Date)

Engr Dr. G.I. [illegible]

(Name of External Examiner)

1/12/10

(Signature and Date)

ACKNOWLEDGEMENTS

I wish to express my appreciation to my project supervisor, Mr. Tola Omokhafe for successfully guiding me through my project work. Also I wish to recognize the effort of Dr. B.A. Oluwade in making sure I become a better engineer and scientist.

My gratitude goes to Ogunlana Olushola and Sholis Studios, Olofu Mark Olofu, Iroiso Ikpokonte, the Contest Programming Team and all the friends of T.jar

To my school families: Mrs Shittu and family, Oyewole Kayode, Odumosu Olamide, Sowumi Temitope and Tunji.

ABSTRACT

The aim of this project is to create a model for the distribution of data in computer networks such that almost every computer system on the network participates in the consumption and re-distribution of this data and no computer system is over-loaded in the process.

The project presents a network design that will offer an extremely low latency that is required in implementation. Also, it creates a new algorithmic pattern in streaming real-time data to an unlimited number of subscribers. And this ensures that all systems receive the same data at every instance so that no user consumes the data before others.

The completion of this project has demonstrated the advantages of distributing data in a balanced dynamic tree pattern and its usage spans data replication and synchronisation in databases, application server clustering and real-time multimedia broadcast: although this work focuses its use primarily in the latter.

Table of Contents

CHAPTER ONE

INTRODUCTION

1.1 Project Background

The deployment of high-performance servers for real-time broadcast has become a norm in our computing societies. These servers offer the required optimum concurrency needed in serving all subscribing client computer systems real-time data. For instance, in order to deploy a server to stream real-time multimedia data to client systems we may have to consider a number of factors: these includes the number of concurrent client systems being connected and the rate of data-on-demand from each client system.

Having known this, it will almost be impossible to broadcast multimedia data to numerous client systems from a single personal computer system due to the fact that this arrangement will either remarkably slow down transmission or hang up the server system. The purpose of this work is to find an optimum approach around this challenge.

Todd (2007) observed that routers, by default, break up a broadcast domain - the set of all devices on a network segment that hear all the broadcasts sent on that segment.

In order to offer the needed familiarity with this thesis, we shall be enumerating the technical terms to be used in context.

A **hub** is a network device that places all network devices attached to it on the same collision and broadcast domain. Hence, all attached devices will compete for the same bandwidth whenever data is to be transmitted or received. In addition, whenever a device is transmitting data, all other devices will be forced to listen(half-duplex).

A **switch** is a network device that places all network devices attached to it on separate collision domains but same broadcast domain. Consequently, all devices connected to the switch will receive all broadcast from the switch. A demerit of this is that if there are too many hosts on the switch, it will flood each attached device with all broadcast; hereby increasing congestion especially in a low-bandwidth environment.

A **router** is a network device that places attached networks on separate collision and broadcast domains. This is of course the best of such devices we have access to. Do note that routers are actually switches, only that they are layer 3 switches.

Latency: Broadly, the time it takes a packet of data to travel from one location to another. In specificnetworking contexts, it can mean either (1) the time elapsed (delay) between the executionof a request for access to a network by a device and the time the mechanism actually is permittedtransmission, or (2) the time elapsed between when a mechanism receives a frame andthe time that frame is forwarded out of the destination port.

Broadcast domain: A group of devices receiving broadcast frames initiating from any device within the group. Because routers do not forward broadcast frames, broadcast domains arenot forwarded from one broadcast to another.

Broadcast: A data frame or packet that is transmitted to every node on the local network segment(as defined by the broadcast domain). Broadcasts are known by their broadcast address,which is a destination network and host address with all the bits turned on. Also called "local broadcast".

Collision domain: The network area in Ethernet over which frames that have collided will be detected. Collisions are propagated by hubs and repeaters, but not by LAN switches, routers,or bridges.

Collision: The effect of two nodes sending transmissions simultaneously in Ethernet. Whenthey meet on the physical media, the frames from each node collide and are damaged.

1.2 Objectives of the Study

This project hopes to achieve the following objectives:

- Create a network design that will offer the extremely low latency required in implementation.
- Create a new algorithmic pattern in streaming real-time data to an unlimited number of subscribers.
- Ensure that there is little or no breakdown in transmission of data.
- Ensure that under no circumstance will the server system be overloaded.
- Ensure that all systems receive the same data at every instance so that no user consumes the data before others.

1.3 Research Methodology

The network model to be employed in this research model shall be a wireless setup. Hence, our transmitting device(s) shall be directly connected to a wireless router. With this arrangement, we shall benefit from the router as discussed by Todd (2007). One of the

most important advantages is that the router will treat all attached devices as possible servers or clients(technically, *datasources or datasinks*). This means that communication will only be established between a requesting system and the requested system. Such communications will only involve an intermediary (most likely an authentication server) if a resource is required (such as authentication).

We shall also be building the software infrastructure on varying layers of operation. These layers shall include the application layer, the logic layer, and the data layer. The application layer shall be the layer representing the visual display of the Television. This layer receives media stream directly from the data layer which resides on different computer systems on the network. However, the reception of this service is been completely controlled by the logic layer. The logic layer describes all business decisions to be enforced in the distribution of data stream.

1.4 Scope

Given the probable complexity of the infrastructure to be developed, it was imperative to limit the scope of the work. The project scope is enumerated below:

- The software will be designed for use on the windows operating system
- TCP/IP is the de-facto communication protocol used on windows. Hence the windows operating system supports TCP/IP and wireless communication.
- Due to the universality of the Hypertext Transmission Protocol(HTTP), we shall be making use of RESTful web services in our implementation of the logic layer. This paradigm is very fast and scalable in order to meet up with the envisioned growth of devices watching TV.

1.5 Project Outline

This project is divided into five (5) chapters. Chapter one gives this introduction, chapter two contains a review of literature related to this project's area of study. The detailed design of the system to be developed is presented in chapter three using UML diagrams. implementation of the final requirements specification in Visual basic.Net, RESTful Web Services and Java. Chapter four discusses the test that were carried out and testing strategies that were adopted. Chapter five contains a summary of the work undertaken during the course of the project, the problems encountered, suggestions for further studies and a conclusion.

CHAPTER TWO

LITERATURE REVIEW

2.1 Review

The transmission of real-time data has been around the world for a very long period of time. It has been used in telephone data transmission, television and radio broadcast, walkie-talkie communication and in recent times web-broadcast. Organizations like youtube inc popularized the webcast technology.

Webcasting, distributing audio, video, and other media content in digital form using a computer network. Webcasting commonly refers to live radio and video programming that is sent over the Internet. The programming is listened to or viewed using a media player on a personal computer rather than a radio receiver or a television set. Unlike traditional radio, television, or cable broadcasting, however, the media content can also be pre-recorded or archived, and stored in a digital form that a user may access at any later time on-demand from a Web site. The content may be viewed as streaming audio or video that can be played as the data is sent. In other cases, the content may be in the form of digital files that the user can download to a computer and later transfer to a portable media device such as an MP3 player or a personal media player (PMP).

2.2 Technology

To listen to or view webcasts, the user needs a media player installed on a computer. Widely used media players include QuickTime, Windows Media Player, and RealMedia. Versions of these products can be downloaded from the Internet, sometimes for free. For best-quality results, the user may need broadband to receive high amounts of data in a

short time. The sender needs equipment and software to record or convert audio or video signals into compressed digital forms that can be streamed live or stored as files on a Web site.

Streaming allows a listener or viewer to experience live broadcasts as the content is received in small digital packets of data sent over the Internet. This process creates a slight delay as each new packet is encoded, sent, received, temporarily stored, and then played. Thus a streaming version of a radio broadcast will arrive after an over-the-air signal. Progressive downloading is a similar process that allows an archived or pre-recorded file to be played as is it is downloaded in small data packets. In this way users do not need to download an entire file to be able to listen to or watch the content.

2.3 Sockets

Sockets provide point-to-point communication. There are three major forms of socket communication: Unicast, Multicast and Broadcast. Unicast sockets create a connection with two well-defined endpoints; there is one sender and one receiver and, although they may switch roles, at any given time it is easy to tell which is which. However, although point-to-point communications serve many, if not most needs (people have engaged in one-on-one conversations for millennia), many tasks require a different model. For example, a television station broadcasts data from one location to every point within range of its transmitter. The signal reaches every television set, whether or not it's turned on and whether or not it's tuned to that particular station. Indeed, the signal even reaches homes with cable boxes instead of antennas and homes that don't have a television. This is the classic example of broadcasting. It's indiscriminate and quite wasteful of both the electromagnetic spectrum and power.

Video conferencing, by contrast, sends an audio-video feed to a select group of people. Usenet news is posted at one site and distributed around the world to hundreds of thousands of people. DNS router updates travel from the site, announcing a change to many other routers. However, the sender relies on the intermediate sites to copy and relay the message to downstream sites. The sender does not address its message to every host that will eventually receive it. These are examples of multicasting, although they're implemented with additional application layer protocols on top of TCP or UDP. These protocols require fairly detailed configuration and intervention by human beings. For instance, to join Usenet you have to find a site willing to send news to you and relay your outgoing news to the rest of the world. To add you to the Usenet feed, the news administrator of your news relay has to specifically add your site to their news config files. However, recent developments with the network software in most major operating systems as well as Internet routers have opened up a new possibility—true multicasting, in which the routers decide how to efficiently move a message to individual hosts. In particular, the initial router sends only one copy of the message to a router near the receiving hosts, which then makes multiple copies for different recipients at or closer to the destinations. Internet multicasting is built on top of UDP.

2.3.1 Multicast Sockets

Multicasting is broader than unicast, point-to-point communication but narrower and more targeted than broadcast communication. Multicasting sends data from one host to many different hosts, but not to everyone; the data only goes to clients that have expressed an interest by joining a particular multicast group. In a way, this is like a public meeting. People can come and go as they please, leaving when the discussion no longer interests them. Before they arrive and after they have left, they don't need to process the

information at all: it just doesn't reach them. On the Internet, such "public meetings" are best implemented using a multicast socket that sends a copy of the data to a location (or a group of locations) close to the parties that have declared an interest in the data. In the best case, the data is duplicated only when it reaches the local network serving the interested clients: the data crosses the Internet only once. More realistically, several identical copies of the data traverse the Internet; but, by carefully choosing the points at which the streams are duplicated, the load on the network is minimized. The good news is that programmers and network administrators aren't responsible for choosing the points where the data is duplicated or even for sending multiple copies; the Internet's routers handle all that.

IP also supports broadcasting, but the use of broadcasts is strictly limited. Protocols require broadcasts only when there is no alternative, and routers limit broadcasts to the local network or subnet, preventing broadcasts from reaching the Internet at large. Even a few small global broadcasts could bring the Internet to its knees. Broadcasting high-bandwidth data such as audio, video, or even text and still images is out of the question. A single email spam that goes to millions of addresses is bad enough. Imagine what would happen if a real-time video feed were copied to all six hundred million Internet users, whether they wanted to watch it or not.

However, there's a middle ground between point-to-point communications and broadcasts to the whole world. There's no reason to send a video feed to hosts that aren't interested in it; we need a technology that sends data to the hosts that want it, without bothering the rest of the world. One way to do this is to use many unicast streams. If 1,000 clients want to listen to a RealAudio broadcast, the data is sent a thousand times. This is inefficient, since it duplicates data needlessly, but it's orders-of-magnitude more efficient than broadcasting the data to every host on the Internet. Still, if the number of interested

clients is large enough, you will eventually run out of bandwidth or CPU power—probably sooner rather than later.

Another approach to the problem is to create static connection trees. This is the solution employed by Usenet news and some conferencing systems (notably CUseeMe). Data is fed from the originating site to other servers, which replicate it to still other servers, which eventually replicate it to clients. Each client connects to the nearest server. This is more efficient than sending everything to all interested clients via multiple unicasts, but the scheme is kludgy and beginning to show its age. New sites need to find a place to hook into the tree manually. The tree does not necessarily reflect the best possible topology at any one time, and servers still need to maintain many point-to-point connections to their clients, sending the same data to each one. It would be better to allow the routers in the Internet to dynamically determine the best possible routes for transmitting distributed information and to replicate data only when absolutely necessary. This is where multicasting comes in.

When people start talking about multicasting, audio and video are the first applications that come to mind; however, they are only the tip of the iceberg. Other possibilities include multiplayer games, distributed filesystems, massively parallel computing, multi-person conferencing, database replication, and more. Multicasting can be used to implement name services and directory services that don't require the client to know a server's address in advance; to look up a name, a host could multicast its request to some well-known address and wait until a response is received from the nearest server. Apple's Rendezvous (a.k.a. Zeroconf) and Sun's Jini both use IP multicasting to dynamically discover services on the local network.

Multicasting should also make it easier to implement various kinds of caching for the Internet, which will be important if the Net's population continues to grow faster than available bandwidth. Martin Hamilton has proposed using multicasting to build a distributed server system for the World Wide Web. ("Evaluating Resource Discovery Applications of IP Multicast", http://martinh.net/eval/eval.html, 1995.) For example, a high-traffic web server could be split across multiple machines, all of which share a single hostname, mapped to a multicast address. Suppose one machine chunks out HTML files, another handles images, and a third processes servlets. When a client makes a request to the multicast address, the request is sent to each of the three servers. When a server receives the request, it looks to see whether the client wants an HTML file, an image, or a servlet response. If the server can handle the request, it responds. Otherwise, the server ignores the request and lets the other servers process it. It is easy to imagine more complex divisions of labour between distributed servers.

Multicasting has been designed to fit into the Internet as seamlessly as possible. Most of the work is done by routers and should be transparent to application programmers. An application simply sends datagram packets to a multicast address, which isn't fundamentally different from any other IP address. The routers make sure the packet is delivered to all the hosts in the multicast group. The biggest problem is that multicast routers are not yet ubiquitous; therefore, you need to know enough about them to find out whether multicasting is supported on your network. As far as the application itself, you need to pay attention to an additional header field in the datagrams called the Time-To-Live (TTL) value. The TTL is the maximum number of routers that the datagram is allowed to cross; when it reaches the maximum, it is discarded. Multicasting uses the TTL as an ad hoc way to limit how far a packet can travel. For example, you don't want packets

for a friendly on-campus game of Dogfight reaching routers on the other side of the world. The figure below shows how TTLs limit a packet's spread.

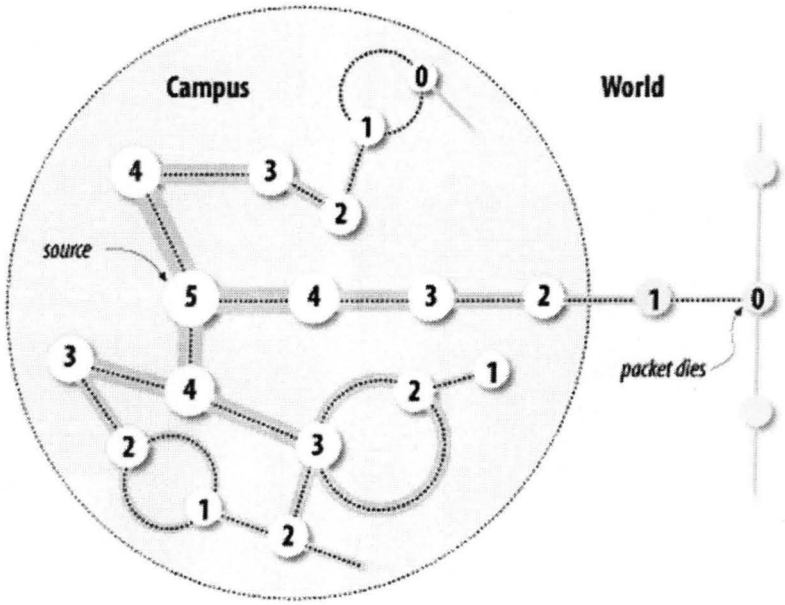


Fig 2.1 Coverage of a packet with a TTL of five

2.3.2 Multicast Addresses and Groups

A multicast address is the shared address of a group of hosts called a multicast group. We'll talk about the address first. Multicast addresses are IP addresses in the range 224.0.0.0 to 239.255.255.255. All addresses in this range have the binary digits 1110 as their first four bits. They are called Class D addresses to distinguish them from the more common Class A, B, and C addresses. Like any IP address, a multicast address can have a hostname; for example, the multicast address 224.0.1.1 (the address of the Network Time Protocol distributed service) is assigned the name ntp.mcast.net.

A multicast group is a set of Internet hosts that share a multicast address. Any data sent to the multicast address is relayed to all the members of the group. Membership in a multicast group is open; hosts can enter or leave the group at any time. Groups can be

either permanent or transient. Permanent groups have assigned addresses that remain constant, whether or not there are any members in the group. However, most multicast groups are transient and exist only as long as they have members. All you have to do to create a new multicast group is pick a random address from 225.0.0.0 to 238.255.255.255, construct an InetAddress object for that address, and start sending it data.

A number of multicast addresses have been set aside for special purposes. all-systems.mcast.net, 224.0.0.1, is a multicast group that includes all systems that support multicasting on the local subnet. This group is commonly used for local testing, as is experiment.mcast.net, 224.0.1.20. (There is no multicast address that sends data to all hosts on the Internet.) All addresses beginning with 224.0.0 (i.e., addresses from 224.0.0.0 to 224.0.0.255) are reserved for routing protocols and other low-level activities, such as gateway discovery and group membership reporting. Multicast routers never forward datagrams with destinations in this range.

2.4 Routers and Routing

The image below shows one of the simplest possible multicast configurations: a single server sending the same data to four clients served by the same router. A multicast socket sends one stream of data over the Internet to the clients' router; the router duplicates the stream and sends it to each of the clients. Without multicast sockets, the server would have to send four separate but identical streams of data to the router, which would route each stream to a client. Using the same stream to send the same data to multiple clients significantly reduces the bandwidth required on the Internet backbone.

Of course, real-world routes can be much more complex, involving multiple hierarchies of redundant routers. However, the goal of multicast sockets is simple: no

matter how complex the network, the same data should never be sent more than once over any given network segment. Fortunately, you don't need to worry about routing issues. Just create a MulticastSocket, have the socket join a multicast group, and stuff the address of the multicast group in the DatagramPacket you want to send. The routers and the MulticastSocket class in the respective programming language take care of the rest.

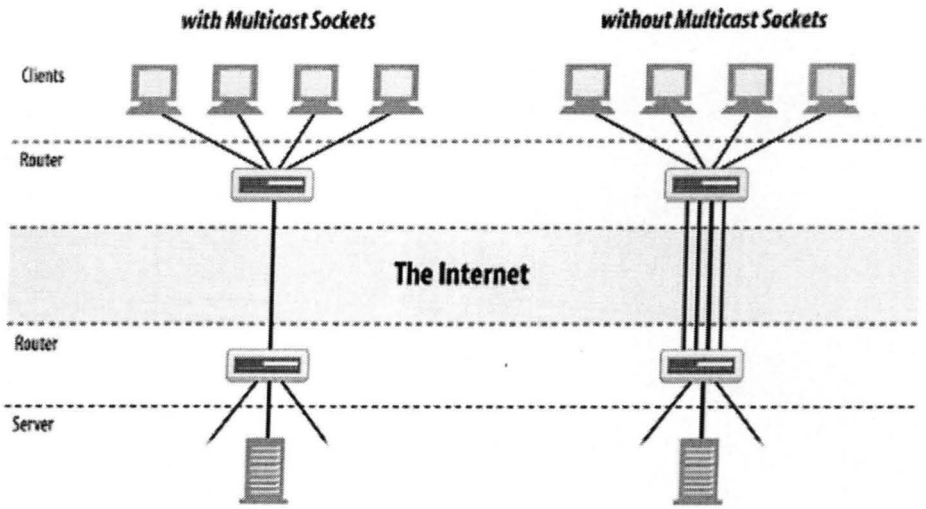


Fig 2.2 With and without multicast sockets

The biggest restriction on multicasting is the availability of special multicast routers (mrouters). Mrouters are reconfigured Internet routers or workstations that support the IP multicast extensions. Many consumer-oriented ISPs quite deliberately do not enable multicasting in their routers. In 2004, it is still possible to find hosts between which no multicast route exists (i.e., there is no route between the hosts that travels exclusively over mrouters).

To send and receive multicast data beyond the local subnet, you need a multicast router. You can also try pinging all-routers.mcast.net. If any router responds, then the network is hooked up to a multicast router:

% ping all-routers.mcast.net

This still may not allow you to send to or receive from every multicast-capable host on the Internet. For your packets to reach any given host, there must be a path of multicast-capable routers between your host and the remote host. Alternately, some sites may be connected by special multicast tunnel software that transmits multicast data over unicast UDP that all routers understand. If you have trouble getting the examples in this chapter to produce the expected results, check with your local network administrator or ISP to see whether multicasting is actually supported by your routers.

2.5 The UDP Protocol

Transmission Control Protocol (TCP) is designed for reliable transmission of data. If data is lost or damaged in transmission, TCP ensures that the data is resent; if packets of data arrive out of order, TCP puts them back in the correct order; if the data is coming too fast for the connection, TCP throttles the speed back so that packets won't be lost. A program never needs to worry about receiving data that is out of order or incorrect. However, this reliability comes at a price. That price is speed. Establishing and tearing down TCP connections can take a fair amount of time, particularly for protocols such as HTTP, which tend to require many short transmissions.

The User Datagram Protocol (UDP) is an alternative protocol for sending data over internet protocol (IP) that is very quick, but not reliable. That is, when you send UDP data, you have no way of knowing whether it arrived, much less whether different pieces of data arrived in the order in which you sent them. However, the pieces that do arrive generally arrive quickly.

The obvious question to ask is why anyone would ever use an unreliable protocol. Surely, if you have data worth sending, you care about whether the data arrives correctly?

Clearly, UDP isn't a good match for applications like FTP that require reliable transmission of data over potentially unreliable networks. However, there are many kinds of applications in which raw speed is more important than getting every bit right. For example, in real-time audio or video, lost or swapped packets of data simply appear as static. Static is tolerable, but awkward pauses in the audio stream, when TCP requests a retransmission or waits for a wayward packet to arrive, are unacceptable. In other applications, reliability tests can be implemented in the application layer. For example, if a client sends a short UDP request to a server, it may assume that the packet is lost if no response is returned within an established period of time; this is one way the Domain Name System (DNS) works. (DNS can also operate over TCP.) In fact, you could implement a reliable file transfer protocol using UDP, and many people have: Network File System (NFS), Trivial FTP (TFTP), and FSP, a more distant relative of FTP, all use UDP. (The latest version of NFS can use either UDP or TCP.) In these protocols, the application is responsible for reliability; UDP doesn't take care of it. That is, the application must handle missing or out-of-order packets. This is a lot of work, but there's no reason it can't be done.

The difference between TCP and UDP is often explained by analogy with the phone system and the post office. TCP is like the phone system. When you dial a number, the phone is answered and a connection is established between the two parties. As you talk, you know that the other party hears your words in the order in which you say them. If the phone is busy or no one answers, you find out right away. UDP, by contrast, is like the postal system. You send packets of mail to an address. Most of the letters arrive, but some may be lost on the way. The letters probably arrive in the order in which you sent them, but that's not guaranteed. The farther away you are from your recipient, the more likely it is that mail will be lost on the way or arrive out of order. If this is a problem, you can write

CHAPTER THREE

DESIGN AND IMPLEMENTATION

3.1 Design

All but the very simplest embedded systems now work in conjunction with a real-time operating system. A real-time operating system manages processes and resource allocation in a real-time system. It starts and stops processes so that stimuli can be handled and allocates memory and processor resources.

The components of a real-time operating system depend on the size and complexity of the real-time system being developed. For all except the simplest systems, they usually include:

1. A real-time clock: This provides information to schedule processes periodically.
2. An interrupt handler: This manages aperiodic requests for service.
3. A scheduler: This component is responsible for examining the processes that can be executed and choosing one of these for execution.
4. A resource manager: Given a process that is scheduled for execution, the resource manager allocates appropriate memory and processor resources.
5. A despatcher: This component is responsible for starting the execution of a process.

In the event of this project work, additional facilities such as disk storage management and fault management facilities, that detect and report system faults and a configuration manager that supports the dynamic reconfiguration of real-time applications.

sequential numbers on the envelopes, then ask the recipients to arrange them in the correct order and send you mail telling you which letters arrived so that you can resend any that didn't get there the first time. However, you and your correspondent need to agree on this protocol in advance. The post office will not do it for you.

Both the phone system and the post office have their uses. Although either one could be used for almost any communication, in some cases one is definitely superior to the other. The same is true of UDP and TCP.

2.6 Nash Equilibrium

In game theory, **Nash equilibrium** (named after John Forbes Nash, who proposed it) is a solution concept of a game involving two or more players, in which each player is assumed to know the equilibrium strategies of the other players, and no player has anything to gain by changing only his or her own strategy unilaterally. If each player has chosen a strategy and no player can benefit by changing his or her strategy while the other players keep theirs unchanged, then the current set of strategy choices and the corresponding payoffs constitute a Nash equilibrium.

Stated simply, Amy and Bill are in Nash equilibrium if Amy is making the best decision she can, taking into account Bill's decision, and Bill is making the best decision he can, taking into account Amy's decision. Likewise, a group of players is in Nash equilibrium if each one is making the best decision that he or she can, taking into account the decisions of the others. However, Nash equilibrium does not necessarily mean the best cumulative payoff for all the players involved; in many cases all the players might improve their payoffs if they could somehow agree on strategies different from the Nash equilibrium (e.g., competing businesses forming a cartel in order to increase their profits).

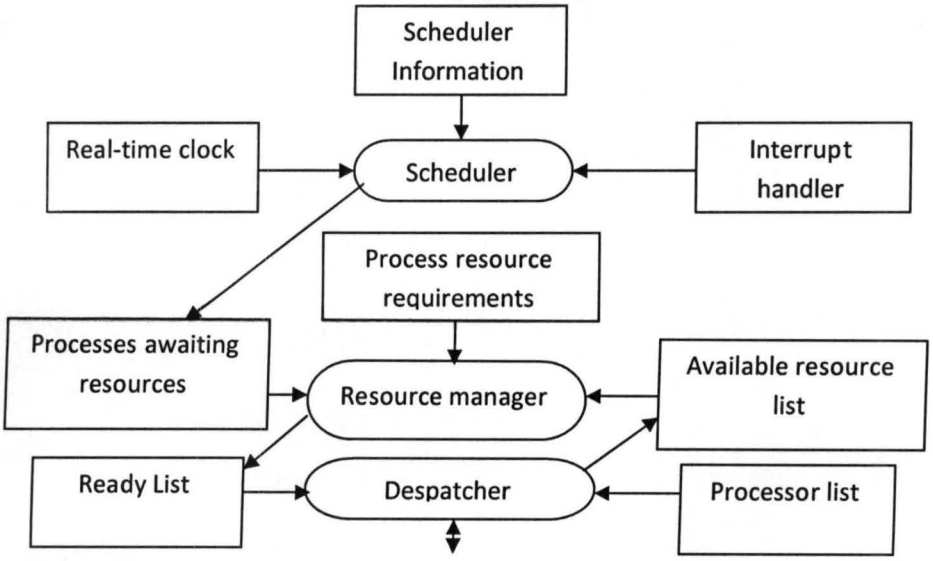


Fig 3.1: Components of a real-time operating system

3.2 Distribution Dynamics

Suppose there are *n* users hoping to listen to streams from the server, does that mean the system will broadcast to all *n* users? Definitely NO!

Using this unique approach, all listening devices can be placed on a queue in the order of their attempt to connect. Therefore, whenever the first subscriber i.e. (n-(n-1)) requests connection, he occupies the first block of the queue, the second user (n-(n-2)) occupies the second block and so on. With this pattern, the first user reads an amount of data from the server, saves it temporarily in memory and consumes it from the temporal storage. Whenever the second system request connection, he shall be granted permission to stream continuously from the first user's system. And the third user does same to the second user.

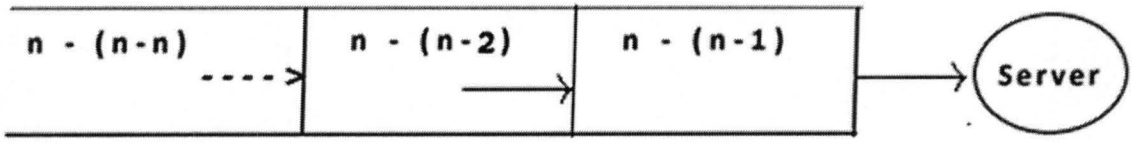


Figure 3.2: A linear based model for the distribution of data amongst client

Based on this linear approach, the server shall only be responsible for the first user's connection but all users' authentication.

3.3 A further review

As brilliant as this technique may appear, there are loop-holes that can completely bring this strategy to disrepute. Firstly, latency will be the in the order of the number of systems connected. And this means that there will be absolutely no hope of coherence in the distribution of TV broadcast or all the subscribing client systems will be forced to wait for a lengthy period of time to achieve this. Secondly, Let us consider a broadcast chain of 1000 users in the linear approach. If the computer system of the 10th subscriber goes out of service, it means the 990 dependents will be completely cut-off from broadcast.

These are of course major drawbacks of the linear approach. However, these can be significantly reduced by merely introducing the **Tree Approach**. Here, every user broadcasts to, on average, two additional users. This is represented in the diagram below:

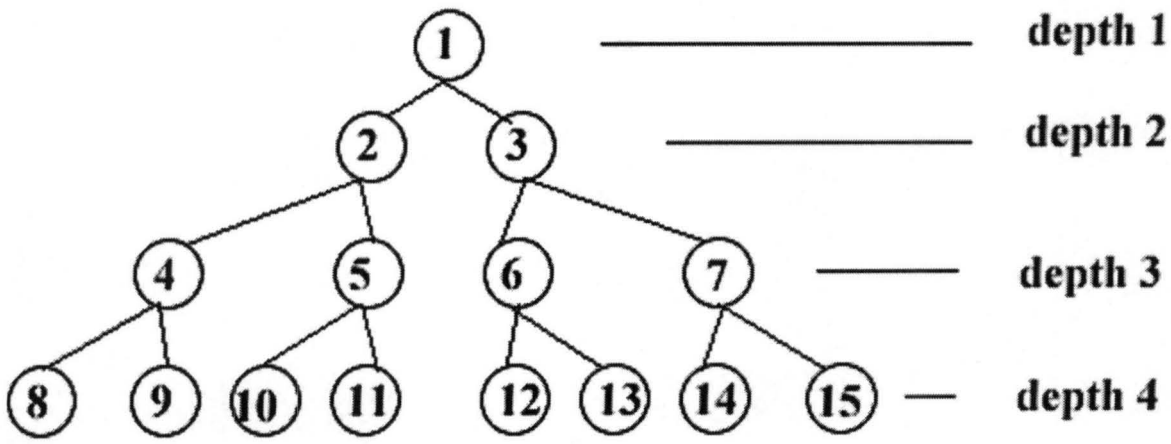


Figure 3.3: A tree based model for the distribution of data amongst client

This is obviously an enhanced form of the linear approach as each strand of the balanced tree shows linearity as in (1) → (2)→(4)→(8). However, the number of users that can be cut-off is inversely related to depth. The total number of depths in a 1000 user system is

$2^d - 1 = 1000$

$2^d = 1001$

$\log_{10} 2^d = \log_{10} 1001$

$d \log_{10} 2 = \log_{10} 1001$

$d = \log_{10} 1001 / \log_{10} 2$

$d = 3.000434 / 0.301029$

$d = 9.9673 \approx 10$

The 10th user can be found at the 4th depth with 7 other users. But between the 1st and the 4th depth, we have 15 users, leaving us with 985 users to evaluate. Since all 8 users at depth 4 will equally share the load of all additional users, the maximum number of users that can get disconnected as a result of the 10th user's failure is $985 / 8 \approx 123$ users. A big 867 users improvement! over the linear approach. Note that this value, shows more significant improvement as the depth increases.

The table below shows this fact.

Number of Users in the system	**Amount of disconnected systems due to 40th subscriber failure in Linear Approach**	**Amount of disconnected systems due to 40th subscriber failure in Tree Approach**
1000	960	29
10,000	9,960	310
100,000	99,960	3,123
1000,000	999,960	31,248
10,000,000	9,999,960	312,498

Table 3.1: Table showing number of systems disconnected at certain depths

One possible drawback of this approach is that broadcast can be negatively influenced if a remarkably slow computer system comes in-between. This can however, be avoided by setting a minimum system requirement.

3.4 Checkmating Error Situations

In any of the approaches, whether 1000 or 10 subscribers disconnects, the goal is to ensure their restoration without loss of data or transmission time. But how can this be ensured? This can simply be ensured in 3 steps. By

1. Monitoring the amount of bytes read by each system
2. Determining which parent node disconnects
3. Then making the node with the highest *bytes read*(among the failed systems), the new parent node so that all others will then re-attach to it.

This process however, assumes that every system on the network has read an amount of byte sufficient to be consumed all through the shutdown and handshaking process (period between loss and regain of connection).

3.5 Yet Another Review

There is yet another challenge. Let us imagine that we are broadcasting a football match and 8 people shouts "its a goal" at time t followed by 16 people at time $t+x$ and 32 at time $t+2x$ and 64 at time $t+3x$ and so on. This means that our broadcast of the football match has no longer been delivered under the pretence of live cast. If the network latency between any two systems is 100 milliseconds, then in the tree approach it means the number of depth will be the amount of milliseconds difference between the first and the last subscribers. This means that in a 30 depth arrangement, the last set of subscribers will lag the topmost subscriber by about 3 seconds. What if we can firstly, delay our transmission by say, 5 seconds in order for all subscribers to read sufficient amount of

bytes and secondly, determine the latency of the network. With these knowledge, we can ensure that a particular block of data gets to every user before any user can watch it. But how can we?

We can use the corrector equation below

$$t = (\sum_{d=2}^{d=n} l / \quad f) \quad \text{(standard measurement in milliseconds)}$$

where t = time to play

l = network latency between any two depths

d = (depth position of the last system in consideration)

f = d - 1

n = number of depths

There's a word of caution here! Firstly, we cannot determine the latency of the first depth position: latency starts from the second depth position. However, the above formula will compute the time of play for almost every depth position including the first. Secondly, the users at the last depth do not need compute this time of play because this equation assumes the last depth as the zeroth frame of reference. Hence, users at this location should consume data once received.

- Time to play (t) is the amount of time a subscriber should wait before consuming a resource. In this regard, it means that every system will compute its own time to start watching the TV content. And this time increases quite insignificantly down the tree path.

3.6 Implementation

3.6.1 Process Management

Real-time systems have to handle external events quickly and, in some cases, meet deadline for processing these events. This means that the event-handling processes must be scheduled for execution in time to detect the event and must be allocated sufficient processor resources to meet their deadline. The process manager in an RTOS is responsible for choosing processes for execution, allocating processor and memory resources and starting and stopping processes execution on a processor.

The Java code snippets below shows the different segments enumerated in section **3.1**

```
/**
* lets every PC check if there's an update in the system
*/
static int current = 0;

/**
* lets the server set its IP
*/
static String serverIP = "";

/**
* gets the current count so that systems can know if there's a
* change in the system.
* Infact, client systems should request this at once in five
seconds
*/
public int getCurrent() {
return current;
}

/**
* gets the IP of the server so that clients can know whom to ping
*to periodically measure their performance.
* Infact, client systems should perform this operation at start-up
*/
public String getServerIP() {
return serverIP;
}

/**
* return the IPs (max of 2) that are to be fed by the supplied IP
```

```
*/
public String getReceivers(String ip)
{
String receiver1 ="";String receiver2 ="";

if(smart.length-1 >= 2*extractIpIndex(ip)+1)
receiver1 = String.valueOf(smart[2*extractIpIndex(ip)+1].getIp());
if(smart.length-1 >= 2*extractIpIndex(ip)+2)
receiver2 = String.valueOf(smart[2*extractIpIndex(ip)+2].getIp());
return receiver1+"*.*"+receiver2;
}

/**
* return the IP of the system that serves this IP streaming media
*/
public String getGiver(String ip)
{
String giver1 ="";
if(extractIpIndex(ip) <= 4 &&extractIpIndex(ip) >= 0)
giver1 = getServerIP();
else    if(smart.length-1    >=    (extractIpIndex(ip)-1)/2    &&
extractIpIndex(ip) >= 0)
giver1 = String.valueOf(smart[(extractIpIndex(ip)-1)/2].getIp());

return giver1;
}

/**
* Searches for the index of the IP so that the giver and receivers
*can be computed. Returns -1 if not found.
*  This algorithm uses a linear search pattern.  It is quite
* inefficient for large data
*/
public int extractIpIndex(String ip)
{
int k = -1;
for(int i = 0;i<smart.length;i++)
{
if(smart[i].getIp().equals(ip))
k = i;
}
return k;
}
```

The process manager has to manage processes with different priorities. For some stimuli, such as those associated with certain exceptional events, it is essential that their processing should be completed within the specified time limits. Other processes may be delayed if a

more critical process requires service. Consequently, the ROTS has to be able to manage at least two priority levels for system processes:

1. Interrupt level: This is the highest priority level. It is allocated to processes that need a very fast response. One of these processes will be the real-time clock process.
2. Clock level: This level of priority is allocated to periodic processes.

In the implementation of this project work, the Java code snippet below shows a part of the process manager that handles interrupt handling.

```
/**
 * removes a malfunctioning system from distribution then
*re-shuffules the remaining clients based on the applied
algorithm.
*/
public boolean removeIP(String ip)
{
booleanbool = false;
Smart sm;
int k = -1;
for(int i = 0;i<smart.length;i++)
 {
if(smart[i].getIp().equals(ip))
    {
sm = smart[i];
set.remove(sm);
Smart[] smm = new Smart[set.size()];
Iterator iter = set.iterator();
for(int ii=0;ii<=smm.length-1;ii++)
    {
smm[ii] = (Smart) iter.next();
                }
smart = sl.mergeSort(smm);
current++;
bool = true;}}
return bool; }
```

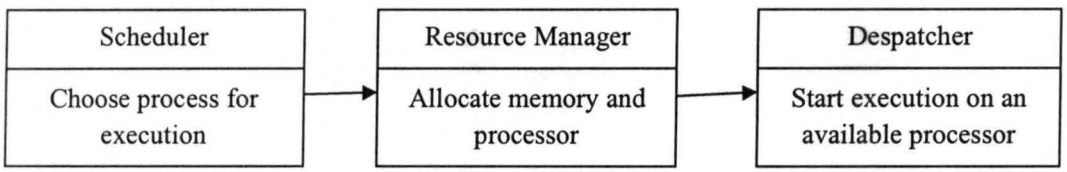


Fig 3.4: RTOS actions required to start a process

There may be a further priority level allocated to background processes (such as a self-checking process) that do not need to meet real-time deadlines. These processes are scheduled for execution when processor capacity is available. Within each of these priority levels, different classes of process may be allocated different priorities. For example, there may be several interrupt lines. An interrupt from a very fast device may have to pre-empt processing of an interrupt from a slower device to avoid information loss. The allocation of process priorities so that all processors are serviced in time usually requires extensive analysis and simulation.

Periodic processes are processes that must be executed at specified time intervals for data acquisition and actuator control. In most real-time systems, there will be several types of periodic processes. These will have different periods (the time between process executions), execution times and deadlines (the time by which processing must complete). Using the timing requirements specified in the application program, the RTOS arranges the execution of periodic processes so that they can all meet their deadlines.

3.6.2 Priority Handling

In order to actualize a stable distribution paradigm, several time-delay and interrupt factors were examined. Among these includes memory availability to process multimedia request (client side), network latency, and processor availability (client side). In the course of this project design, it was concluded that network latency is of a higher relevance for today's computing hardware.

Having considered the real-time nature of the project, the distribution model uses merge-sort in arranging client systems according to their current performance state.Merge sort is an example of a divide-and-conquer algorithm. In such an algorithm, we divide the data

into smaller pieces, recursively conquer each piece, and then combine the results into a final result.

An implementation of the merge sort algorithm in the Java language as it is used in the project is shown below

```
public class SortLatency {

public static Smart[] mergeSort(Smart[] data) {
return mergeSortHelper(data, 0, data.length - 1);
 }

protected static Smart[] mergeSortHelper(Smart[] data, int bottom,
int top) {
if (bottom == top) {
return new Smart[] { data[bottom] };
 } else {
int midpoint = (top + bottom) / 2;
return merge(mergeSortHelper(data, bottom,
midpoint),mergeSortHelper(data, midpoint + 1, top));
 }
}

 /**
  * Combine the two sorted arrays a and b into one sorted array.
  */
protected static Smart[] merge(Smart[] a, Smart[] b) {
Smart[] result = new Smart[a.length + b.length];
int i = 0;
int j = 0;
for (int k = 0; k <result.length; k++) {
if ((j == b.length) || ((i <a.length) && (a[i].getLatency() <=
b[j].getLatency()))) {
result[k] = a[i];
i++;
     } else {
result[k] = b[j];
j++;
     }
   }
return result;
 }

}
```

3.6.3 Determining latency

Latency can be programmatically determined by sending a byte of data to a system at a certain depth following all connecting nodes in the tree connection[3]. The value of the time taken to send and receive the byte forms the round-trip latency. This value should be computed over a range and the average divided by 2 should be the useful latency. The code snippet below shows the Java language representation of this task.

```
InetAddress in = InetAddress.getByName("targetName");
long startTime = System.currentTimeMillis();
boolean bool = in.isReachable(5000);
long endTime = System.currentTimeMillis();
long latency = endTime-startTime;
if(bool)
System.out.println("Target is reachable, latency is: "+latency);
else
System.out.println("System is unreachable, timed-out after 5
seconds");
```

CHAPTER FOUR

TESTS, RESULTS AND DISCUSSION

4.1 Tests

The server and clients systems were tested for response time and stability. In the case of response time, the following system configuration was used to conduct the test.

Component	Quantity or Size
Number of Processors	2
Processor Speed	2.1Ghz each
RAM Size	2 GB Average

Other Parameters

Network Mode	**Peer-to-peer over wireless LAN**
Distance(s) between systems	Average is lesser than 20 metres
Expected Multimedia network throughput	150kbps (average)

Having built the data transmission system over socket connection (point-to-point connection), data was transmitted from the server down the nodes as they move down the balanced binary tree.

Also, the holistic system behavior was studied under individual system failure.

4.2 Results

After transmitting data in a depth-first traversal pattern it was realized that the response time was on average, 0 milliseconds. This value shows a remarkable improvement over the anticipated latency (response time) of 100 milliseconds as predicted

in section **3.5**. In addition, a break between failover and re-transmission was noticed whenever there is a breakdown in any client system.

The diagrams below shows what happens if there is a failure in any of the systems.

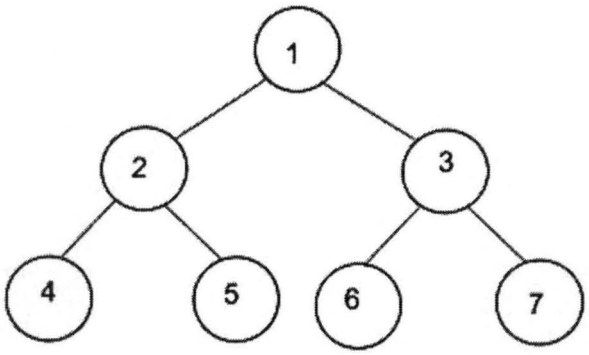


Fig 4.1: Diagram showing a 7 user system

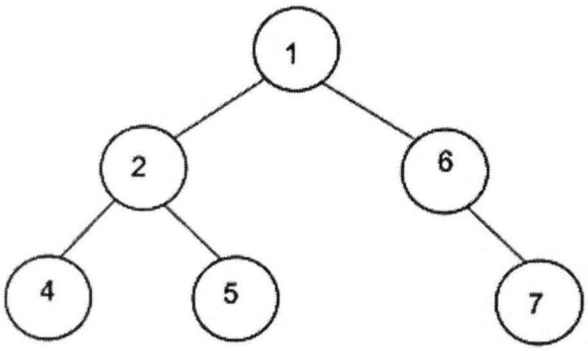


Fig 4.2: A recovery model assuming system 6 is performing better than 7

4.3 Discussion

It is strongly recommended that the 100 milliseconds delay be implemented before feeding any client system media data. This recommendation may be ignored except of course, there has been a major improvement in hardware and network technologies.

In the case of the noticed break in display, the system can be significantly enhanced by buffering data on client side for the period of reception of broadcast.

CHAPTER FIVE

CONCLUSION

5.1 Conclusion

With all these in place, the system has definitely been trained so as to behave optimally in such terrible load conditions. In addition, this design will be able to broadcast to an infinite amount of subscribers. All feeding from one personal computer.And as technology advances and computing facilities gets more sophisticated, this work becomes more meaningful: such that it becomes a network paradigm.

5.2 Problems Encountered

1. It was at first very challenging to converge all sub-systems of this algorithm in order to form this relatively new data distribution model.
2. The time-to-play predictor equation was not readily available and it had to be necessary to be derived before equality in display could be achieved.
3. As it has been emphasized in this work that there are several delay factors; it was quite time consuming before it could be finally established that latency could be positioned in such a way as to represent them all.

5.3 Recommendation

1. It is recommended that the distribution of data between different layers be properly analyzed in subsequent work.
2. The use of this model in a network area where there is no multicast-enabled device (or where the user is unsure), is absolutely encouraged.
3. Most Nigerian firms and aspiring individuals may not have the financial base to start an online broadcast system. This is the primary reason of this research. It is highly encouraged that it is used in such situations.

References

1) Todd, L (2007) CCNA: Cisco Certified Network Associate Study Guide. Wiley Publishing, Inc., Indianapolis, Indiana.

2) Elliotte, R (2004) Java Network Programming, 3rd Edition. O'Reilly Publishing, Inc., Indianapolis, Indiana.

3) Microsoft ® Encarta ® 2009. © 1993-2008 Microsoft Corporation.

4) David, R and Michael, R (2002) Java Network Programming and Distributed Computing, Addison Wesley.

5) Nash Equilibrium – Wikipedia the free encyclopedia (2010).

6) Anna, H (2003): Mobile Telecommunications Protocols for Data Networks. Wiley Publishing, Inc., John Wiley & Sons Ltd.

7) Richard. S (1993): TCP/IP Illustrated, Volume 1: The Protocols

8) A Beautiful Mind (2001): Movie by Universal Pictures, DreamWorks Pictures, and Imagine Entertainment; based on the life of Nobel Prize-winning American mathematician John Nash.

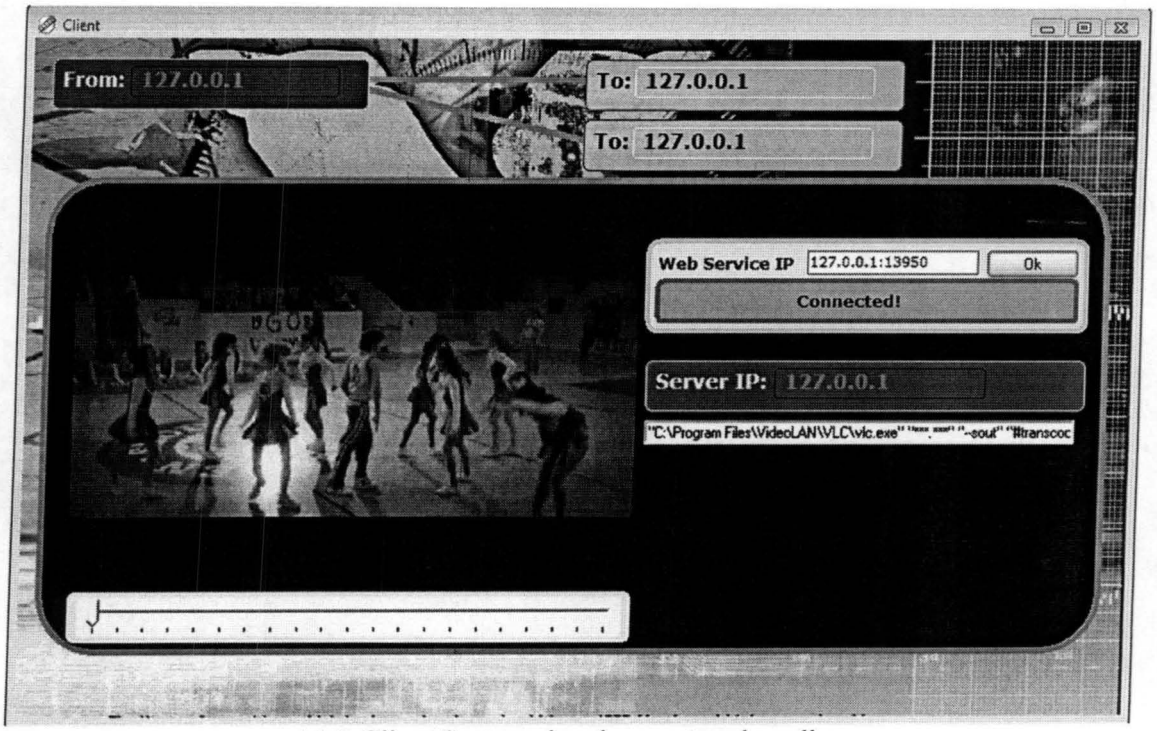
9) Peter, D (2005): Data Structures and Algorithms in Java. Prentice Hall.

10) Robert, S (2002): Algorithms in Java- Parts 1-4. Addison-Wesley.

11) Ian, S (2007): Software Engineering 8. Addison-Wesley.

APPENDIX 1

A1.1 Server System showing streaming media



A1.2 Client System showing captured media

APPENDIX 2

```
/*
 *
 * Created on 04 August 2010, 19:46
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

package Controller;

import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.ProduceMime;
import javax.ws.rs.ConsumeMime;
import javax.ws.rs.DefaultValue;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.GET;
import javax.ws.rs.QueryParam;

/**
 * REST Web Service
 *
 * @author Oguntade Philip temitope
 */

@Path("generic")
public class Beauty {
    @Context
    private UriInfo context;

    /** Creates a new instance of Beauty */
    public Beauty() {
    }
Intelligence intel = new Intelligence();
    /**
     * Retrieves representation of an instance of Controller.Beauty
     * @return an instance of java.lang.String
     */
    @GET
    @ProduceMime("text/plain")
    public String getText(@QueryParam("methods")@DefaultValue("0")String
methods,@QueryParam("arg1")String arg1,@QueryParam("arg2")String arg2
            ,@QueryParam("arg3")String arg3) {
        String value = "null";
        if (methods.equals("getCurrent")){
             value = String.valueOf(intel.getCurrent());
         }
        if (methods.equals("getServerIP")){
             value = String.valueOf(intel.getServerIP());
         }
        if (methods.equals("setServerIP")){
             value = String.valueOf(intel.setServerIP(arg1));
         }
        if (methods.equals("getServerClients")){
             value = String.valueOf(intel.getServerClients());
         }
```

```
        if (methods.equals("submit")){
             value = String.valueOf(intel.submit(arg1,arg2,arg3));
         }
        if (methods.equals("getReceivers")){
             value = String.valueOf(intel.getReceivers(arg1));
         }
        if (methods.equals("getGiver")){
             value = String.valueOf(intel.getGiver(arg1));
         }
        if (methods.equals("removeIP")){
             value = String.valueOf(intel.removeIP(arg1));
         }
       if (methods.equals("extractIpIndex")){
             value = String.valueOf(intel.extractIpIndex(arg1));
         }
        return value;
    }

    /**
     * PUT method for updating or creating an instance of Beauty
     * @param content representation for the resource
     * @return an HTTP response with content of the updated or created
resource.
     */
    @PUT
    @ConsumeMime("text/plain")
    public void putText(String content) {
    }
}


package Controller;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

/**
 *
 * @author Oguntade Philip temitope
 */
public class Intelligence {
     static List<Smart> set = new ArrayList<Smart>();
     SortLatency sl = new SortLatency();
     SortBytesRead sbr = new SortBytesRead();
     static Smart[] smart ;
     static StringBuffer str = new StringBuffer();
    /**
     * lets every PC check if there's an update in the system
     */
    static int current = 0;
    /**
     * lets the server set its IP
     */
    static String serverIP = "";
    /**
     * gets the current count so that systems can know if there's a change
in the system.
     * Infact, client systems should request this at once in five seconds
     */
    public  int getCurrent() {
```

```
        return current;
    }

    /**
     * gets the IP of the server so that clients can know whom to ping to
priodically measure their performance.
     * Infact, client systems should perform this operation at start-up
     */
    public  String getServerIP() {
        return serverIP;
    }

    /**
     * This method should be called whenever the server system starts up.
It will set the
     * Server IP so that client systems can call it and ping the server
system
     */
    public  boolean setServerIP(String serverIP) {
        Intelligence.serverIP = serverIP;
        return true;
    }


    /**
     * Returns all the PC (maximum of five[5]) that are to be fed by the
server
     */
    public  String getServerClients()
    {
        int i = 0;
        StringBuffer clients = new StringBuffer();

        for(Smart e:smart)
        {
            if(i<5)
            {
                clients.append((String)e.getIp());
            clients.append("*.*");
            i++;
            }
        }
        return clients.toString();
    }

     /**
     * At start-up distribution should be ordered by latency; call
startUpLatency(). For future sorting/distribution, sort by
     * the amounts of bytes read. Hence, accept all client data using this
method
     */
    @SuppressWarnings("static-access")
    public  String submit(String lat,String byt, String ip){
        int latency = Integer.valueOf(lat);
        int bytesRead = Integer.valueOf(byt);
    Smart sm = new Smart();
    sm.setBytesRead(bytesRead);
    sm.setLatency(latency);
    sm.setIp(ip);
   set.add(sm);
    Smart[] smm = new Smart[set.size()];
```

```
    Iterator iter = set.iterator();
    for(int i=0;i<=smm.length-1;i++)
    {
    smm[i] = (Smart) iter.next();
   }
   smart = sl.mergeSort(smm);
    current++;
    return "true";
    }

    /**
     * At start-up distribution should be ordered by latency. Don't use for
now
     */
    @SuppressWarnings("static-access")
    public void startUpLatency(String lat,String byt, String ip){
        int latency = Integer.valueOf(lat);
        int bytesRead = Integer.valueOf(byt);
    Smart sm = new Smart();
    sm.setBytesRead(bytesRead);
    sm.setLatency(latency);
    sm.setIp(ip);

    current++;
    }
    /**
     * removes the system from distribution
     */
    @SuppressWarnings("static-access")
    public boolean removeIP(String ip)
    {
        boolean bool = false;
       Smart sm;
       int k = -1;
        for(int i = 0;i<smart.length;i++)
        {
            if(smart[i].getIp().equals(ip))
            {
            sm = smart[i];
            set.remove(sm);
            Smart[] smm = new Smart[set.size()];
  Iterator iter = set.iterator();
                for(int ii=0;ii<=smm.length-1;ii++)
                {
                smm[ii] = (Smart) iter.next();
                }
            smart = sl.mergeSort(smm);
            current++;
            bool = true;
            }
        }

        return bool;
    }
    /**
     * return the IPs (max of 2) that are to be fed by the supplied IP
     */
   public String getReceivers(String ip)
   {
       String receiver1 ="";String receiver2 ="";
```

```
        if(smart.length-1 >= 2*extractIpIndex(ip)+1)
            receiver1 =
 String.valueOf(smart[2*extractIpIndex(ip)+1].getIp());
        if(smart.length-1 >= 2*extractIpIndex(ip)+2)
            receiver2 =
 String.valueOf(smart[2*extractIpIndex(ip)+2].getIp());
        return receiver1+"*.*"+receiver2;
    }
    /**
     * return the IP of the system that serves this IP streaming media
     */
    public String getGiver(String ip)
    {
        String giver1 ="";
        if(extractIpIndex(ip) <= 4 && extractIpIndex(ip) >= 0)
            giver1 = getServerIP();
        else if(smart.length-1 >= (extractIpIndex(ip)-1)/2 &&
 extractIpIndex(ip) >= 0)
            giver1 = String.valueOf(smart[(extractIpIndex(ip)-
 1)/2].getIp());

       return giver1;
    }
    /**
     * Searches for the index of the IP so that the giver and receivers can
 be computed. Returns -i if not found.
     * This algorithm uses a linear search pattern. It is quite inefficient
 for large data
     */
    public int extractIpIndex(String ip)
    {
        int k = -1;
        for(int i = 0;i<smart.length;i++)
        {
            if(smart[i].getIp().equals(ip))
                k = i;
        }
        return k;
    }

}


package Controller;

/**
 *
 * @author Oguntade Philip temitope
 */
public class SortLatency {

public static Smart[] mergeSort(Smart[] data) {
   return mergeSortHelper(data, 0, data.length - 1);
 }

protected static Smart[] mergeSortHelper(Smart[] data, int bottom, int top)
{
 if (bottom == top) {
   return new Smart[] { data[bottom] };
 } else {
int midpoint = (top + bottom) / 2;
```

```
return merge(mergeSortHelper(data, bottom, midpoint),mergeSortHelper(data,
midpoint + 1, top));
 }
}

 /**
  * Combine the two sorted arrays a and b into one sorted array.
  */
 protected static Smart[] merge(Smart[] a, Smart[] b) {
   Smart[] result = new Smart[a.length + b.length];
   int i = 0;
   int j = 0;
   for (int k = 0; k < result.length; k++) {
     if ((j == b.length) || ((i < a.length) && (a[i].getLatency() <=
b[j].getLatency()))) {
      result[k] = a[i];
       i++;
     } else {
       result[k] = b[j];
       j++;
     }
   }
   return result;
 }

}


package Controller;

/**
 *
 * @author Oguntade Philip Temitope
 */
public class Smart {
long latency;
long bytesRead;
String ip;

    public String getIp() {
        return ip;
    }

    public void setIp(String ip) {
        this.ip = ip;
    }
    public long getLatency() {
        return latency;
    }

    public void setLatency(long latency) {
        this.latency = latency;
    }

    public long getBytesRead() {
        return bytesRead;
    }

    public void setBytesRead(long bytesRead) {
        this.bytesRead = bytesRead;
    }}
```