AU J.T. 15(1): 23-31 (Jul. 2011)

Real-Time Multi-Node Data Transmission: Designing a Low Latency Computer Network

Omokhafe James Tola and Philip Temitope Oguntade

Department of Electrical and Computer Engineering, Federal University of Technology Minna, Niger State, Nigeria
E-mail: <omokhafe@gmail.com>

Abstract

There is a need 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 the data and no computer system is over-loaded in the process. This paper 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 primarily its use in the latter.

Keywords: *Real-time multimedia broadcast, low latency, computer network.*

Introduction

The deployment of high-performance servers for real-time broadcast has become a norm in the computing societies. These servers offer the required optimum concurrency needed in serving the real-time data of all subscribing client computer systems. For instance, in order to deploy a server to stream real-time multimedia data to client systems one may have to consider a number of factors: these include 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 the transmission or hang up the server system. The purpose of this work is to find an optimum approach around this challenge.

Todd Lammle 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 (Lammle 2007).

Design and Implementation

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 allocations in a real-time broadcast. 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 network being developed (Sommerville 2007). For all systems, except the simplest ones, they usually include (see Fig. 1):

- A real-time clock: The clock provides signals to schedule processes periodically.

- An interrupt handler: This component manages aperiodic requests for service.
- A scheduler: This component is responsible for examining the processes that can be executed and choosing one of these for execution.
- A resource manager: Given a process that is scheduled for execution, the resource manager allocates appropriate memory and processor resources.
- A dispatcher: This component is responsible for starting the execution of a process.

In this work, additional facilities are needed, 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.

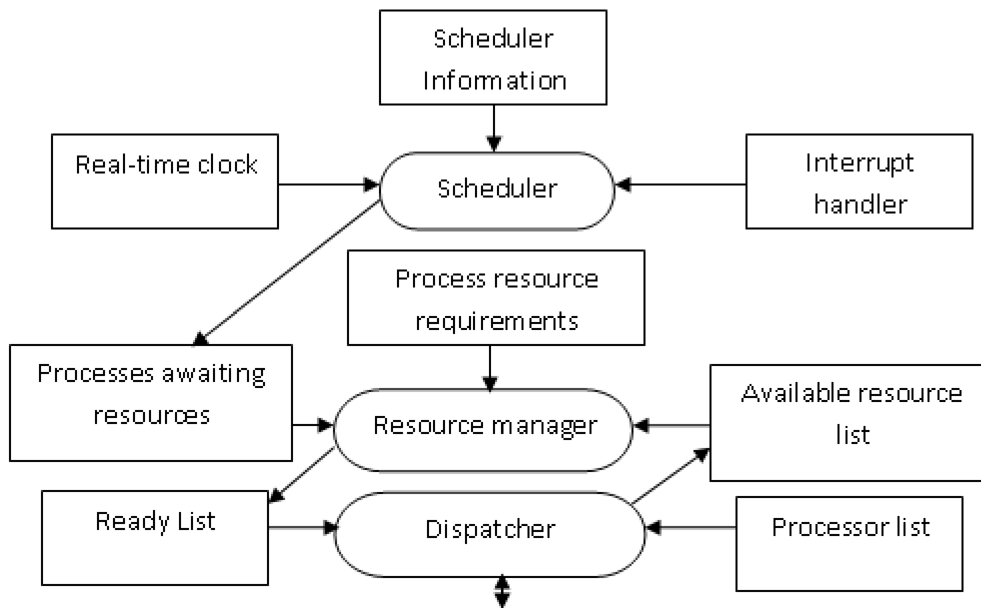

Fig. 1. Components of a real-time operating system.

Distribution Dynamics

Suppose there are n users hoping to receive streams from a server. However, there is no guarantee that the system will be able to broadcast the said streams to all n users.

Using the unique approach proposed in this work, all receiving devices can be placed in a queue in the order of their attempt to connect to the server as shown in Fig. 2. Therefore, whenever the first subscriber (n-(n-1)) requests a 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 its memory and consumes it from this temporal storage. Whenever the second user requests a connection, it shall be granted a permission to stream continuously from the first user's system. Then the third user does the same to receive the stream from the second user (Reilly and Reilly 2002).

According to a linear model for the distribution of data among clients based on this approach, the server shall only be responsible for the first user's connection but also for all users' authentication.

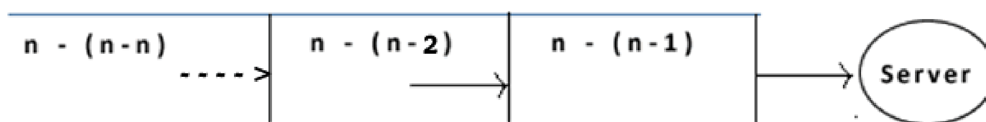

Fig. 2. A queue of receiving devices attempting to connect to the server.

A Further Review

As brilliant as the proposed technique may seem to be, there are loopholes that can completely bring the streaming strategy into disrepute. Firstly, the latency will depend on the number of systems connected. 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, consider a broadcast chain of 1,000 users using the linear approach. If the computer system of the 10th subscriber goes out of service, it means that the remaining 990 dependents will be completely cut off from the broadcast (Hac 2003).

These are, of course, major drawbacks of the linear approach. However, these can be significantly reduced by merely introducing the *tree-based approach.* Here every user broadcasts on the average to two additional users. This is represented schematically in the diagram shown in Fig. 3.

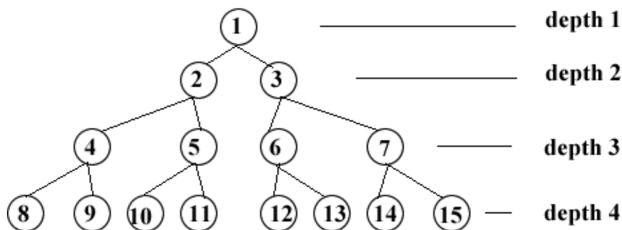

Fig. 3. A tree-based model for the distribution of data among clients.

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 the depth of the tree. The total number of depths in a 1,000-user system is

2^d-1 = 1,000, or

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

$d = 3.000434 / 0.301029$,

$d = 9.9673 \approx 10$.

The 10th user can be found at the 4th depth with 7 other users. But between the 1st and the 4th depth, there are 15 users, thus leaving 985 users for evaluation. 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 10 th user's failure is 985 / 8 ≈ 123 users. This is a big 867-user improvement over the linear approach. Note that this value shows a more significant improvement as the depth increases (Stevens 1994). Table 1 shows this improvement for eventual 40th subscriber failure.

Table 1. Amount of disconnected systems due to eventual 40th subscriber failure.

Number of users in the system	Amount of disconnected systems due to eventual 40th subscriber failure	
	Linear approach	Tree-based approach
1,000	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

One possible drawback of this approach is that the 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.

Checkmating Error Situations

In any of the two approaches, the goal is to ensure the streaming restoration in the presence of disconnected subscribers without loss of data or increased transmission time. This can simply be ensured in 3 steps by:

- Monitoring the amount of bytes read by each system;
- Determining which parent node disconnects;
- 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 in the network has read an amount of bytes sufficient to be consumed all through the shutdown and handshaking process (period between loosing and regaining a connection).

Yet Another Review

There is yet another challenge. Iimagine that a football match is being broadcast and 8 people shout “it’s a goal” at time *t*, followed by 16 people at time $t+x$, 32 at time $t+2x$, 64 at time $t+3x$, and so on. This means that the 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-based approach it means that the number of depths will correspond to the amount of milliseconds of the time 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 subscribers by about 3 seconds. What if one can, firstly, delay the 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 this knowledge, one can ensure that a particular block of data gets to every user before any other user can watch it (Drake 2005, Sedgewick 2002).

To accomplish this task, one can use the following time corrector:

$$t = \left(\sum_{d=2}^{d=n} l/f\right), \quad (1)$$

where:

t = time to play (standard measurement in milliseconds);

l = network latency between any two depths;

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

$f = d - 1$;

n = number of depths.

There is a word of caution here. Firstly, one 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 one. Secondly, the users at the last depth do not need to compute this time of play because the last depth is the zeroth frame of reference in Eq. (1). Hence, the users at the last depth should just consume the 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 subscriber system will compute its own time to start watching the TV content and this time increases down the tree.

Process Management

Real-time systems have to handle external events quickly and, in some cases, meet deadlines for processing these events. This means that the event-handling processes must be scheduled for execution in time to detect an event and must have sufficient processor resources to meet its deadline. The process manager in a real-time operating system is responsible for choosing processes for execution, allocating processor and memory resources, and starting and stopping the execution of processes on a processor as shown in Fig. 4.

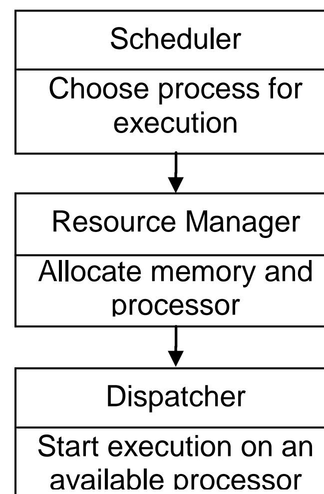

Fig. 4. Actions of a real-time operating system required to start a process.

The Java code snippets below show the different segments described in the ‘Design’ section above (Harold 2004).

```
/**
* lets every PC check if there's an update in
the system
*/
staticint 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
*/
publicintgetCurrent() {
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() {
returnserverIP;
}

/**
* 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].g
etIp());
if(smart.length-1 >= 2*extractIpIndex(ip)+2)
receiver2 =
String.valueOf(smart[2*extractIpIndex(ip)+2].g
etIp());
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
*/
publicintextractIpIndex(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 real-time operating systems have to be able to manage at least two priority levels for system processes:

- 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.
- Clock level: This level of priority is allocated to periodic processes.

For the implementation of this work, the Java code snippet below shows a part of the process manager that deals with interrupt handling.

```
/**
 * removes a malfunctioning system from
distribution then *re-shuffules the remaining clients
based on the applied algorithm.
*/
publicbooleanremoveIP(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;
 }
 }
returnbool;
}
```

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 one of these priority levels, different classes of processes 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 the 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 the processing must be completed). Using the timing requirements specified in the application program, the real-time operating system arranges the execution of periodic processes so that they can all meet their deadlines.

Priority Handling

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

Having considered the real-time nature of the project, the distribution model uses merge-sort algorithm in arranging client systems according to their current performance state. The merge-sort algorithm is an example of a divide-and-conquer algorithm. In such an algorithm, one divides the data into smaller pieces, recursively conquers each piece, and then combines the partial results into a final result.

An implementation of the merge-sort algorithm in Java language, as it is used in this work, is shown below.

```
public class SortLatency {

public static Smart[] mergeSort(Smart[] data) {
returnmergeSortHelper(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;
 }
}
```

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-based connection (Microsoft® Encarta® Multimedia Encyclopedia 2009). 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");
longstartTime = System.currentTimeMillis();
booleanbool = in.isReachable(5000);
longendTime = 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");
```

Tests, Results and Discussion

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 as shown in Table 2. Having built the data transmission system over a socket connection (point-to-point connection), data was transmitted from the server down the nodes as they are located down the balanced binary tree.

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

Table 2. Parameters of the data transmission system.

Component	Quantity or Size
Number of Processors	2
Processor Speed	2.1 GHz each
RAM Size	2 GB Average
Other Parameters	
Network Mode	Peer-to-peer over wireless LAN
Distance(s) between systems	Average is less than 20 metres
Expected Multimedia network throughput	150 kbps (average)

Results

After transmitting data in a depth-first traversal pattern it was realized that the response time was 0 milliseconds on the average. This value shows a remarkable improvement over the anticipated latency (response time) of 100 milliseconds as predicted in section ‘Another Review’ above. In addition, a break between failover and re-transmission was noticed whenever there was a breakdown in any client system.

The diagrams in Figs. 5 and 6 show what happens if there is a failure in any of the systems. Figures 7 and 8 show streaming and captured media.

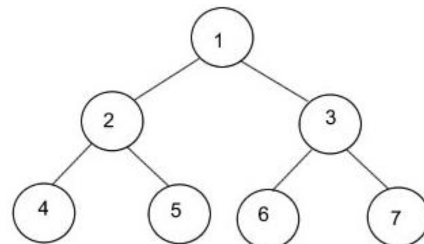

Fig. 5. A diagram showing a 7-user system.

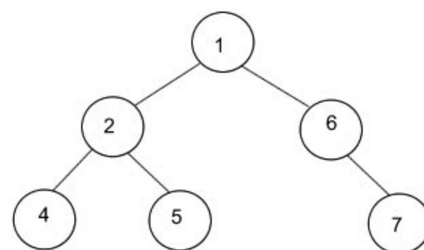

Fig. 6. A recovery model assuming that a 6-user system 6 is performing better than the 7-user system.

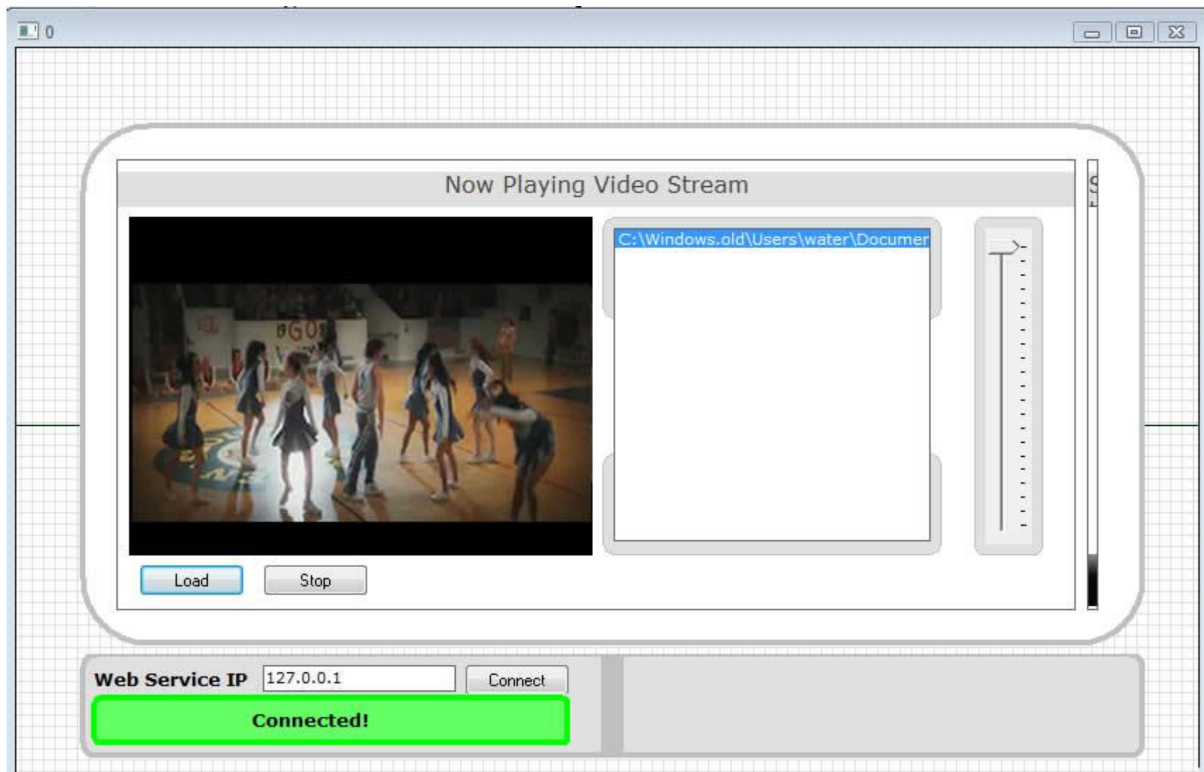

Fig. 7. A server system is showing streaming media.

Fig. 8. A client system is showing captured media.

Discussion

It is strongly recommended that the 100-millisecond delay be implemented before feeding any client system with media data. This recommendation may be ignored only if 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 the client side for the period of reception of the broadcast.

Conclusion

With all these considerations 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 arbitrary amount of subscribers with all the feeding coming from one personal computer. As the technology advances and the computing facilities get more sophisticated, this work becomes more meaningful; such that it could possibly become a network paradigm.

Recommendations

1. It is recommended that the distribution of data between different layers be properly analyzed in a 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 about it), 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

Drake, P. 2005. Data structures and algorithms in Java. Prentice Hall, Inc., Upper Saddle River, NJ, USA.

Hac, A. 2003. Mobile telecommunications protocols for data networks. John Wiley & Sons, Ltd., Chichester, West Sussex, England, UK.

Harold, E.R. 2004. Java network programming. 3rd ed., O'Reilly Media, Inc., Sebastopol, CA, USA.

Lammle, T. 2007. CCNA: Cisco® certified network associate study guide. 6th ed., Wiley Publishing, Inc., Indianapolis, IN, USA.

Microsoft® Encarta® Multimedia Encyclopedia. 2009. © 1993-2009, Microsoft Corporation, Redmond, WA, USA.

Reilly, D.; and Reilly, M. 2002. Java network programming and distributed computing, Addison Wesley, Boston, MA, USA.

Sedgewick, R. 2002. Algorithms in Java. Parts 1-4. 3rd ed., Addison-Wesley, Boston, MA, USA.

Sommerville, I.F. 2007. Software engineering. 8th ed., Addison-Wesley, Harlow, England, UK.

Stevens, W.R. 1994. TCP/IP illustrated. Volume 1: The protocols. Addison-Wesley, Boston, MA, USA.