Complexity Analysis Involving Heterogeneous System

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Computer Science & Engineering

By:
Kuldeep Sharma
(Regn. No.80732012)

Under the supervision of:
Dr. Deepak Garg
Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

JUNE 2009

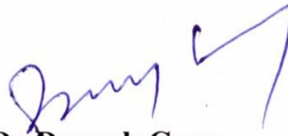
Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Complexity Analysis Involving Heterogeneous System"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in Computer Science and Engineering submitted in Department of Computer Science and Engineering , Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of **Dr.Deepak Garg** refers other researcher's works which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

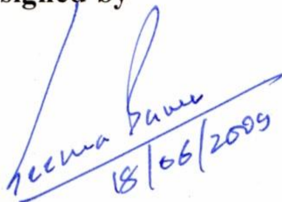
Kuldeep Sharma
Roll No: - 80732012

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.



Dr.Deepak Garg
Assistant Professor
Department of Computer Science and Engineering
Thapar University,Patiala

Countersigned by



Dr. (Mrs.) SEEMA BAWA
Professor & Head
Computer Science & Engineering. Department
Thapar University, Patiala

Dr. R.K.SHARMA
Dean (Academic Affairs)
Thapar University, Patiala

Acknowledgement

I am thankful to **Prof.(Mrs.)Seema Bawa**, Head Department of computer science and Engineering Thapar University and **Mrs. Inderveer Channa**, P.G. Coordinator, for the motivation and inspiration that triggered me for the seminar work.

No volume of words is enough to express my gratitude towards my guide, **Dr. Deepak Garg** Assistant Professor, Computer Science and Engineering Department, Thapar University, who has been very concerned and has aided for all the guidance essential for the thesis report. He has helped me to explore this vast topic in an organized manner and provided me all the ideas on how to work towards a research-oriented venture.

I would also like to thank the staff members and my colleagues who were always there in the need of the hour and provided with all the help and facilities, which I required, for the completion of my thesis.

Most importantly, I would like to thank my parents and the Almighty for showing me the right direction out of the blue, to help me stay calm in the oddest of the times and keep moving even at times when there was no hope.

Kuldeep Sharma

Roll No.80732012

Abstract

Complexity analysis is one of the most complicated topics in mathematics. It involves an unusual concept and some tricky algebra. This report is a humble trail to demystify the idea in detail. Heterogonous systems are becoming bigger and more complex. While the complexity of large-scale heterogeneous systems has been acknowledged to be an important challenge, there has not been much work in defining or measuring system complexity. Thus, today, it is difficult to compare the complexities of different systems, or to state that one system is easier to program, to manage or to use than another. Here we try to understand the factors that cause heterogeneous systems to appear very complex to people. We define different aspects of system complexity and propose metrics for measuring these aspects. We also show how these aspects affect the system. Based on the aspects and metrics of complexity, we propose general guidelines that can help to measure the complexity of systems.

There are many types of analysis and various methods available to analyze the algorithms like apriori analysis, posterior analysis, micro analysis, macro analysis, amortized analysis, big O notation, theta notation, potential method, accounting method. All analysis and methods are situation specific. Our report does a comparative study of various method and techniques of algorithm analysis giving their specific advantages and disadvantages.

In this report online banking system is discussed in detail. In this systems there are various factors that will play significant roles in the overall complexity, there are many algorithms that are running simultaneously like fault tolerance, authentication, encryption, routing tables, communication protocols and for error recovery. Every algorithm contributes in the system. In this report we will measure how much contribution they have. Every algorithm have time and space complexity, if the algorithms are running parallel then the complexity will be highest out of them . E.g. we have complexities n, n+1,n^2 then the complexity will be $n^{2.}$ If we have N algorithms and they are running sequentially then the complexity will be sum of all these.eg. We have N algorithms running simultaneously then the total complexity will be Tc = 1+2+……+N.

Table of Content

Chapter 1 Introduction……………………………………………………………1

1.1 Guidelines to measure complexity……………………………2

Chapter 2 State of Art in Complexity Analysis ………………………………....5

2.1 Algorithm…………………………………………………...5

2.2 Complexity Analysis of an algorithm…………………………6

2.3 Apriori Analysis………………………………………………6

2.4 Posterior Analysis……………………….........................9

2.5 Micro Analysis…………………………………………….10

2.6 Macro Analysis……………………………………………11

2.7 Best, Average, Worst Case Analysis………………………..12

Chapter 3 Measure Complexity…………………………………………………14

3.1 Big *O* Notation………………………………...................14

3.2 Theta Notation………………………………….................15

3.3 Omega Notation……………………………………………..16

3.4 Amortized Analysis………………………....................17

3.5 Accounting Method……………………………………………18

3.6 Potential Method……………………………....................18

Chapter 4 Problem Statement…………………………………………………...20

Chapter 5 Heterogeneous System……………………………………………….21

5.1 Encryption…………………………………………………22

5.1.1 Encryption Algorithm………………………………..23

5.1.2 DES/3DES…………………………………… ………..23

5.1.3 IDEA……………………………………………………23

5.1.4 SEAL……………………………………………………23

5.1.5 RC4…………………………………………… ………..24

5.2 Authentication………………………………………………24

5.2.1 Passwords……………………………………………24

5.2.2 Needham and Schroeder Protocol……………………25

5.2.3 Public Key Encryption………………………………25

5.2.4 Authentication Algortihtm…………………………...25

5.3 Routing and Routing Algorithms…………………................26

5.3.1 Distance Vector Algorithms…………………………26
5.3.2 Link-State Algorithms ………………………………26
5.3.3 Complexity of Routing Algorithm…………………...32
5.4 Concurrency Control and Algorithms………………………35
5.4.1 Trasaction ACID Rules………………………………35
5.5 Fault Tolerance……………………………………………….36
5.5.1 Checkpoint Recovery Method……………….. ………..37
Chapter 6 Results & Discussion…………………………………………...39
6.1 Fault Tolerance and Recovery Algorithm Complexity………39
6.2 Authentication Algorithm Complexity………………………39
6.3 Routing Complexity…………………………………………39
6.4 Backup Algorithm Complexity………………………………40
6.5 Communication Complexity…………………………………40
Chapter 7 Future work & Conclusion……………………………….……….42
References……………………………………………………………..……….44
Publications……………………………………………………………………46

List of Figures

Figure 1.0 Banking Growth……………………………………………………………2

Figure 2.1 Apriori Analysis…………………………………………………………7

Figure 2.2 Posterior Analysis……………………………….. ………………..10

Figure 3.1 Big O………………………………………………………………………15

Figure 3.2 Omega Notation…………………………………. ………………..16

Figure 5.1 Online Banking System…………………………………………………22

Figure 5.2 Routers in a network and the routing process………………………..30

Figure 5.3 Routing Table ………………………………………………………..33

Figure 5.4 Interval Routing…………………………………………………………34

Figure 5.5 Checkpoint Mechanism…………………………………………………37

List of Tables

Table 1.0 Heterogeneous systems module wise complexity……………………..3

Table 3.0 Comparison of all methods…………………………………………..19

Table 5.0 Routing Complexity………………………………………………….35

Table 6.1 Complexity of message passing in wide area network……………....40

Table 6.2 Complexity of all Modules…………………………………………...40

Table 6.3 System Complexity…………………………………………………..41

Chapter 1

Introduction

The size and complexity of heterogeneous systems have been increasing inexorably in the recent past. Large-scale distributed systems such as internet systems, ubiquitous computing environments, grid systems, storage systems, enterprise systems and sensor networks often contain immense numbers of heterogeneous and mobile nodes. These systems are highly dynamic and fault-prone as well. As a result, developers find it difficult to program new applications and services for these systems; administrators find it difficult to manage and configure these complex, device-rich systems; and end-users find it difficult to use these systems to perform tasks. The computational aspects of complexity have been studied extensively. The concepts of time and space complexity of different kinds of algorithms are well understood. However, the human aspect of complexity is still poorly understood. This human aspect of complexity is a significant challenge today, requiring urgent solutions. The complexity of heterogeneous systems for people has been widely identified to be an important problem [1,2]. Many enterprises, governments and other organizations have large, complex computing systems that are difficult to manage, maintain and change. Application and service developers often face steep learning curves before they can start programming large distributed systems. Organizations also tend to spend huge amounts in administration and maintenance costs, which often exceed the cost of buying the system in the first place. End-users, too, are often overwhelmed with the complexity of a single computer, let alone multiple, heterogeneous devices. In spite of the fact that system complexity has been widely talked about, the term “complexity” is often used loosely in connection with computer systems. There are no standard definitions of complexity or ways of measuring the complexity of large systems. As with so many complex things, computer system complexity means different things to different people. In this report, we identify the different algorithms that take part in the complexity of heterogeneous system: Fault tolerance algorithm, routing algorithms, backup algorithm, error detection and correction algorithms etc. We describe the rationale behind these different aspects and show how these aspects manifest themselves in heterogeneous systems. This report shows how these aspects of complexity impact the overall system. The computer systems research community

has been actively looking at different approaches to reduce the complexity of systems. These approaches often take the form of middleware or programming frameworks to simplify the task of developers [3], various system management tools for administrators; and intuitive user interfaces for end-users. However, today, there is no way of formally, and quantitatively, saying that a certain solution does reduce complexity, or that one solution is better than another. We hope that our proposed aspects and metrics of complexity will allow people to compare solutions in a more scientific way, as well as guide future solutions to tackling the problem of complexity. So far, approaches to tackling system complexity have been rather ad-hoc in manner. The main contribution of this report is in addressing the problem of heterogeneous system complexity in a more formal and scientific way.

At last we will find the complexity of Internet banking system. Internet Banking developed due to increasing demand of online banking transactions. The biggest advantages of Internet Banking consist of complex banking solutions, 24 hours availability, quick and secure access to the back-end application through Internet. Customer requests for quick access and no matter the location at their bank accounts, or at financial/banking transactions, all of these have determined the banking institutions worldwide to adopt the internet as the optimal solution for the presented demands. Through the internet network banks are able to connect front-end applications with back-end. Based on such advantages, internet banking applications were created. The evolution of bank presence on web from simple, static applications to complex, dynamic application, to complex, dynamic application with numerous transactions, is presented below (figure 1.0). The advantages of internet banking applications consist of: quickness; secured access to sensitive data as accounts, personal data of customers, transactions; account management; operating sale-purchase transactions in real-time and at long distance; suppressing the stress of staying in bank for a transaction; low costs for the maintenance of this kind of applications.

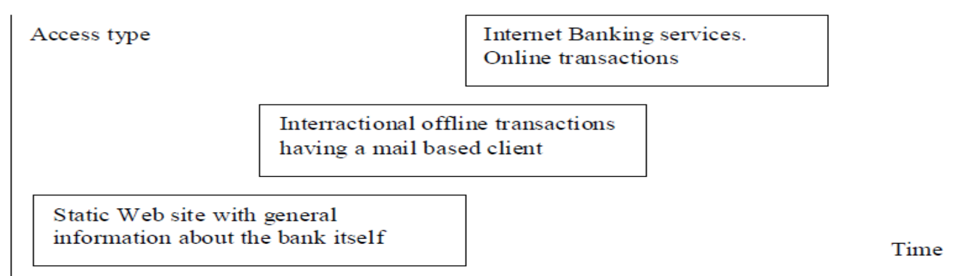


Figure1.0 Banking Growth

1.1 Guide Lines to Measure complexity in Heterogeneous System

	Module 1	*Module 2*	*Module 3*	*Module 4*	*Module 5*
H_system 1	n^2	$n \log n$	2^n	$\log n$	n
H_system 2	$\log n$	$n!$	n^3	$n \log n$	n
H_system 3	n^3	n	$\log n$	$O(1)$	$n \log n$
H_system 4	n^2	$(\log n) / 2$	n^4	N	$O(1)$
H_system 5	n^2	$n \log n$	$n^2 \log n$	$(\log n)^2$	$n \log n$

Table 1.0 Heterogeneous Systems and module wise complexity.

In above shown table we have assume five heterogeneous systems and each system has five different modules, in the table 1.0 their module wise complexity is given . Now we will calculate the overall complexity of the system (T_C).

Scenario1. In H_system 1 we have a very simple system and no module is running in parallel, all the modules are running in sequence, simply by adding the complexity of each module will give the overall complexity of the system (*i.e.* $T_c = n^2 + n \log n + 2^n + \log n + n$). $T_C = 2^n$ .It is an easy example and these types of systems are rare in use. Now we have more dynamic system with inter connections and multiple dependencies with distributed, parallel or other techniques of advance computing.

Scenario2. In H_system2 some of the modules are running parallel, so approach to measure the complexity will be different. If module 3 and module 5 are running in parallel fashion, so the highest complexity from both the module will be taken in to the account (the complexity of module 3 and module 5 is n^3 and n respectively so will take n^3 highest one), so the total complexity (T_c) of the system H_system 2 will be $T_c = \log n + n! + n^3 + n \log n$, $T_C = n^3$. These types of systems are most common one. Some time all the modules are running in parallel manner then the highest complexity will be the Total complexity (T_c) of the system. In actual it may not be that simple and many more parameters may affect overall complexity.

Scenario3. Heterogeneous system may include one or more critical module like security where it cannot be compromised in context to complexity, so in this type of cases one module can affect the overall system's complexity. Like in banking system or online payment system security, atomicity, accuracy, consistency, isolation and

durability are modules those cannot be compromised we have to take care these modules, they can increase the total complexity of the system.

Scenario4. In this scenario we may have some dependencies. One module may dependent on another one so the complexity will increase. Like in H_system 3 we can't start the second module until first module is not finished if the complexity of module one is high then automatically the overall system's complexity will get effected and output of Module 3 is dependent module 2 if output varies then it may has effect on T_C.

Scenario5. Now we have system in which two modules (module 1and module 2) are running in parallel and have one complex module 3 the output of module 5 is dependent on module 4. If we do some computations on client side rather than to do all computations on server side, we do so the complexity of module 3 is decreased n^4 to *n.* the overall system complexity will automatically decreased.

Organization of Thesis

The thesis is organized as follows:

Chapter-2	Discussion on fundamentals of complexity and various analysis methods.
Chapter-3	This chapter presents various methods to calculate complexity.
Chapter-4	Problem statement for complexity analysis involving heterogeneous system.
Chapter-5	It presents heterogeneous system and various algorithms used in the system.
Chapter-6	It shows results & discussion.
Chapter-7	Conclusion.

Chapter 2

State of the Art in Complexity Analysis

2.1 Algorithm

An *algorithm* is a well-defined sequence of steps to solve a problem of interest. It is a procedure for solving a problem, with special focus on solving problems using a computer. Some examples of standard algorithms are listed as follows:

- ✓ Search Algorithms (Binary search, Fibonacci search, Golden section search, etc.)
- ✓ Sorting Algorithms (Insertion Sort, Bubble Sort, Quick Sort, Heap Sort, etc.)
- ✓ Shortest Path Algorithms (Dijkstra's Algorithm, Floyd's Algorithm, etc.)

Informally, an algorithm is any well-defined computational procedure that takes some values, or set of values as input and produces some values, or set of values, as output.

An algorithm is thus a sequence of computational steps that transform the input into the output.

Formal Definition: - An Algorithm is a correct solution for a problem in finite sequence of steps where each step is unambiguous and which terminates for all possible inputs in a finite amount of time and memory [4].

Components of an Algorithm

- ❖ Input
- ❖ Output
- ❖ Logic to convert from Input to output.

Input: - It is value or the parameters we pass (given as input to process).

Output: - It is the desired result which we want to produce.

Logic: - Sequence of computational steps that transform the input into the output. In an algorithm logic is the computational part. In which we perform some logic or calculation to precede the output.

Only computational problems not others can be solved using computer algorithms others like whether God exists, whether you love me or not, whether he fears or not?

2.2 Complexity Analysis of an Algorithm

The complexity of an algorithm means a function representing the number of steps (times) required to solve a problem under the worst-case behavior. An algorithm consists of asset of steps. For some problems, the algorithm will be executed for all its steps for its subsets of steps. The worst-case behavior of an algorithm means the maximum number of steps executed (or time taken) to solve a problem. Hence, for any algorithm, even before implementing it, its complexity function should be analyzed. Such analysis will help us predict the maximum magnitude of time required to solve a problem using the algorithm. There is one more type of complexity function, namely volume complexity function, which represents the maximum primary memory requirement while executing the algorithm. [4]

Time complexity: As stated earlier, the time complexity function of an algorithm gives the worst-case estimate in terms of the number of steps to be executed for the algorithm. The order of the algorithm is defined like (n^2), $O(n!)$, $O(2^n)$, etc. The big O means the maximum order of the algorithm. $O(n^2)$ means that order of the algorithm is n^2, which indicates the maximum number of steps required to solve any problem by that algorithm is n^2 where n is the problem size. The time complexity function may be either polynomial or exponential.

Volume Complexity: The volume complexity function of an algorithm represents the amount of prime memory space required while executing the algorithm. Again this may be either polynomial or exponential. In the case of the branch and bound technique, if Breath First Search (BFS) is used, the function representing the memory space required to store the sub-problems will be in exponential from; if Depth First Search (DFS) is used, the function representing the memory space required to store the sub-problems will be in polynomial form. The type of the algorithm as well as data structure affects the volume complexity. This analysis will be helpful in deciding the types of computer to be used for implementing the algorithm.

2.3 Apriori Analysis

Apriori – Designing then making. The principle of apriori Analysis was introduced by ***Donald Knuth***. In apriori Analysis first we analyze the system or problem then we design or write the code for the problem.

Path "P"
Place A
Place B
Path "Q"

Figure 2.1 Apriori Analysis

Consider the above given example (figure 2.1). Suppose that we need to go from place A to place B and there exists two paths P and Q for the same. It is clear that for any given method (in this physical example Method will comprises- vehicle, its speed etc.) Path P will always take less time. Similarly for a given problem in computer science the method would consist of the processor, main memory etc. and the algorithm will correspond to the path. So given two are more algorithms for a problem we are interested in doing an machine independent analysis of these algorithms and say algorithm A is better than the remaining algorithms. This leads to the principle of ***Apriori Analysis***, apriori analysis means doing an analysis of the solutions before we code.

Foundation of Algorithm Analysis

- ✓ To do an apriori Analysis we need some tools. The tools include the instruction count and size of input.
- ✓ To do an apriori analysis we need to identify the basic operations (or instructions) in the algorithm and do a count (Instruction Count) on these operations. The instruction count is used as a figure of merit in doing the analysis.
- ✓ The size of input (input size) is used as a measure to quantify the input.
- ✓ Worst Case Complexity refers to the maximum instruction count among all instruction counts. Let $\mathbf{I_n}$ denote the set of all inputs of size **n** and let **IC(i)** denote the instruction count on a specific input **i**.

Worst Case Complexity, **WCC = Max IC(i), for all I in** $\mathbf{I_n}$**.**

Since the choice of a data structure or algorithm is generally made before a program is written, *apriori* (before the fact) analysis is necessary.

Frequency Count Analysis

A frequency count of statements executed is the most direct form of apriori analysis of the time used by an algorithm. Each statement in a program adds the value of 1 to the frequency count each time it is executed. The major failing of frequency count

analysis of an algorithm written in a high level programming language is that each statement in such a programming language does not generate the same number of binary instructions when compiled, nor do the binary instructions each take the same time to execute. A frequency count of data structure elements used is the most direct form of apriori analysis of data structures. Each element in a data structure adds the value of 1 to a frequency count analysis of a data structure. The major failing of frequency count analysis of a data structure is that data elements in a high level programming language program are not all equal in space used. Even though frequency count analysis produces a very specific looking result, remember that it really produces an approximation. In fact, all forms of apriori analysis are approximations. These approximations are tolerated because of they produce a good enough result that allows competing algorithms and data structures to be compared before programmer time is spent coding a solution. The assumption made in a frequency count analysis is that these differences will average out. Since they are approximations, the results can apply equally over solutions to be produced for all possible hardware.

Example of Frequency Count

Algorithms involving nested loops are very common. The following is a fragment that uses a nested for-loop:

```
int b;
for (int x=0; x<5; x++)
        for (int y=0; y<4; y++)
        {
                b = x * y;
                    cout << b;
                    }
```

Since there are three simple variables, space use is 3. The only complication in analyzing this fragment for time is the nested for-y-loop, which will be performed in its entirety each time the for-x-loop makes one pass.

The number of times that the for-y-loop will be started from its beginning conditions (the number of passes through the body of the for-x-loop) will have to be used to multiply the time count of the statements in the for-y-loop. The frequency count analysis of this fragment is as follows:

```
int b;                                   // add 1 to space
for (int x=0; x<5; x++)                  // add 1 to space, 6 to time
    for (int y=0; y<4; y++) {            // add 1 to space, 5 times 5 to time
        b = x * y;                       // add 4 times 5 to time
            cout << b; }                 // add 4 times 5 to time
```

This gives:

Time	Space
6	1
+25	+1
+20	+1
+20	3
71	

If, in the previous example fragment, the upper limit of the for-x-loop and the for-y-loop are represented by variables, the problem of frequency analysis becomes marginally more complex. Again, if the worst case assumption is used, both the for-x and for-y loops will have a big enough upper limit, so both upper limits can be represented by n. The fragment and the analysis of the fragment become:

```
int b;                                   // add 1 to space
    for (int x=0; x<n; x++)              // add 1 to space, n +1 to time
        for (int y=0; y<n; y++) {        // add 1 to space, n times n + 1 to time
            b = x * y;                   // add n times n to time
                cout << b;}              // add n times n to time
```

This gives:

Time	**Space**
$n+1$	1
$+ n^2 + n$	+ 1
$+ n^2$	+1
$+ n^2$	3
$3n^2 + 2n + 1$	

Notice that one loop dependent on n gives a result in the form of *(an + b),* where the largest order of magnitude is 1. Two nested loops dependent on n give the result in the form $(an^2 + bn + c)$, where the largest order of magnitude is 2. In fact, the pattern continues. With a triply nested set of loops that depend on a limit of n, the result is in

the form of $(an^3 + bn^2 + cn + d)$, where the largest order of magnitude is 3. This pattern continues as nesting of loops continues.

2.4 Posterior Analysis

Posterior Analysis refers to the technique of coding a given solution and then measuring its efficiency. Posterior Analysis provides the actual time taken by the program. This is useful in practice. The drawback of Posterior Analysis is that it depends upon the programming language, the processor and quite a lot of other external parameters.

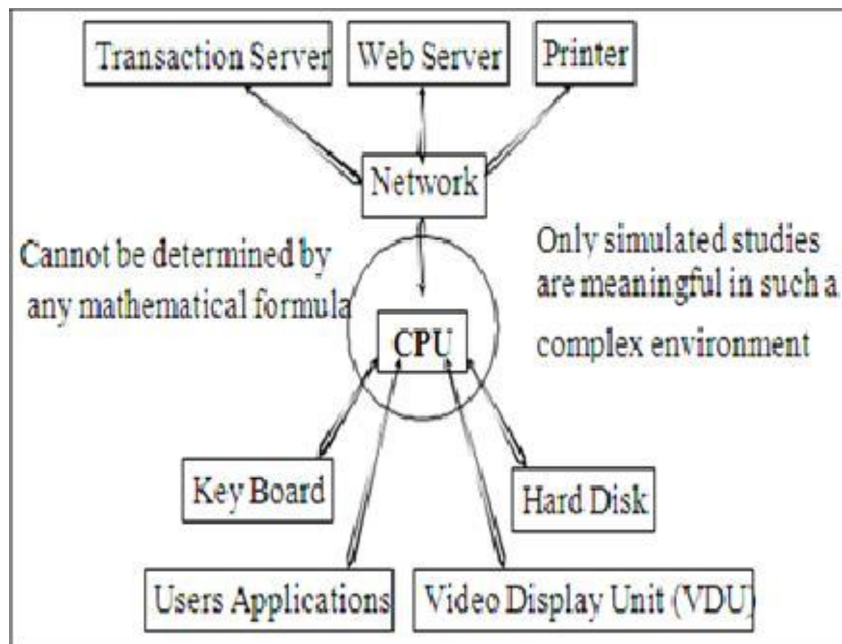


Figure 2.2 Posterior Analysis

The efficiency of a solution can be found by implementing it as program and measuring the time taken by the code to give the desired output. The amount of memory occupied by the code can also be calculated. This kind of analysis is called Posterior Analysis. Posterior means *"later in time"*. The analysis is done after coding the solution and not at the design phase. We could do a posterior analysis for the different possible solutions of a given problem. The inherent drawback of posterior analysis is highlighted in the following: Consider a complex system as shown in figure 2.2. The efficiency in terms of a time taken by a program depends on the CPU's performance apart from other factors. Suppose a given problem has more than one solution and we need to find out the better one using posterior analysis. In such a complex system it is difficult to analyze these solutions practically because there could interrupts to the CPU when one of the solutions is being analyzed in which case

the different solutions may not be comparable.

2.5 Micro Analysis

Perform the instruction count for all operations, to count each and every operation of the program. Micro analysis takes more time and is complex and tedious (Average lines of codes are in the range of 3 to 5 million lines).Those operations which are not dependent upon the size or number of the input will take constant time and will not participate in the growth of the time or space function, So they need not be part of our analysis.

Example:

Power Algorithm (1):

To compute bn:

1. Begin
2. Set p to 1
3. For i = 1… n do:
    3.1 Multiply p by b
4. Output p
5. End

Micro Analysis: Step 2 is an assignment statement which is counted as **one unit**. **Step 3** is a for loop inside in which it performs the multiplication operation (**step 3.1**) which is again counted as **one unit** of operation. As **step 3** executes **n** times and each time it performs **one** multiplication, the total number of operations performed by **step 3** is **n**. **Step 4** is an output statement which is counted as **one unit**. So the worst case complexity for this power algorithm (1) is **(1 + n + 1) = (n+2)**.

2.6 Macro Analysis

Macro Analysis Perform the instruction count only for dominant operations or selective instructions which are costliest. The following are a few examples of dominant (or basic) operations.

- Comparisons and Swapping are basic operations in sorting algorithms.
- Arithmetic operations are basic operations in math algorithms.
- Comparisons are basic operations in searching algorithms.
- Multiplication and Addition are basic operations in matrix multiplication algorithms.

Example:

Power Algorithm (1):

To compute b^n :

1. Begin
2. Set p to 1
3. For i = 1… n do:
    3.1 Multiply p by b
4. Output p
5. End

Macro Analysis: Step 3.1 performs the multiplication operation and this step is performed **n** times. So the worst case complexity (or running time) for this power algorithm (1) is **n**.

2.7 Best, Average and Worst Case Analysis

B**est**, **worst** and **average cases** of a given algorithm express what the resource usage is at least, at most and on average, respectively. Usually the resource being considered is running time, but it could also be memory or other resource. In real-time computing, the worst-case execution time is often of particular concern since it is important to know how much time might be needed in the worst case to guarantee that the algorithm would always finish on time.

Average performance and worst-case performance are the most used in algorithm analysis. Less widely found is best-case performance, but it does have uses, for example knowing the best cases of individual tasks can be used to improve accuracy of an overall worst-case analysis. Computer scientists use probabilistic analysis techniques, especially expected value, to determine expected running times.

The term best-case performance is used in computer science to describe the way an algorithm behaves under optimal conditions. For example, a simple linear search on an array has a worst-case performance $O(n)$ (for the case where the desired element is the last, so the algorithm has to check every element; see Big O notation), and average running time is $O(n)$ (the average position of an element is the middle of the array, i.e. at position $n/2$, and $O(n/2)=O(n)$), but in the best case the desired element is the first element in the array and the run time is $O(1)$.Development and choice of algorithms is rarely based on best-case performance: most academic and commercial enterprises are more interested in improving average performance and worst-case performance. To understand the best, worst and average case we take the example of Insertion Sort.

Example: - It sequentially processes a list of records. Each record is inserted in turn at the correct position.

Best case complexity is *O(n)* and it occurs when the input array is already sorted. Because in this case always the left side element is smaller than the right side element (since the array is sorted) the inner while loop of the insertion sort algorithm (page 47 of course book) will not execute any time. So only due to outer for loop the algorithm becomes linear ie O(n) (suppose n is the size of the array).

Worst case complexity is O(n2) and it occurs when the input array is reversely sorted. Because in this case always the left side element is greater than the right side element the inner while loop will execute every time. So considering both outer for loop and inner while loop the algorithm becomes quadratic ie O(n2).

Average case complexity is O(n2) and it occurs when the input array is randomly sorted. In this case the inner while loop will execute randomly still making the algorithm O(n2) complexity.

Chapter 3
Measure Complexity

3.1 Big *O* Notation

The analysis of the competing data structures and algorithms are approximations of the requirements of the data structures, frequency count analysis of algorithms and data structures is a fairly good if labor intensive way to compare algorithms and data structures. Because it is labor intensive, frequency count analysis has the same drawback of actually writing competing of code and then selecting the best. Because it is labor intensive, frequency count analysis is seldom used to evaluate algorithms and data structures. Instead, frequency count analysis has been presented here as the necessary background material to understand Omicron (or Big *O*) notation in the analysis of data structures and algorithms.

Big *O* notation is used to represent the worst case growth of an algorithm in time or a data structure in space when they are dependent on *n*, where *n* is big enough [5].

The concept of big enough takes effect when *n* is large enough that the differences between the expressions produced by frequency count analysis of competing algorithms becomes, for all practical purposes, completely dependent on the size of *n*. In other words, if two competing algorithms produced the following results for time:

$$5n^2 + 3n + 16 \qquad\qquad 3n^2 + 9n + 7$$

A big enough n means that the differences produced by adding in the constants 16 and 7 are negligible, the differences produced by multiplying n by 3 and 9 are negligible, and even the effects produced by multiplying n^2 by 5 and 3 are negligible. A big enough n means that n^2 will so completely dominate the growth curve of both polynomials that n^2 is all that is needed to compare these two algorithms. In fact, a big enough n means the largest magnitude of n is all that is needed to a priori rates an algorithm or data structure. This means that the results of both examples are n^2 and are, for all practical purposes, identical.

If we accept the premise that the largest magnitude of n is a good enough approximation to evaluate an algorithm or data structure, then all that needs to be

analyzed for time in an algorithm dependent on n is the most frequently occurring statement. All that needs to be analyzed for space is the most frequently occurring increase in the data structure. By accepting this premise, the comparison of algorithms and data structures becomes less precise, but the rating produced is good enough to produce a usable rating for an acceptable amount of work.

The worst case analysis of the growth rate of an algorithm in time or data structure in space that is produced with the above premise is written

O (largest magnitude using *n*) and is pronounced Omicron-expression or Big *O*-expression.

For example, if an algorithm produced a rating of in time of n^3, it would be written $O(n^3)$. Here is a graph of the curves of common Omicron results:

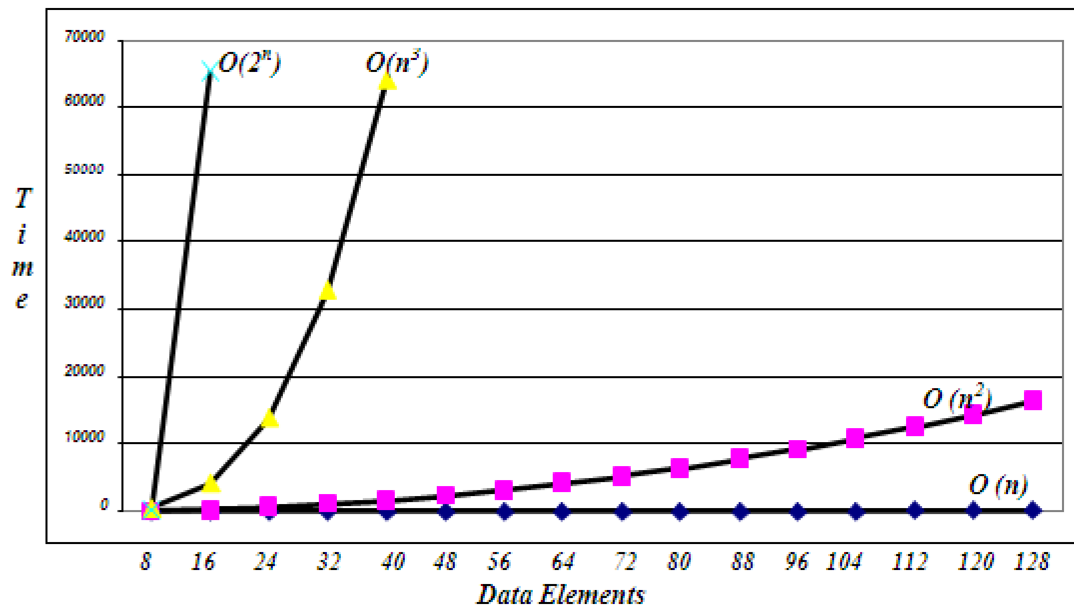


Figure 3.1 Big *O*

Thus far, we have used data structures that at worst O (*n*) and algorithms that are at Worst O (n^2) in terms of their growth based on *n*.

3.2 Theta Notation

Theta notation relates to analyzing an algorithm and finding the number of operations it performs. At this stage we won't specify what operation we are counting. It could be additions, multiplications, matrix multiplications, recursive calls, etc. All we know is that there is some operation that we count, and the number of operations performed will depend on the size of the problem, *n* [6]. The measure of the size of the problem, *n*, may itself be complicated. For example *n* might be the number of items to be

sorted, or the size of a matrix. For now, let us just assume that *n* measures the size of the problem, and that f (*n*) is the number of operations needed to solve the problem when the input size is *n*. The faster *f*(*n*) grows, the slower the algorithm will be, because the number of operations, i.e. the amount of work, performed by the algorithm is growing rapidly. We try to gain an understanding of how fast *f*(*n*) grows with *n*. In fact what we are seeking, technically speaking is upper and lower bounds, so that we can say "*f*(*n*) grows no faster than *x*" and "*f*(*n*) grows faster than *y*". If possible we try to get *x* and *y* to differ by no more than a constant multiple, for then we have considerable knowledge about how fast *f*(*n*) grows. The formal definition of theta notation is a two part definition. We seek first of all an upper bound, some function *g*(*n*), say. We then say that *f*(*n*) is of order at most *g*(*n*), or *f* (*n*) = O(*g*(*n*)) . Technically, if we can find constants *c*1 and *n*1 with 1 | *f* (*n*) |£ *c* | *g*(*n*) | for all *n*³*n*1, then we say *f* (*n*) = O(*g*(*n*)).

3.3 Omega Notation

- f(n) = Ω(g(n)) if there are positive constants c and n0 such that f(n) >= cg(n) for all n >= n_0. This notation is known as Big-Omega notation
- The Big-Omega notation can be considered as a lower bound for the f(n) which is the actual running time of an algorithm.
- Informally Ω (g(n)) denotes the set of all functions with a larger or same order of growth as g(n). For example, n^2 belongs Ω(n).

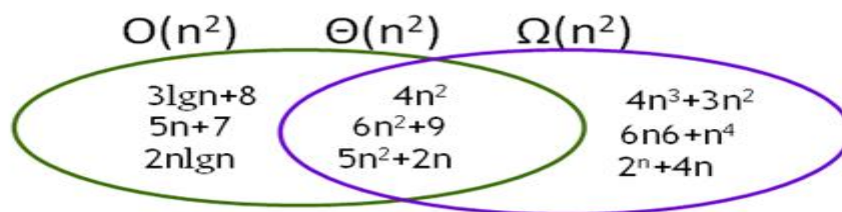


Figure 3.2 Omega Notation

- Consider the set of problems to find the maximum of an ordered set of n integers. Clearly every integer must be examined at least once. So Ω(n) is a lower bound for that. For matrix multiplication we have $2n^2$ inputs so the lower bound can be $\Omega(n^2)$.
- For all sorting & searching we use comparison trees for finding the lower bound.

- For an unordered set the searching algorithm will take Ω(n) as the lower bound. For an ordered set it will take Ω(log n) as the lower bound. Similarly all the sorting algorithms cannot sort in less then Ω(nlogn) time so Ω(nlogn) is the lower bound for sorting algorithms.

3.4 Amortized Analysis

Amortized analysis refers to finding the average running time per operation over a worst-case sequence of operations. Amortized analysis differs from average-case performance in that probability is not involved; amortized analysis guarantees the time per operation over worst-case performance. The method requires knowledge of which series of operations are possible. This is most commonly the case with data structures, which have state that persists between operations. The basic idea is that a worst case operation can alter the state in such a way that the worst case cannot occur again for a long time, thus "amortizing" its cost. As a simple example, in a specific implementation of the dynamic array, we double the size of the array each time it fills up. Because of this, array reallocation may be required, and in the worst case an insertion may require $O(n)$. However, a sequence of n insertions can always be done in $O(n)$ time, so the amortized time per operation is $O(n) / n = O(1)$.Notice that average-case analysis and probabilistic analysis are not the same thing as amortized analysis. In average-case analysis, we are averaging over all possible inputs; in probabilistic analysis, we are averaging over all possible random choices; in amortized analysis, we are averaging over a sequence of operations. Amortized analysis assumes worst-case input and typically does not allow random choices.

There are several techniques used in amortized analysis:[7]

Aggregate analysis determines the upper bound $T(n)$ on the total cost of a sequence of n operations, and then calculates the average cost to be $T(n) / n$.

Accounting method determines the individual cost of each operation, combining its immediate execution time and its influence on the running time of future operations. Usually, many short-running operations accumulate a "debt" of unfavorable state in small increments, while rare long-running operations decrease it drastically.

Potential method is like the accounting method, but overcharges operations early to compensate for undercharges later.

Common Use

- ✓ In common usage, an "amortized algorithm" is one that an amortized analysis has shown to perform well.
- ✓ *Online algorithms* commonly use amortized analysis.

3.5 Accounting Method

The accounting method is a method of amortized analysis based on accounting. The accounting method often gives a more intuitive account of the amortized cost of an operation than either aggregate analysis or the potential method. Note, however, that this does not guarantee such analysis will be immediately obvious; often, choosing the correct parameters for the accounting method requires as much knowledge of the problem and the complexity bounds one is attempting to prove as the other two methods. The accounting method is most naturally suited for proving a $O(1)$ bound on time. The method as explained here is for proving such a bound. Preliminarily, we choose a set of elementary operations which will be used in the algorithm, and arbitrarily set their cost to 1. The fact that the costs of these operations may in reality differ presents no difficulty in principle. What is important is that each elementary operation has a constant cost. Each aggregate operation is assigned a "payment". The payment is intended to cover the cost of elementary operations needed to complete this particular operation, with some of the payment left over, placed in a pool to be used later. The difficulty with problems that require amortized analysis is that, in general, some of the operations will require greater than constant cost. This means that no constant payment will be enough to cover the worst case cost of an operation, in and of itself. With proper selection of payment, however, this is no longer a difficulty, the expensive operations will only occur when there is sufficient payment in the pool to cover their costs [9].

3.6 Potential Method

The potential method is a method used to analyze the amortized time and space complexity of an algorithm. It can be thought of as a generalization of the accounting method the debit method. It is useful in cases where it is hard to assign credit to specific elements of the structure.

The potential function φ is set to be a positive-valued function from states of the data structure in question. If c_i represents the actual cost of the ith operation, which updates

the data structure from state A_{i-1} to state A_i, then the *effective cost* d_i of this operation is defined to be $d_i = c_i + \varphi(A_i) - \varphi(A_{i-1})$ (i.e., the effective cost is the actual cost plus the difference in potential).

If φ is chosen so that the starting state of the data structure has potential 0, then the sum of the effective costs $d_1, \ldots, d_n$ is greater than or equal to the sum of the actual costs $C_1, \ldots, C_n$. So if an upper bound on the effective cost of each operation can be shown, this implies an upper bound on the total cost of n operations, which gives an upper bound on the amortized cost per operation.

Comparison of all the analysis method

Apriori Analysis	Require less effort because we don't actually implement it. Less risky Simple and Easy
Posterior Analysis	Whole effort may go waste if analysis gives negative data. Actually resources or infrastructure is required. Not applicable in many instances.
Micro Analysis	Checks all instruction. It takes more time. Not useful for large code.
Macro Analysis	Check only dominant operation It takes less time comparative to Micro Analysis As useful as micro analysis
Big*O* Notation	Represent worst case growth of an algorithm in time and space when they are dependent on *n*. Big *O* represents an upper bound. Easy to calculate and widely acceptable.
Theta Notation	It represents tight bound. Useful for many problems.
Omega Notation	It represents a lower bound. Not used very often.
Amortized Analysis	Probability is not involved. Guarantees the time per operation over worst-case performance
Accounting Method	It determines the individual cost of each operation. Balancing can be done between heavy and light operations.
Potential Method	Use in Online Algorithms. Good to calculate the overall cost of a data structure.

Table 3.0 Comparison Table

Chapter 4

Problem Statement

The term complexity has been widely used in different contexts by different people. In general, though, system complexity can be described as a measure of how understandable a system is and how difficult it is to perform tasks in the system. A system with high complexity requires great mental or cognitive effort to comprehend and use, while a system with low complexity is easily understood and used. In this section, we attempt to capture some of the aspects of systems that make them difficult to understand. Heterogeneous systems are becoming complex. To determine the complexity of such system is a challenging task.

Research Question

How to calculate the complexity of heterogeneous system? How different algorithms affect the final complexity of system? How they take part in system?

Our goal is to calculate the complexity of heterogeneous system such as online banking system. In such systems there is number of algorithms running in parallel. How they take part in the system?

Justification

There are number of algorithm that contributes in the system's complexity like fault tolerance, authentication and encryption, routing table**, c**ommunication protocol with error recovery mechanism, backup algorithm, recovery algorithm etc. Some algorithms may run in parallel, some may in sequence, how these algorithms affect the overall system complexity. There are various other factors that will play significant roles in the overall complexity. Communication is one of them. Depending upon the distance of locations where servers may be placed due to security or feasibility of the application. Backup is another issue where system needs some time to reconsolidate with the back end links or system. Depending upon the types of backup (hot or cold backup) different timings may be taken up by the system. In case of database application there may be delays due to commit and other updates on various types of storage media, which needs to be taken into account.

In this report we will calculate the complexity of online banking system.

Chapter 5

Heterogeneous System

Heterogeneous system contains many different kinds of hardware and software working together in cooperative fashion to solve problems. There may be many different representations of data in the system. This might include different representations for integers, byte streams, floating point numbers, and character sets. Most of the data can be marshaled from one system to another without losing significance. Attempts to provide a universal canonical form of information is lagging. There may be many different instructions sets. An application compiled for one instruction set cannot be easily run on a computer with another instruction set unless an instruction set interpreter is provided. There is no universal binary making process migration difficult. Recent developments in the web and Java may provide a universal interpreted language on most computers. Though a computer language is not an instruction set, this is a good compromise. Some components in the distributed system may have different capabilities than other components. Among these might include faster clock cycles, larger memory capacity, bigger disk farms, printers and other peripherals, and different services. Seldom are any two computers exactly alike. Heterogeneous System May be Based On Software as well as hardware.eg we have a system in which our data is on site A and on site B ,Site A having the SQL Server and Site B having Oracle. In this scenario if we have same hardware but are platforms are different. In another case we may have different Hardware specifications that will also cover under the type of Heterogeneous system. Let's take an example of banking system it has both different hardware as well as software. Now example illustrates the concept to measure the complexity of a heterogeneous system. This banking application utilizes all kinds of algorithms from symbolic through real time through fault tolerant. Figure 5.1 illustrates a simplified form the architectural building blocks which comprise this banking system, and algorithm classes written to execute on it. The web servers are in turn connected to a set of application servers for implementing banking rules and policies. The application server's access mirrored and/or replicated data storage.

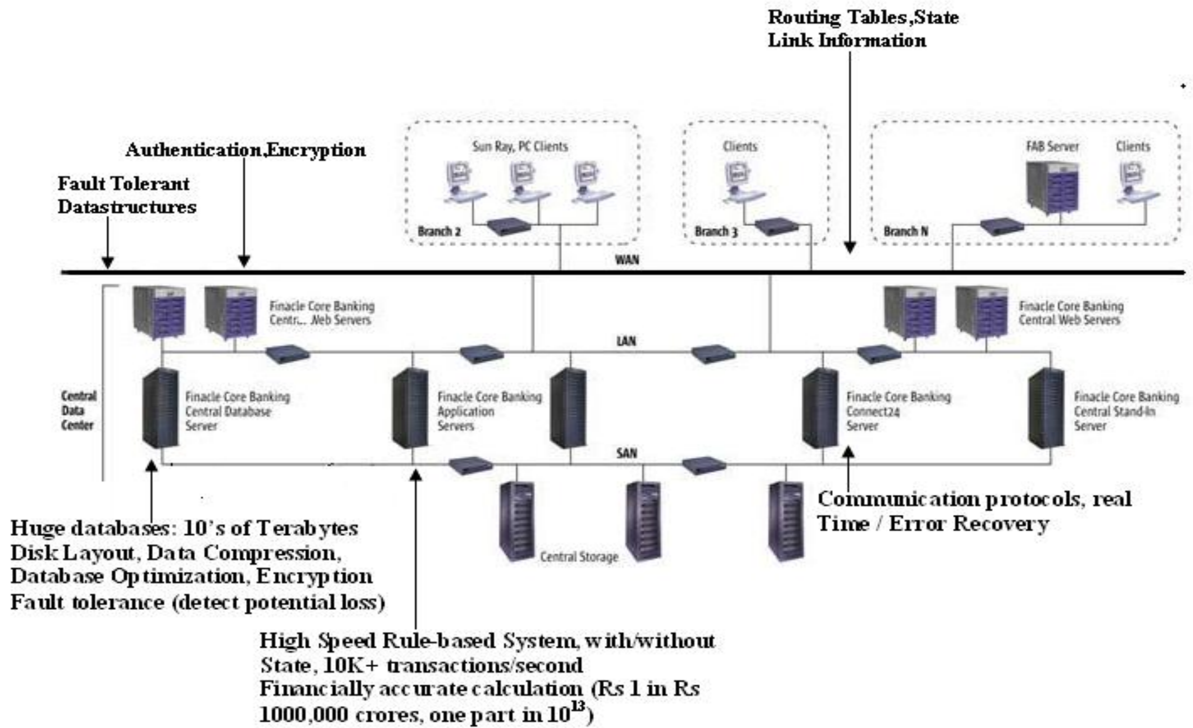


Figure 5.1 Online Banking System

Redundancy is present in the network also. Telephone banking using an Interactive Voice Response System is used as a backup if the branch terminals break down. The design of the authentication hardware and software requires fault tolerance – the users should not have to re-login if one or more server's fail some state should be stored in the form of cookies in non-volatile storage somewhere. The banking calculations require very high accuracy (30+ digit accuracy). Various kinds of fault tolerance schemes are used for storage. For example, two mirrored disks always keep identical data. A write to one disk is not considered complete till the other is written also.

5.1 Encryption

Encryption is the process of converting a plaintext message into cipher text which can be decoded back into the original message. An encryption algorithm along with a key is used in the encryption and decryption of data. There are several types of data encryptions which form the basis of network security. Encryption schemes are based on block or stream ciphers. The type and length of the keys utilized depend upon the encryption algorithm and the amount of security needed. In conventional symmetric encryption a single key is used. With this key, the sender can encrypt a message and a recipient can decrypt the message but the security of the key becomes problematic. In

asymmetric encryption, the encryption key and the decryption key are different. One is a public key by which the sender can encrypt the message and the other is a private key by which a recipient can decrypt the message [8].

5.1.1 Encryption Algorithms

Different encryption algorithms use proprietary methods of generating these keys and are therefore useful for different applications. Here are some nitty gritty details about some of these encryption algorithms. Strong encryption is often discerned by the key length used by the algorithm.

5.1.2 DES/3DES

The Data Encryption Standard (DES) was developed and endorsed by the U.S. government in 1977 as an official standard and forms the basis not only for the Automatic Teller Machines (ATM) PIN authentication but a variant is also utilized in UNIX password encryption. DES is a block cipher with 64-bit block size that uses 56-bit keys. Due to recent advances in computer technology, some experts no longer consider DES secure against all attacks; since then Triple-DES (3DES) has emerged as a stronger method. Using standard DES encryption, Triple-DES encrypts data three times and uses a different key for at least one of the three passes giving it a cumulative key size of 112-168 bits [10].

5.1.3 IDEA

International Data Encryption Algorithm (IDEA) is an algorithm that was developed by Dr. X. Lai and Prof. J. Massey in Switzerland in the early 1990s to replace the DES standard. It uses the same key for encryption and decryption, like DES operating on 8 bytes at a time. Unlike DES though it uses a 128 bit key. This key length makes it impossible to break by simply trying every key, and no other means of attack is known. It is a fast algorithm, and has also been implemented in hardware chipsets, making it even faster.

5.1.4 SEAL

Rogaway and Coppersmith designed the Software-optimized Encryption Algorithm (SEAL) in 1993. It is a Stream-Cipher, i.e., data to be encrypted is continuously encrypted. Stream Ciphers are much faster than block ciphers (Blowfish, IDEA, DES) but have a longer initialization phase during which a large set of tables is done using the Secure Hash Algorithm. SEAL uses a 160 bit key for encryption and is considered very safe [11].

5.1.5 RC4

RC4 is a cipher invented by Ron Rivest, co-inventor of the RSA Scheme. It is used in a number of commercial systems like Lotus Notes and Netscape. It is a cipher with a key size of up to 2048 bits (256 bytes), which on the brief examination given it over the past year or so seems to be a relatively fast and strong cypher. It creates a stream of random bytes and 'XORing' those bytes with the text. It is useful in situations in which a new key can be chosen for each message.

5.2 Authentication

The ability to determine the identity of a party for an interaction and to ensure that a message came from, who it claims to have come from. Authentication is seldom used in isolation. Authentication is used as the basis for authorization [10] (determining whether a privilege will be granted to a particular user or process), privacy (keeping information from becoming known to non-participants), and non-repudiation (not being able to deny having done something that was authorized to be done based on the authentication).

There are three main algorithms for authentication: passwords, Needham and Schroeder protocol (used in Kerberos), and public key encryption. In all of them the central issue is to never allow the secret information outside a secured environment, while at the same time allowing the recipient to verify that the secret was used. The descriptions that appear below only give a flavor of the algorithms and discuss their advantages and disadvantages.

5.2.1 Passwords

Passwords are simply secrets that are provided by the user upon request by a recipient. Passwords are often stored on a server in an encrypted form so that a penetration of the file system does not reveal password lists. The problem with password-based systems is that the password becomes known to the recipient, who can then impersonate the user. Even if the recipient is trusted not to do this, passwords are dangerous in network environments since they are susceptible to interception during transmission. In general, passwords are unacceptable security in a network environment [13].

5.2.2 Needham and Schroeder Protocol

In the Needham/Schroeder protocol used in Kerberos, the secret information used for verification is never transmitted in the clear and is never seen by a recipient. Instead,

an "authentication server" creates a collection of "session secrets" (derived from its knowledge of the secrets of the sender and receiver in a particular interchange) that are used by the sender and receiver for authentication of messages during a particular interaction. Session information is good only between session participants, and can be time stamped to protect against replaying of messages. New interactions (even between the same client and server) require new session keys [11].

5.2.3 Public Key Encryption

Public key encryption can also be used for authentication using "digital signatures". In public key encryption, each user, i, has both a public key, Ei, which is made publicly available, and a private key, Di, which only user i knows. The keys are mathematically related, and both are generated by the user. Thus, there is no need for anyone else to hold the private key, which enhances Security Public and private keys are inverses and are symmetrical, in the sense that for a given message m, Ei(Di(m)) = Di(Ei(m)) = m. To preserve privacy, a user X will obtain the public key Ey for user Y and compute Ey(m)). Since only Y knows Dy, only Y can decrypt. A checksum or some other identifying pattern is embedded into m so that a valid decryption can be verified. Digital signatures work similarly, except that when X wants to sign a message to Y, X uses his/her private key Dx and computes Dx(m). Upon receipt, Y computes Ex(Dx(m)) = m. Since only X had knowledge of Dx, only X could have signed the message. Privacy encryption can be combined with digital signatures by computing Ey(Dx(m)), which is decrypted as Ex(Dy(Ey(Dx(m)))) = m. The public key register of the Ei need not be read secure, since the Ei are given away freely. The registry must be protected against corruption, since that would allow fraudulent keys to be given out. The channel to the registry must be secure to prevent "spoofing" attacks, but this can be done using public key encryption.
The disadvantage of public key encryption is that it is several orders of magnitude slower than conventional encryption because of the nature of the encryption algorithms. Thus, for instances where a session involves many messages or where high performance is required, a Kerberos-based system may be more appropriate [14].

5.2.4 Authentication Algorithm

Authentication is the process of verifying identity so that one entity can be sure that another entity is who it claims to be. In this Alice and Bob, public key cryptography is

easily used to verify identity. The notation {something} key means that something has been encrypted or decrypted using key.

Suppose Alice wants to authenticate Bob. Bob has a pair of keys, one public and one private. Bob discloses to Alice his public key. Alice then generates a random message and sends it to Bob:

A->B random-message

Bob uses his private key to encrypt the message and returns the encrypted version to Alice:

B->A {random-message} bobs-private-key

Alice receives this message and decrypts it by using Bob's previously published public key. Alice compares the decrypted message with the one originally sent to Bob; if they match, Alice knows talking to Bob. An imposter presumably wouldn't know Bob's private key and would therefore be unable to properly encrypt the random message for Alice to check., instead of encrypting the original message sent by Alice, Bob constructs a message digest and encrypts that. A message digest is derived from the random message in a way that has the following useful properties:

The digest is difficult to reverse. Someone trying to impersonate Bob couldn't get the original message back from the digest. An impersonator would have a hard time finding a different message that computed to the same digest value. By using a digest, Bob can protect himself. He computes the digest of the random message sent by Alice and then encrypts the result. He sends the encrypted digest back to Alice. Alice can compute the same digest and authenticate Bob by decrypting Bob's message and comparing values.

The technique just described is known as a digital signature. Bob has signed a message generated by Alice, and in doing so he has taken a step that is just about as dangerous as encrypting a random value originated by Alice. Consequently, our authentication protocol needs one more twist: some (or all) of the data needs to be originated by Bob.

A->B hello, are you bob?

B->A Alice, This is bob

{Alice is Bob} bobs-private-key

When he uses this protocol, Bob knows what message he is sending to Alice, and he doesn't mind signing it. He sends the unencrypted version of the message first, "Alice,

This Is Bob." Then he sends the digested-encrypted version second. Alice can easily verify that Bob is Bob, and Bob hasn't signed anything he doesn't want to.

How does Bob hand out his public key in a trustworthy way? Let's say the authentication protocol looks like this:

A->B	hello
B->A	Hi, I'm Bob, bobs-public-key
A->B	prove it
B-A	Alice, This is bob

{Alice, this is Bob}Bobs-private-key

With this protocol, anybody can be Bob. All you need is a public and private key. You lie to Alice and say you are Bob, and then you provide your public key instead of Bob's. Then you prove it by encrypting something with the private key you have, and Alice can't tell you're not Bob.

To solve this problem, the standards community has invented an object called a certificate. A certificate has the following content:

- ✓ The certificate issuer's name.
- ✓ The entity for whom the certificate is being issued.
- ✓ The public key of the subject.
- ✓ Some time stamps.

The certificate is signed using the certificate issuer's private key. Everybody knows the certificate issuer's public key (that is, the certificate issuer has a certificate, and so on...). Certificates are a standard way of binding a public key to a name.

By using this certificate technology, everybody can examine Bob's certificate to see whether it's been forged. Assuming that Bob keeps tight control of his private key and that it really is Bob who gets the certificate, then all is well. Here is the amended protocol:

A->B	hello
B->A	Hi, I'm Bob, bobs-public-key
A->B	prove it
B-A	Alice, This is bob
	{Alice, this is Bob}Bobs-private-key

Now when Alice receives Bob's first message, she can examine the certificate, check the signature (as above, using a digest and public key decryption), and then check the

subject (that is, Bob's name) and see that it is indeed Bob. She can then trust that the public key is Bob's public key and request Bob to prove his identity. Bob goes through the same process as before, making a message digest of his design and then responding to Alice with a signed version of it. Alice can verify Bob's message digest by using the public key taken from the certificate and checking the result.

A bad guy - let's call him Mallet - can do the following:

A->M hello

M->A Hi, I'm Bob, bobs-certificate

A->M prove

M->A ????

But Mallet can't satisfy Alice in the final message. Mallet doesn't have Bob's private key, so he can't construct a message that Alice will believe came from Bob. Once Alice has authenticated Bob, she can do another thing - she can send Bob a message that only Bob can decode:

A->B {secret} bobs-public-key

The only way to find the secret is by decrypting the above message with Bob's private key. Exchanging a secret is another powerful way of using public key cryptography. Even if the communication between Alice and Bob is being observed, nobody but Bob can get the secret.

This technique strengthens Internet security by using the secret as another key, but this time it's a key to a symmetric cryptographic algorithm (such as DES, RC4, or IDEA). Alice knows the secret because she generated it before sending it to Bob. Bob knows the secret because Bob has the private key and can decrypt Alice's message. Because they both know the secret, they can both initialize a symmetric cipher algorithm and then start sending messages encrypted with it. Here is a revised protocol:

A->B hello

B->A Hi, I'm Bob, bobs-certificate

A->B {Alice, This is Bob} bobs-private-key

B->A ok bob,here is asecret {secret}bobs-public-key

{some message}secret-key

How secret-key is computed is up to the protocol being defined, but it could simply be a copy of secret. Alice and Bob can introduce a message authentication code (MAC) into their protocol. A MAC is a piece of data that is computed by using a secret and

some transmitted data. The digest algorithm described above has just the right properties for building a MAC function that can defend against Mallet:

MAC:= Digest[some message, secret]

Because Mallet doesn't know the secret, he can't compute the right value for the digest. Even if Mallet randomly garbles messages, his chance of success is small if the digest data is large. For example, by using MD5 (a good cryptographic digest algorithm invented by RSA), Alice and Bob can send 128-bit MAC values with their messages. The odds of Mallet's guessing the right MAC are approximately 1 in 18,446,744,073,709,551,616 - for all practical purposes, never.

Here is the sample protocol, revised yet again:

A->B	hello
B->A	Hi,I'mBob, bobs-certificate
A->B	prove it
B->A	Alice, This Is bob
	{ digest[Alice, This Is Bob] } bobs-private-key
	ok bob, here is a secret {secret} bobs-public-key
	{some message,MAC}secret-key

Complexity Analysis:-

Input

A has array of n numbers

B has array of n numbers

Output

Median of all 2n numbers, Numbers are O(log n) bits long

A sends all of his numbers to B

B calculates median of all 2n numbers

Cost:- Each number is O(log n) bits, n numbers are sent [15]

Total cost is O (n*log n) bits.

Naive Solution

1. A sorts his array and sends his median (M_A) to B
2. B sorts his array and sends his median (M_B) to A.
3. If $M_A = M_B$ then return $M_A(= M_B)$.
4. If $M_A > M_B$ then A throws top (n/2) elements B throws low (n/2) elements
5. Otherwise, vice versa We reduces the size of the problem by half

6. Back to step 1, until size of arrays = 1

We will repeat this algorithm until the size of the array will be 1, while every loop the array is cut in half, and log n bits transferred [15].

Total cost is $CC_{mid} = O(\log^2 n)$ bits.

5.3 Routing and Routing Algorithms

Routing is the process of moving information packets and messages across a network from a source host to a destination host. Along the way, at least one router is encountered. Routing involves two basic activities: determining optimal routing paths and transporting message packets through a network. Routing takes place at the Network Layer—Layer 3 in the OSI 7 layers model. Routed protocols such as IP and IS-IS are the layer 3 protocols, which include the source and destination addresses of the packets. Routing protocols such as OSPF and BGP are used to evaluate what path will be the best for a packet to travel. Routing tables contain information used by switching software to select the best route. Destination/next hop associations tell a router that a particular destination can be reached optimally by sending the packet to a particular router representing the "next hop" on the way to the final destination. When a router receives an incoming packet, it checks the destination address and attempts to associate this address with a next hop. Routing tables may also contain other information, such as data about the desirability of a path.

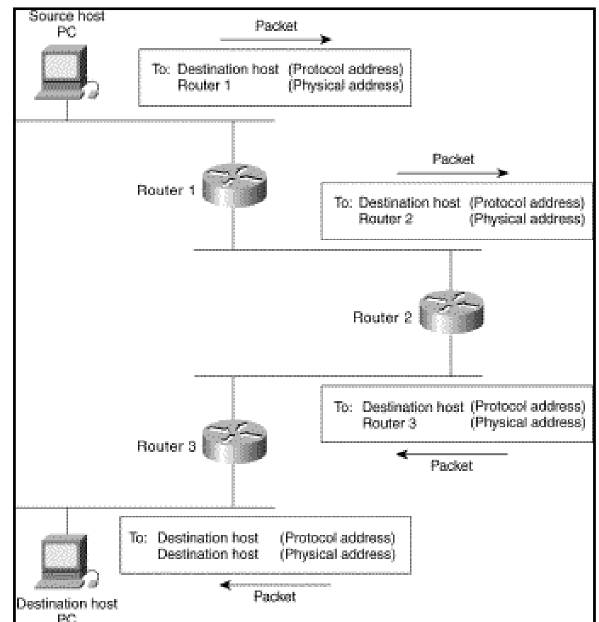


Figure 5.2 The Routers in a network and the Routing Process [23]

5.3.1 Distance Vector Algorithms

A distance vector algorithm uses metrics known as costs to help determine the best path to a destination. The path with the lowest total cost is chosen as the best path.

When a router utilizes a distance vector algorithm, different costs are gathered by each router. These costs can be completely arbitrary, administrator-assigned numbers, such as five. Although the number five might not be of any significance to an outside

observer, the administrator might have assigned it to a particular link to represent the reliability of that link. Costs can also be dynamically gathered values, such as the amount of delay experienced by routers when sending packets over one link as opposed to another. All the costs (assigned and otherwise) are compiled and placed within the router's routing table. All the costs gathered are then used by the algorithm to calculate a best path for any given network scenario. Although there are many resources that will offer complex mathematical representations of what distance vector algorithms are and how they compute their decisions, the core concept remains the same—by adding the metrics for every optional path on a network, you will come up with at least one best path. The formula for this is as follows:

$M(i,k) = \min [M(i,t) + M(t,k)]$

This formula states that the best path between two networks (M(i,k)) can be found by finding the lowest (min) value of paths between all network points. Plugging this information into the formula, we see that the route from A to B to C is still the best path:

$5(A,C) = \min[2(A,B) + 3(B,C)]$

Whereas the formula for the direct route A to C looks like this:

$6(A,C) = \min[6(A,C)]$

Distance Vector Algorithms

Multiple instances of (some variant of) the BF Algorithm from each node:

- Each node is the source in one instance
- n instances run in parallel (n = #nodes)

Execution: Send distance vector when changes detected or upon timeout.

- Complexity: message size $O(n)$, space $O(n)$
- for path-vector both $O(\text{Diam}\cdot n)$ (Diam = max path length)

5.3.2 Link-State Algorithms

Link-state algorithms work within the same basic framework that distance vector algorithms do in that they both favor the path with the lowest cost. However, link-state protocols work in a somewhat more localized manner. Whereas a router running a distance vector algorithm will compute the end-to-end path for any given packet, a link-state protocol will compute that path as it relates to the most immediate link. That is, where a distance vector algorithm will compute the lowest metric between Network A and Network C, a link-state protocol will compute it as two distinct paths,

A to B and B to C. This process is best for larger environments that might change fairly often. Link-state algorithms enable routers to focus on their own links and interfaces. Any one router on a network will only have direct knowledge of the routers and networks that are directly connected to it (or, the *state* of its own *links*). In larger environments, this means that the router will use less processing power to compute complicated paths. Another advantage to such localized routing processes is that protocols can maintain smaller routing tables. Because a link-state protocol only maintains routing information for its direct interfaces, the routing table contains much less information than that of a distance vector protocol that might have information for multiple routers.

In distance vector, router knows only distance to each destination

- Hides much information
- Link state: Let each router know entire network topology and compute routes locally

Key elements

- Topology dissemination
- Route computation

Complexity of Link state Algorithms

Denote m = # links.

- Communication complexity: message size O(m),
- Space complexity: O(m)

5.3.3 Complexity of routing algorithm.

The Task of routing algorithm is to determine the suitable port for delivering the message addressed to one network node and aim is to have the fastest routing available in the network. The case of every node sending the message to each other by correct delivery can be considered as optimal one. This can be achieved by keeping the table of size $O(n)$ by every node, where n is the number of nodes, but it's not suitable for larger networks. To ensure effective routing with the best possible throughput it is important to indentify and analyze complexity of routing strategies.[17]

Routing Table: - The principle of routing with routing tables is straight forward. Every node keeps a table with entries for each other node. Using these entries, it could be determined via which outgoing port a message had to be sent.

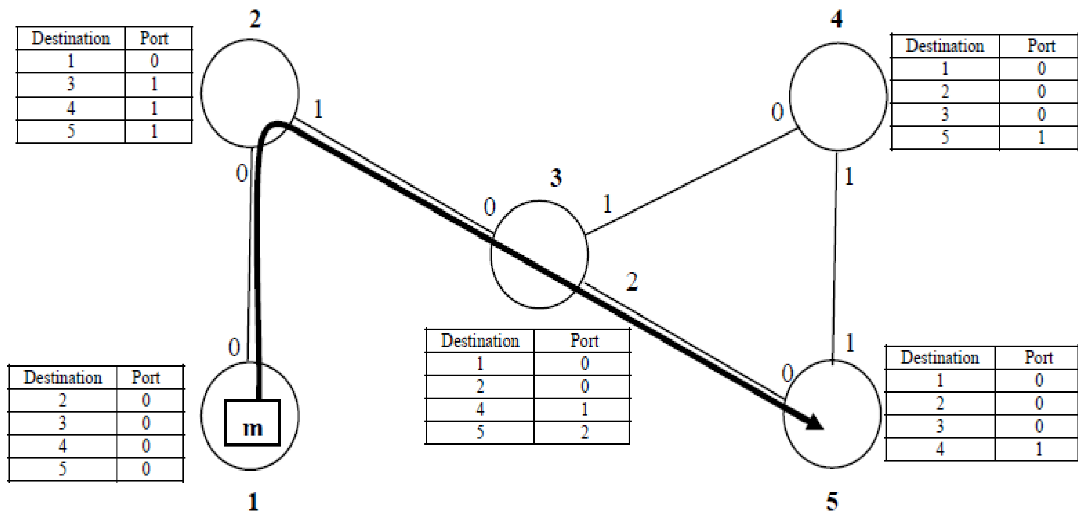


Figure 5.3 Routing Tables (message m destined to node 5)[17]

Routing with routing tables is today a dominant strategy. There are lots of methods to reduce the memory demand for the data holding required for routing. The aggregation is one of the most important. It is based on that we don't hold information for each node in network but only for sub-networks to which nodes belong. This way one can excessively reduce the size of routing tables.

Complexity Analysis

Algorithm 1 contains a description of a routing algorithm with routing tables in pseudo code. Graph=(V,E) with maximum degree Δ.

Algorithm 1.

The algorithm is understood for routing in node $u \in V$.

Input: destination node d, message m.

A data structure *table* [] is a field which implements the routing table. Its size is n and it stores identification labels of ports which are positive integers. It is organized in such a manner that in i^{th} position is stored an ID-number of port via which a message addressed to a node $w \neq u$ is forwarded.

```
receive (m, d);
if d = = u then OK
else
        p = table[d];
        send (m, d) to p;
```

Lemma 1. To store a positive integer j it is required a register with at least $[\log_2 j]$ bits.

Proof. It is possible to vary at most j = 2n values in n-bit register. Taking logarithm of j=2n resulted in log2j = n. It means for storing the j value is an n bit register necessary. Indeed, $[log_2\ j]$ bits are necessary, since the number of bits must be an integer.

Theorem 1. Routing with routing tables requires O (n logΔ) bits to store the routing information.

Proof. The size of registers in RAMs depends on the number stored in it[3]. We need $[\log_2 \Delta]$ log bits[4] to keep port numbers. We need to store n such values. Hence the memory requirements are n $[\log_2 \Delta]$ = O $(n\ log\ \Delta)$ bits. Using routing tables represented by data structure described in previous section the process of finding the corresponding port is very fast. The time of executing the algorithm doesn't depend on network parameters. It is a constant time performance.

Theorem 2. Time complexity of Algorithm 1 in RAM model with unit cost criterion Is $O(1)$.

Proof. Algorithm 1 consists of 4 operations. It is necessary to carry out exactly these 4 = O (1) operations for optional input, i.e. constant amount, independent on input.

Interval Routing

Interval routing is a space efficient routing strategy used in computer network. This method gaining a great interest because of accessible implementation chips available on the market. The basis of interval routing idea is to all of the nodes with integer from one set(for example {1,2,3,....,n}) and labelling of all arcs with interval(in range of number of nodes). These intervals means that a message addressed to a node labelled with u, via a port labelled with interval which includes a node u is forwarded.

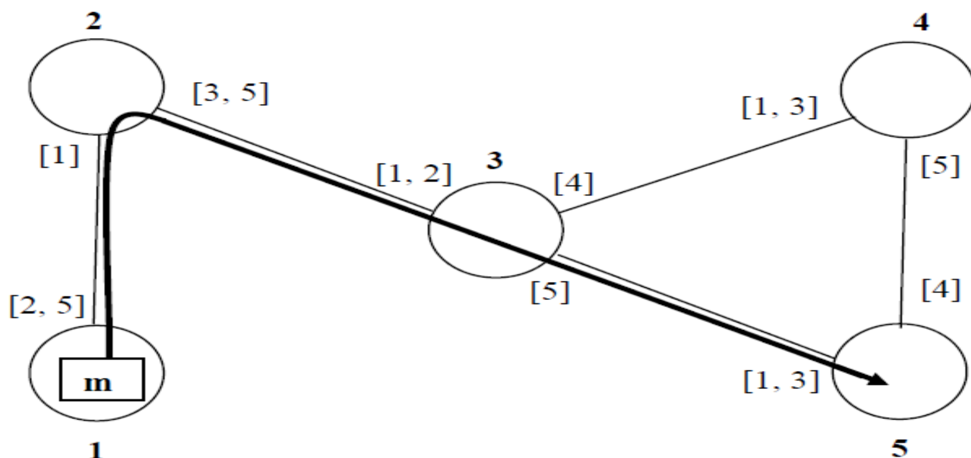


Figure 5.4 Interval routing (message m destined to node 5)[17]

Definition 1:

Interval labeling scheme (ILS) of graph $G = (V, E)$ is a scheme, where:

1. A node labeling is an assignment of unique labels to nodes of V
2. For each node v ∈ V, an edge labeling is an assignment of disjoint intervals to arcs e ∈ I(v), where I(v) denotes the set of arcs outgoing from v.

An interval routing scheme, denoted also IRS, in a graph is a valid ILS. The validity means that the routing strategy guarantees that the message will always be delivered to its destination using that ILS.

Proposition 1. The validity of every ILS can be checked in O(n2) time, where n is the number of nodes of the graph. Intervals used in labeling are understood as cyclic: [a, b] = {a, a + 1,…, V, 1, …, b} for a > b. In case of IRS using only linear interval I = [a, b], where a = b, it is called a linear (LIRS). If the labels of all outgoing arcs are grouped into at most k intervals then the scheme is denoted as k-IRS. The memory requirements of interval routing schemes depends also on k. The Compactness of a graph G is the smallest integer k such that G supports k-IRS.[16]

	Routing Tables	*Interval routing*
Space complexity	*O(n log Δ)*	*O(k Δ log n)*
Time complexity	*O(1)*	*O(log(k Δ))*

Table 5.0 Routing Complexity

5.4 Concurrency Control and Algorithms

Concurrency control in database management systems (DBMS) ensures that database transactions are performed concurrently without the concurrency violating the data integrity of a database. Executed transactions should follow the ACID rules, as described below. The DBMS must guarantee that only serializable (unless serializability is intentionally relaxed), recoverable schedules are generated. It also guarantees that no effect of committed transactions is lost, and no effect of aborted (rolled back) transactions remains in the related database.

5.4.1 Transaction ACID rules

Atomicity - Either the effects of all or none of its operations remain when a transaction is completed - in other words, to the outside world the transaction appears to be indivisible, atomic.

Consistency - Every transaction must leave the database in a consistent state.

Isolation - Transactions cannot interfere with each other. Providing isolation is the main goal of concurrency control.

Durability - Successful transactions must persist through crashes.

The main categories of concurrency control mechanisms are:

Optimistic - Delay the synchronization for a transaction until its end without blocking (read, write) operations, and then abort transactions that violate desired synchronization rules.

Pessimistic - Block operations of transaction that would cause violation of synchronization rules. Quite many methods for concurrency control exist. Major methods, which have each many variants, include:

1. Two-phase locking
2. Timestamp ordering

5.5 Fault -tolerance

Fault-tolerance or graceful degradation is the property that enables a system (often computer-based) to continue operating properly in the event of the failure of (or one or more faults within) some of its components. If its operating quality decreases at all, the decrease is proportional to the severity of the failure, as compared to a naively-designed system in which even a small failure can cause total breakdown. Fault-tolerance is particularly sought-after in high-availability or systems. Fault-tolerance is not just a property of individual machines; it may also characterize the rules by which they interact. For example, the Transmission Control Protocol (TCP) is designed to allow reliable two-way communication in a packet-switched network, even in the presence of communications links which are imperfect or overloaded. It does this by requiring the endpoints of the communication to *expect* packet loss, duplication, reordering and corruption, so that these conditions do not damage data integrity, and only reduce throughput by a proportional amount.

Recovery from errors in fault-tolerant systems can be characterized as either roll-forward or roll-back. When the system detects that it has made an error, roll-forward recovery takes the system state at that time and corrects it, to be able to move forward. Roll-back recovery revert the system state back to some earlier, correct version, for example using check pointing, and moves forward from there. Roll-back recovery requires that the operations between the checkpoint and the detected erroneous state can be made idempotent. Some systems make use of both roll-forward and roll-back

recovery for different errors or different parts of one error. Within the scope of an individual system, fault-tolerance can be achieved by anticipating exceptional conditions and building the system to cope with them, and, in general, aiming for self-stabilization so that the system converges towards an error-free state. However, if the consequences of a system failure are catastrophic, or the cost of making it sufficiently reliable is very high, a better solution may be to use some form of duplication.

5.5.1 Checkpoint Recovery Method

Checkpoint-Recovery is a common technique for imbuing a program or system with fault tolerant qualities, and grew from the ideas used in systems which employ transaction processing. It allows systems to recover after some fault interrupts the system, and causes the task to fail, or be aborted in some way. While many systems employ the technique to minimize lost processing time, it can be used more broadly to tolerate and recover from faults in a critical application or task. The basic idea behind checkpoint-recover is the saving and restoration of system state. By saving the current state of the system periodically or before critical code sections, it provides the baseline information needed for the restoration of lost state in the event of a system failure. While the cost of checkpoint-recovery can be high, by using techniques like memory exclusion, and by designing a system to have as small a critical state as possible may minimize the cost of checkpointing enough to be useful in even cost sensitive embedded applications.[19]

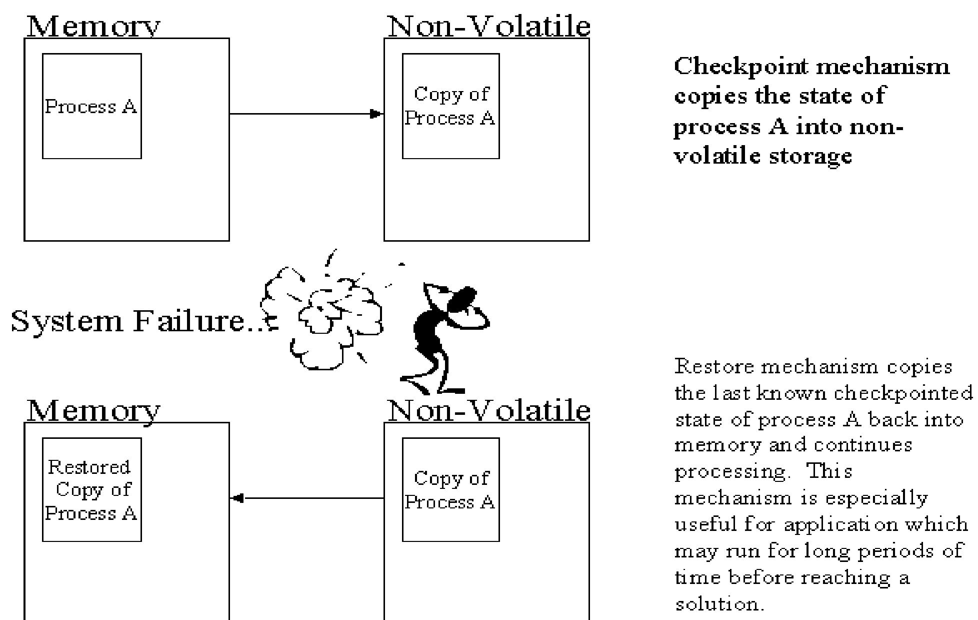


Figure 5.5 Checkpoint Mechanism [19]

When a system is checkpointed, the state of the entire system is saved to non-volatile storage. The checkpointing mechanism takes a snapshot of the system state and stores the data on some non-volatile storage medium. Clearly, the cost of a checkpoint will vary with the amount of state required to be saved and the bandwidth available to the storage mechanism being used to save the state.

In the event of a system failure, the internal state of the system can be restored, and it can continue service from the point at which its state was last saved. Typically this involves restarting the failed task or system, and providing some parameter indicating that there is state to be recovered. Depending on the task complexity, the amount of state, and the bandwidth to the storage device this process could take from a fraction of a second to many seconds.

This technique provides protection against the transient fault model. Typically upon state restoration the system will continue processing in an identical manner as it did previously. This will tolerate any transient fault, however if the fault was caused by a design error, then the system will continue to fail and recover endlessly. In some cases, this may be the most important type of fault to guard against, but not in every case.[20]

Complexity Analysis: - The worst case message complexity of the check pointing algorithm is $O(kn)$ when k initiators initiate concurrently. The time complexity is $O(n)$. For the recovery algorithm, time and message complexities are both $O(n)$.[18]

Chapter 6
Results & Discussion

In online banking system we have following modules:-

1) Fault tolerance and Recovery
2) Authentication and encryption
3) Routing
4) Backup
5) Communication

6.1 Fault Tolerance and Recovery Algorithm Complexity: - Check pointing with rollback recovery is a well-known method for achieving fault-tolerance in distributed systems. The check pointing algorithms can handle multiple concurrent initiations by different processes. While taking checkpoints, processes do not have to take into consideration any application message dependency. The synchronization is achieved by passing control messages among the processes. Application messages are acknowledged. Each process maintains a list of unacknowledged messages. Here a logical checkpoint is being used, which is a standard checkpoint (i.e., snapshot of the process) plus a list of messages that have been sent by this process but are unacknowledged at the time of taking the checkpoint. The worst case message complexity of the check pointing algorithm is $O(kn)$ when k initiators initiate concurrently. The time complexity is $O(n)$. For the recovery algorithm, time and message complexities are both $O(n)$[18].

6.2 Authentication Algorithm Complexity:- In Alice and Bob Protocol Cost of each number is O(log n) bits, n numbers are sent so, Total cost is O(n*log n) bits[15].

6.3 Routing Complexity: - In a communication network routing is one of the latency sources. Routing algorithm has task to determine the suitable port for delivering message address to one network node. Correct delivery of message from source node to destination node is considered as optimal routing. The aim is to have the fastest and optimal routing available in the network.

The principle of routing with routing tables is straightforward. Ever node keeps a table with entries for each node. Using these entries, it could be determined via which outgoing port a message had to be sent. Interval routing is a space efficient routing strategy used in computer networks. The basis of interval routing idea is to label all of

the nodes with integer from one set (for example {1, 2, 3...*n*}) and labeling of all arcs with interval (in range of number of nodes). These intervals means that a message addressed to a node labeled with *u*, via a port labeled with interval which includes a node *u* is forwarded [16].

	Routing Tables	*Interval routing*
Space complexity	$O(n \log \Delta)$	$O(k \Delta \log n)$
Time complexity	$O(1)$	$O(\log(k \Delta))$

Table 6.1 Complexity of message passing in wide area networks**.**

Note: - Δ is maximum degree of the graph, n is number of nodes in the graph, k is interval

6.4 <u>Backup Algorithm Complexity:-</u> The backup tree algorithm to compute a set of n-1 backup multicast delivery trees from the default multicast tree for application level multicast. For each backup multicast tree exactly one link of the default multicast tree is replaced by a backup link from the set of available links. The backup tree algorithm calculates the n-1 trees with a complexity of O (m log n)[17].

Note: - m is number of messages and n are number of nodes (systems)

6.5 <u>Communication Complexity</u>:- It depends upon the algorithm used in system for Distance Vector routing scheme complexity will be $O(m)$.In case of Link state Algorithm complexity will be $O(n)$.

<u>Module wise Complexity Analysis</u>

Communication	Fault Tolerance	Recovery	Authentication	Routing	Backup
$O(m)$	$O(kn)$	$O(n)$	$O(n*\log n)$	$O(1)$	$O(m \log n)$

Table 6.2: Complexity of all Modules

The above shown results are just an approximation, complexity may vary according to the algorithm .There are number of things that can affect the system, front end and network speed plays a vital role in this. In this type of system any algorithm may change the complexity of the system. There are number of algorithms available for particular task. In above case if we choose Open Ear Decompositions for fault tolerance routing scheme then complexity will be $O(m\ )$ for that particular module [21]. If we change routing scheme in the system then also the complexity will be

change like we may switch to distance vector routing then the communication complexity will be $O(n)$ instead of $O(m)$.

Communication	Fault Tolerance	Recovery	Authentication	Routing	Backup
$O(n)$	$O(m)$	$O(n)$	$O(n*\log n)$	$O(1)$	$O(m \log n)$

Table 6.3. System Complexity

As the user login first the authentication is being done then routing protocol comes in to existence ,then depends upon the mode of transaction, the user may only view the account statement in this case the complexity will be $\{O(n*\log n) + O(1)+O(m)\&\& O(kn)\}$,authentication , routing will run in sequence viewing account statement fault tolerance will run in parallel. If user wants to transfer the money from one account to another then complexity will be different approximate $\{O(n*\log n)+O(n)+O(1)+O(n)+[O(m)\&\&O(m\log n)]\}$.

Chapter 7

Future Work and Conclusion

The size and complexity of heterogeneous systems have been increasing inexorably in the recent past. We are currently trying to evaluate online banking system using the complexity metrics. In particular, we are trying to evaluate large, complex heterogeneous system like ubiquitous computing systems, enterprise systems and internet-based systems. This would help establish the generality of the metrics. Some preliminary evaluation in this regard as well as a description of some of the strategies for alleviating complexity in a ubiquitous computing system is discussed in [22].

Another area of future work is to try to extend the set of metrics to include a measure for debugging and failure-recovery. Heterogeneous systems may fail due to a number of reasons - OS faults, software bugs, hardware errors, human error, etc. While self-repair and high-level programming can reduce the occurrence of failures and provide automated, graceful failure-recovery, systems may still fail and require human intervention. Besides, self-configuring and self-repairing systems may misbehave or take wrong actions. Hence, heterogeneous systems should provide mechanisms for uncovering the different layers of abstraction and deducing the cause of faults. The complexity of debugging a system is an important measure of the ease of using and maintaining the system. System complexity forms just one of the factors that influence the productivity of developers, administrators and end-users. Other factors that affect complexity include communication changing requirements, technology churn, etc. While we are still far from being able to quantify these other factors, they would all form part of any evaluation of the productivity of different kinds of users.

Heterogonous systems are becoming bigger and more complex. While the complexity of large-scale heterogeneous systems has been acknowledged to be an important challenge, there has not been much work in defining or measuring system complexity. Thus, today, it is difficult to compare the complexities of different systems, or to state that one system is easier to program, to manage or to use than another.. In this report, we try to understand the factors that cause heterogeneous systems to appear very complex to people. We define different aspects of system complexity and propose metrics for measuring these aspects. In conclusion, we have tried to formalize the notion of complexity of heterogeneous systems. We have proposed a number of

metrics to measure different aspects of system complexity. In this report, we have study about various methods of analysis and notations like apriori analysis, posterior analysis, micro analysis, macro analysis, amortized analysis, big *O* notation, theta notation, potential method, accounting method. All analysis and methods are situation specific. Our report does a comparative study of various method and techniques of algorithm analysis giving their specific advantages and disadvantages. First of all we have analyzed the problem in detail, then study the algorithm in context to the online banking system.

Generally people ask the question: How to calculate the complexity of heterogeneous system? How different algorithms affect the final complexity of system? How they take part in system? These questions don't have an easy or unambiguous answer, however. The complexity depend quite heavily on the environment what the system is?

Calculating the complexity of this type of system is just an approximation because complexity may vary on the mode of transaction. One algorithm may affect the system. In this system there are various factors that will play significant roles in the overall complexity like one minor change may increase or decrease the complexity of the system e.g. if we change authentication algorithm then the overall system complexity will affected. There are many algorithms that are running simultaneously as we discussed earlier like fault tolerance, authentication, encryption, routing tables, communication protocols and for error recovery. Every algorithm contributes in the system, so every algorithm have time and space complexity, if the algorithms are running parallel then the complexity will be highest out of them. E.g. we have complexities n, n+1, n2 then the complexity will be $n^{2.}$ If we have N algorithms and they are running sequentially then the complexity will be sum of all these.eg. We have N algorithms running simultaneously then the total complexity will be $Tc = 1+2+\ldots\ldots+N$. There are various other factors that will play significant roles in the overall complexity. Communication is one of them. Depending upon the distance of locations where servers may be placed due to security or feasibility of the application. Backup is another issue where system needs some time to reconsolidate with the back end links or system. Depending upon the types of backup (hot or cold backup) different timings may be taken up by the system. In case of database application there may be delays due to commit and other updates on various types of storage media, which needs to be taken into account.

References

[1] Asprey, W. et al., “Conquer System Complexity: Build Systems with Billions of Parts,” in CRA Conference on Grand Research Challenges in Computer Science and Engineering, Warrenton, VA (2002), pp. 29-33.

[2] Kephart, J.; Chess, D., “The Vision of Autonomic Computing,” IEEE Computer, Vol. 36, No. 1 (2003), pp. 41-50.

[3] Campbell, A.T.; Coulson, G.; Kounavis, M.E. “Managing Complexity: Middleware Explained”, IEEE Computer Society’s IT Professional Magazine, Vol 1, No 5, September/October 1999

[4] Design and Analysis of Algorithms By R.Panneerselvam Prentice-Hall India. ISBN:978-81203-3278-2.

[5] Nicholas J. Higham, Handbook of writing for the mathematical sciences, SIAM. ISBN 0898714206, p. 25

[6] Donald Knuth. Big Omicron and big Omega and big Theta, ACM SIGACT News, Volume 8, Issue 2, 1976.

[7] Allan Borodin and Ran El-Yaniv (1998). Online Computation and Competitive Analysis. Cambridge University Press, 20,141.

[8] ANSI X9. 17-1985, American National Standard for Financial Institution Key Management (wholesale), Department of the Treasury Directives Manual, Electronic Funds and Securities Transfer Policy, Chapter TD 81, Section 80.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 17: Amortized Analysis, pp.405–430.

[10] Federal Register vol 70, number 96, Announcing Approval of the Withdrawal of Federal Information Processing Standard (FIPS) 46–3, Data Encryption Standard (DES); FIPS 74, Guidelines for Implementing and Using the NBS Data Encryption Standard; and FIPS 81, DES Modes of Operation.

[11] Balenson, D. Automated distribution of cryptographic keys using the financial institution key management standard, Communications Magazine, IEEE Volume 23, Issue 9, Sep 1985 Page(s): 41 – 46.

[12] FIPS PUB 1-2, Code for Information Interchange, Its Representations, Subsets, and Extensions.

[13] FIPS PUB 113 - Computer Data Authentication - the Federal Information Processing Standard publication that defines the Data Authentication Algorithm.

[14] ANSI X9.9-1982, American National Standard for Financial Institution Message Authentication.

[15] Based on lecture by Dr. Ely Porat, Complexity course, Computer Science Department , Bar-Ilan University, January 2008

[16] Complexity Analysis of Routing Algorithms in computer network, M. Bieliková (Ed.), IIT.SRC 2005, April 27, 2005, pp. 69-76.

[17] Torsten Braun,Vijay Arya and Thierry turletti " A backup Tree Algorithm for Multicast Overlay Network" in Springer Berlin/Heidelberg Volume 3462/2005 ISBN 978-3-540-25809-4.

[18] Pratha Sarathi Mandal and Krishnendu Mukhopadyaya "Concurrent checkpoint initiation and recovery algorithms on asynchronous ring networks" in Journal of Parallel and Distributed Computing, Volume 64 , Issue 5 (May 2004)Year of Publication: 2004 ISSN:0743-7315.

[19] Plank, J., Chen, Y., Li, K., Beck, M., Kingsley, G., Memory Exclusion: Optimizing the Performance of Checkpointing Systems, Technical Report UT-CS-96-335, University of Tennessee, August, 1996

[20] Russinovich, M., Segall, Z., Application-Transparent Checkpointing in Mach 3.0/UX, Proceedings of the 28th Annual Hawaii International Conference on System Sciences, 1995

[21] Algorithms by Kenneth A. Berman and Jerome L. Paul Algorithms, 2002 by Course Technology.ISBN-13:978-81-315-0512-2.

[22] A. Ranganathan, "An Task-Execution Framework for Autonomic Ubiquitous Computing" Phd Thesis, University of Illinois at Urbana-Champaign, 2005.

[23] Internetworking Technology Handbook Cisco Publication Chapter 5

Publications

1) Kuldeep Sharma & Deepak Garg ”Complexity Analysis in Heterogeneous System” in International Journal of Canadian Center of Science and Education Vol.2 Page No.48-52 ISSN 1913-8989.