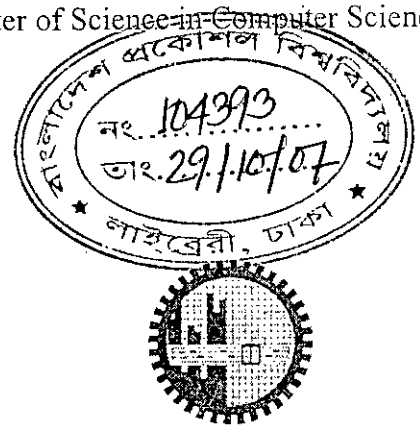
A NEW MODEL FOR RELIABLE WEB SERVICE

BY

MALIHA SULTANA

A thesis submitted to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY

DHAKA, BANGLADESH

SEPTEMBER 2007

CERTIFICATE OF APPROVAL

The thesis titled "A New Model for Reliable Web Service System" submitted by Maliha Sultana, Roll No: 100505055F, Session: October, 2005 to the Department of Computer Science and Engineering of Bangladesh University of Engineering and Technology has been accepted as satisfactory in partial fulfillment of the requirements for the degree of M.Sc. Engg. in Computer Science and Engineering and approved as to its style and contents. The examination has been held on September 24, 2007.

Board of Examiners

Dr. Md. Mostofa Akbar Associate Professor Department of CSE, BUET, Dhaka.	Chairman (Supervisor)
Dr. Muhammad Masroor Ali Professor and Head Department of CSE, BUET, Dhaka.	Member (Ex-officio)
Dr. M. Kaykobad Professor Department of CSE, BUET, Dhaka.	Member
Dr. Reaz Ahmed Assistant Professor Department of CSE, BUET, Dhaka.	Member
Dr. Md. Ahsan Akhter Hasin Professor Department of IPE, BUET, Dhaka	Member (External)

Department of Computer Science and Engineering

Bangladesh University of Engineering and Technology

September, 2007

CANDIDATE'S DECLARATION

It is hereby declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree or diploma.

Maliha Sultana

To the Almighty

To my family

ACKNOWLEDGEMENT

All praise to Allah, the most benevolent and the Almighty, for His boundless grace in successful completion of this thesis.

I would like to express my sincere respect and gratitude to my thesis supervisor, Dr. Md. Mostofa Akbar, Associate Professor of the Department of Computer Science and Engineering (CSE), Bangladesh University of Engineering and Technology (BUET), Dhaka, for his thoughtful suggestions, constant guidance and encouragement throughout the progress of this research work. He was always more confident than I was about my being able to complete this thesis. The ever-interesting part of working with him was: whenever I almost devised a solution, he was there to find a small new patch to the original problem; although the solution part was hardly ever patchable and I had to endure the fun of doing re-research.

My husband, Mushfiqur Rouf, a graduate of the Department of CSE, BUET, Dhaka, helped me a lot on several occasions by providing me an opportunity to learn how to debug critical code segments. He loves to do critical coding, but not to debug his different 'features' that did not quite match up to the original requirements.

I also express my profound thanks to Dr. Chowdhury Mofizur Rahman, Head and Professor, Dept. of CSE and Pro-Vice Chancellor, United International University, Dhaka for granting me study leave and helping me to complete this degree before leaving Bangladesh.

Finally, this thesis would not be possible if my father-in-law, Dr. Md. Abdur Rouf, Professor, Dept of Civil Engg (CE) and Dean, Faculty of CE, BUET, Dhaka, were not there constantly pushing, encouraging and inspiring me.

I would like to extend my sincere thanks to my parents, parents-in-laws and my husband whose continuous inspiration, sacrifice and support encouraged me to complete the thesis successfully.

ABSTRACT

In this thesis a new model for a distributed web service system is presented. The proposed system is composed of multiple web service components having multiple alternative versions distributed among multiple servers. Each of the versions of a component has its own multidimensional resource requirements. A request is placed by a client for a specific web service component to a broker, which can allocate more than one version to increase satisfaction of the client by redundant allocation. The satisfaction of a client depends on the reliable service of redundant versions served by multiple servers. In the proposed model an allocation is to be found that maximizes total client satisfaction subject to the resource constraints of the servers.

A variant of the network flow maximization algorithm has been used to address the proposed problem. The resource requirements and constraints can be mapped onto a multidimensional flow model. Each augmenting path represents one allocation of a version of a web service component at a server. Clearly, there is no scope for partial flows and the flow conservation principle must not be broken. Considering these two algorithmic constraints, the basic Push-Relabel algorithm is modified to solve the problem. The resource allocation problem considered here is a Multidimensional Knapsack Problem which is NP hard and has exponential time complexity. In the proposed algorithm this Multidimensional Knapsack Problem has been reduced to a Single Knapsack Problem at each server with necessary communications among the servers. This communication among the servers is achieved through the flow of resources in the network. Thus the presented heuristic running with polynomial time complexity and producing suboptimal solution is attractive for a web service system, a real time application.

The proposed heuristic algorithm has been compared with a Brute Force and a Greedy algorithm. It is found that the solutions of the proposed algorithm are very close to the optimal solutions provided by the Brute Force algorithm. On the other hand, the proposed algorithm provides far better solutions than the Greedy algorithm. Since the running components can not be allocated to another server in the middle of their execution, the new component requests must be allocated using the remaining free resources. That means each

iteration is an independent problem to be solved using this heuristic algorithm. In addition to that, the network flow model presented in this thesis inherently rules out the number of requests from contributing in time complexity of the algorithm.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

COTS: Commercial Off-The-Shelf

GA: Genetic Algorithm

NVP: N-Version Programming

QoS : Quality of Service

RBS: Recovery Block Scheme

RPC: Remote Procedure Call

SA: Simulated Annealing

SOAP: Simple Object Access Protocol

SORM: Service-Oriented Software Reliability Model

TS: Tabu Search

UDDI: Universal Description, Discovery and Integration

WS: Web Service

WSDL: Web Service Description Language

XML: Extensible Markup Language

List of Symbols

a_{kr}: Amount of Resource Type r available at the Server k.

C_j^i: Version j of Component i

$c(u, v)$: Capacity of the edge (u, v)

$c_b(u, v)$: Basic capacity of the edge (u, v)

$c_f(u, v)$: Residual capacity of the edge (u, v)

M: Total number of versions of all web service components

m_i : Number of alternative versions of Component i

n : Number of requests

n_i : Number of requests for Component i

N: Total number of web service components

S: Total number of Servers

w_{jr}^i: Amount of resource type r required *by* C_j^i

Chapter 1. Introduction

1.1. Motivation

With the advances of the internet, web services are becoming more and more popular. A web service performs a set of tasks and can be accessed via the network. When a request comes for a service, the specific web service component is executed in the server and the result is passed to the client. Because of the constraints of the servers (for example, limited hardware resources) it is not always possible to provide service to each incoming request. The following issues regarding web service systems motivate the investigation of new models:

- Redundant allocation of a component is a way to provide clients with uninterrupted reliable service.
- Distribution of the web service components to multiple servers is necessary to serve more clients as the resources of a server are limited.
- Allocation of web service components to clients from appropriate servers is an important function to be done by some entity.

The following figure depicts a typical distributed web service system.

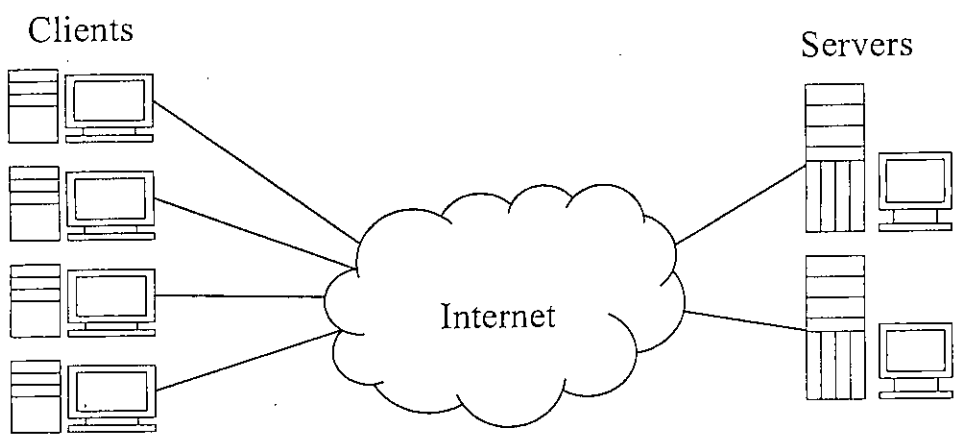


Figure 1-1 The client server model of a distributed web service system.

1.2. Problem Definition

In this thesis a new model is presented that addresses the problem of allocating requested web service components to the clients in a distributed web service system. The following points describe the problem:

- *Multiple versions of a component*: Each web service component can have multiple alternative versions. The versions of a component are developed by separate group of programmers from the same initial specification. Thus they provide similar functionality but different satisfaction values. Here the satisfaction provided by a web service component can be considered as a function of some non functional criteria such as reliability, performance, cost, security etc of that component.
- *Components residing in multiple servers*: The versions of a web service component are hosted by multiple servers.
- *Requests of the clients:* The clients request for web services which are executed in the server using server's resources and the results are passed to the clients over the network. A server can reject to serve a client if it can not satisfy the resource requirement needed to serve that client.
- *Allocation of redundant versions*: Multiple alternative versions of a component can be allocated to a client to provide more reliable service which will increase the client's satisfaction.
- *Maximization of satisfaction*: The allocation of services from multiple servers is done in such a way so as to maximize the total satisfaction of all the clients by providing reliable services allowing redundancy whenever possible.

1.3. Objective and Scope of the Thesis

The objective of this thesis is to design an optimization model for the decision making process of a distributed web service system. The optimization model should maximize the total satisfaction of all clients while respecting the resource constraints of the servers. Also the optimization model should find a solution quickly since the decision

making process is executed at run time for each set of requests, keeping the clients waiting for an answer.

The optimization model presented in this thesis is solved by a heuristic using network flow maximization algorithm. The heuristic will not always provide an optimal solution in all scenarios. However, the solution whether it is optimal or not, will be generated in polynomial time. The time complexity of the algorithm does not even depend on the number of requests.

The algorithm for an exact solution is out of scope of the thesis. The problem is NP hard and it is practically impossible to solve the problem for larger data sets. That is why the proposed algorithm is compared with a simple greedy heuristic. Also to determine the performance of the proposed algorithm, the proposed algorithm has been compared with a Brute Force algorithm for smaller data sets. A detailed study of the optimality of the algorithm is out of scope of the thesis.

In the proposed model, the total service satisfaction of all clients is maximized. Here the satisfaction of a client is considered as a function of reliability of the service but there is no straight forward relationship between reliability and satisfaction. This relationship is also out of scope of this research.

A worst case complexity analysis has been presented in the thesis. Finding the complexity in the average case requires statistical analysis of the client's requests, which is not studied in this phase of research.

1.4. Outline

The remaining part of the thesis is organized as follows.

Chapter 2 discusses some existing reliability optimization models and algorithms. An overview of the network flow problem, its applications and algorithms are also presented in this chapter.

The proposed new model is described in Chapter 3, along with its mathematical formulation. Section 3.2 is devoted to the description of the proposed network flow heuristic for solving the model. A worst case complexity analysis of the algorithm will be found in Section 3.7.1.

In Chapter 4, a comparative result analysis of the algorithm with a Brute Force and a Greedy algorithm is presented.

Chapter 5 concludes the thesis and provides some directions for further research in this field.

Chapter 2. Literature Review

Reliability optimization techniques have been used heavily in the literature for component based software systems. Reliability optimization in this context means finding an optimal set of components from alternatives for a software system so as to maximize reliability of the whole system while maintaining some cost or resource constraints. The model proposed in this thesis maximizes total satisfaction (which is considered here as a function of reliability) while respecting multidimensional resource constraints and is thus similar to these models. But the new part of the proposed model is that it is not for "developing a single software system", rather it deals with web services and finds an optimal set of redundant versions of each web service component to satisfy each request over the network.

The chapter starts with some basic definitions needed to understand the proposed model. Then some reliability optimization models from the literature are described and finally some network flow algorithms and their applications are presented. The Push-Relabel network flow algorithm has been described in detail since a variation of this method is used to solve the model presented in this thesis.

2.1. Some Basic Definitions

Next few sections provide the definitions of some basic terms used to describe the model presented in this thesis.

2.1.1. Web Service

A web service is an interface that describes a collection of operations that are network accessible to remote users [1], [2]. Web services provide a standardized method of communication among software applications and make software application resources available over the network in a standardized way. Using web service technology, one application can call on another to perform functions which can be anything from simple processes to complicated business tasks, even if the applications are running

on different machines with different operating systems and are written in different languages. In other words a web service makes its resources available in such a way that any client application, regardless of its internal implementation can use it.

To be accessed easily, web services must adhere to a set of standards. Some of the most frequently used standards are described in [1] and are presented below:

WSDL: WSDL is the Web Service Description Language. A service provider formally describes its services through a WSDL file. It is an interface standard that abstracts from any platform and programming language specific details of how application code is actually invoked. WSDLs are generally publicly available and provide the details that a client needs to interact with the service. For example, if a web service translates English sentences into French, the WSDL file will explain how the English sentences should be sent to the web service, and how the French translation will be returned to the requesting client.

UDDI: The UDDI (Universal Description, Discovery and Integration) registry serves as a means of discovering web services described using WSDL. The service vendors register their services into an UDDI directory. The service consumers then search the UDDI directory to find an appropriate service from a set of alternatives.

XML and SOAP: XML (Extensible Markup Language) messages provide the common language by which different applications can interact. To operate a web service a client sends an XML message containing a request for the web service to perform some operation; in response to the request, the web service sends back another XML message containing the results of the operation. Typically these XML messages are sent using SOAP, an acronym for Simple Object Access Protocol that specifies a standard format for applications to call each other's methods and pass data to one another.

2.1.2. Reliability

Reliability refers to the property that a system can run continuously without failure [3]. Reliability is defined in terms of a time interval instead of an instant of time. The

formal definition of software reliability is "the probability that the software will be functioning without failures under a given environmental condition during a specified period of time" [4], [5]. Other approaches measure software reliability in terms of percentage of failures for a given number of attempts [6]. Software reliability is one of the important measures that guide a client while choosing a software system from a set of alternative options.

Zo et. al. [6] consider web services as Remote Procedure Call (RPC) methods over the internet. Considering this characteristic the approach that measures reliability as a percentage of failures other than a probability that a failure will occur during a specified period of time is adopted for the case of web services. For example a web service that exhibits 99.1% reliability indicates that the service will perform successfully (that is without failure) 991 times out of a 1000 attempts. Prior research in Quality of Service (QoS) area has identified a set of criteria that plays significant roles in web service selection process. The results of a survey shows the relative importance of web service selection criteria and it has been revealed that reliability and security plays the most important roles and are almost twice as much as important than cost and performance criteria [7].

2.1.3. Redundancy

Redundancy is being used as a common technique for increasing the reliability of a software system. The technique of redundancy can be applied when the software system is decomposed into several modules or components each performing a separate function and having its own reliability measures. For component based software systems, redundancy means the employment of functionally equivalent alternative components to improve reliability of software operations. The inherent idea here is that if a component fails, its alternative version can take its place instantly and the execution of the total software system is not interrupted. Recently two distinct approaches have been investigated which employ alternate software components to achieve software fault tolerance. The approaches are: *Recovery Block Scheme (RBS)*: The recovery blocks are organized in a manner similar to the dynamic redundancy (standby sparing) technique in hardware. The primary

objective is to perform run time software error detection and to implement error recovery by taking an alternative path of operation that is, by activating an alternative stand-by module [8], [9], [10].

N-Version Programming (NVP): It is analogous to the static (replication and voting) redundancy approach in hardware. NVP is defined as the independent generation of $N \geq 2$ functionally equivalent programs, called "*versions*" from the same initial specification [8]. The "independency" in the generation of the programs is achieved by developing the *N*-versions by *N* groups of software developers and using different algorithms and/or programming languages and/or platforms whenever possible. The inherent idea here is to execute all the versions in parallel and to employ a decision maker to combine the results of the independent versions by majority voting technique.

Since the redundant components require additional resources, additional costs in terms of programming effort, hardware requirements and time, the redundancy level to achieve fault tolerance in a software system must be carefully determined and if possible, optimized.

2.2. Reliability Optimization Models

Reliability optimization models deal with the selection of an optimal set of components for a software system that maximizes the reliability of the software system as a whole. In these optimization models it is assumed that individual reliability of each component is available, either from vendors or from published third party sources. In many cases reliability varies inversely with cost and resource consumption of software components. Since developing a highly reliable component requires exhaustive design and testing effort, the cost of the component increases automatically. Thus the reliability optimization models offer a trade off between reliability and budget and other resource constraints. Two types of reliability optimization problems arise in the literature: the first one attempts to minimize the amount of resource consumption while maintaining the reliability of the system at a given level, while the second one maximizes the overall system reliability considering

the cost or resource constraints of the system. The model proposed in this thesis is a generalization of the second type.

2.2.1. Cost-Reliability Models

The models described in this section attempt to maximize reliability while meeting cost or budgetary constraints. Berman et. al. [11] present four different optimization models for maximizing reliability of modular software systems while ensuring that the expenditures remain within budget. Models 1 and 2 select an optimal set of modules for one function system with and without redundancy respectively. Models 3 and 4 do the same for a system with K functions. The authors presented a Dynamic Programming algorithm to solve the models. Another four models for optimizing software and hardware reliability for fault tolerant distributed systems have been presented in [12]. The models find the optimal system structure while considering reliability and cost of available software and hardware components. A simulated annealing approach is presented to solve this optimization problem. The optimization models presented by Jung et. al. [13] optimizes quality of software while maintaining a budgetary constraint by selecting the best Commercial Off-The-Shelf (COTS) products among alternatives for each module of the software system.

2.2.2. Resource-Reliability Models

Belli et. al. [9], [14] present two different models for reliability optimization via redundancy allocation; one using RBS method and one using NVP method. They also present a model that uses a combination of both RBS and NVP methods. The authors assumed that the software system is made up by independent modules connected in series. The problem is reduced to finding the optimal redundancy level for each module within the system while respecting the overall resource consumption. No solution to the problems has been offered. Caserta et. al. propose a similar optimization model in [15], [16] and offer a Tabu Search based metaheuristic algorithm to solve the model.

2.2.3. Reliability Models for Web Services

Zo et. al. [6] present a model for the selection of a set of appropriate web services to support the tasks needed for the development of an application composed of web services. The assumption here is that multiple web services are available for each task and a single web service can support multiple tasks. Assuming a set of m tasks, with n web services available for each task, the total number of viable combinations is given by n^m, which becomes combinatorially explosive and infeasible for an exhaustive search. The authors present a Genetic Algorithm formulation for the above model.

Chang et. al. [17] present the formulation of an Evolutionary Algorithm to solve a similar model that optimizes the composition of web service components based on good quality and performance service components over different service providers.

Tsai et. al. [18] propose a Service-Oriented Software Reliability Model (SORM) that evaluates the reliability of web services (WS) in two steps: (1) the reliabilities of atomic web services are computed using a group testing technique, where an atomic service represents a service that does not call other WS and is thus treated as a unit that is not to be broken. (2) The reliability of a composite service is evaluated based on the reliabilities of the component services (they can be either atomic or composite services) and the structure (relationships) among the component services.

2.3. Reliability Optimization Algorithms

It is normally difficult to develop exact methods for reliability optimization problems because such methods involve a large amount of computational effort and normally require larger computer memory. For these reasons more emphasis has been given on heuristic and metaheuristic approaches for solving various reliability optimization problems. An overview of such algorithms will be found in [19].

Kim and Yum [20] present a heuristic algorithm for solving redundancy optimization problems in complex systems. They allow *excursions* (examination of a series of infeasible solutions generated by some rules) from a current solution in a bounded

infeasible region thus eliminating the chance of being trapped in a local optimum. At each iteration the excursions eventually return to the feasible region with a possibly improved solution than the starting one. This improved solution now becomes the current solution and the algorithm proceeds in the same way. The problem of this heuristic method is that it requires a large number of iterations and is thus very slow.

In recent years, metaheuristics such as *Genetic Algorithm* (*GA*) [6], [21], [22], *Simulated Annealing* (*SA*) [12] and *Tabu Search* (*TS*) [15], [16], [23] are being used in solving reliability optimization problems with redundancy allocation. A Genetic Algorithm (GA) is a heuristic search method that finds exact or approximate solutions to optimization problems by implementing the techniques of evolutionary biology, such as inheritance, mutation, selection and crossover. Usually the solution is encoded as a binary string, called the population. In each iteration, the fitness of every individual in the population is evaluated using a problem specific fitness function; a set of individuals is selected from the current population (based on their fitness) and modified by the application of evolutionary techniques to form a new population. Unlike GA, where the objective value is improved continuously, Simulated Annealing (SA) involves probabilistic transitions among the solutions of the problem. A superior solution is always accepted; an inferior solution is accepted probabilistically based on the difference in quality and a temperature parameter. Thus SA can encounter some adverse changes in the objective value in the course of its progress. Such changes are intended to lead to a global optimal solution other than the local one. Another metaheuristic that avoids local optima is the Tabu Search (TS), originally proposed by Fred Glover [24]. Tabu Search can simply be seen as a local search method with a short time memory structure, called the *Tabu List* used to prevent cycling when moving away from the local optima through non-improving moves.

The GA, SA and TS – all are heuristic algorithms, meaning that performance of these search methods can not be guaranteed. Since the search space for the reliability optimization problems with redundancy allocation often become combinatorially explosive, exact methods are difficult to find and so metaheuristics can be a good choice for this type of problems. However for a large solution space, the metaheuristic

search methods often take a large number of iterations before convergence thus making themselves unsuitable for systems where real time decision making is needed.

2.4. Introduction to the Network Flow Maximization Problem

A network flow maximization problem aims at the maximization of "*flow*" through a "*network*". Consider a network of pipes where valves allow flow in only one direction. Each pipe has a capacity per unit time. Now this network can be modeled with a set of vertices for the junctions and a set of edges for the pipes weighted by the pipe capacity. Given two junctions, as the source (s) and the sink (t), the network flow maximization problem finds the maximum amount of flow from s to t. The flow network can be used to model many real life scenarios, for example, roads with traffic capacities, parts through assembly lines, information through communication networks and so forth.

2.4.1. Flow Network

As described in [25], [26] a *flow network* $G = (V, E)$ is a directed graph in which each edge $(u,v) \in E$ has a non negative capacity $c(u,v) \geq 0$. If $(u, v) \notin E$, $c(u, v) = 0$. Two vertices are distinguished in a flow network: a source s and a sink t. A flow in G is a real valued function $f: V \times V \rightarrow R$, that satisfies the following three properties:

Capacity Constraint: For all $u, v \in V$, $f(u, v) \leq c(u, v)$.
Skew Symmetry: For all $u, v \in V$, $f(u, v) = -f(v, u)$.
Flow Conservation: For all $u \in V-\{s, t\}$,

$$\sum_{v \in V} f(u, v) = 0$$

Here, $f(u, v)$ is called the *flow* from vertex u to vertex v. It can be positive, zero or negative.

The value of a flow f is defined as,

$$|f| = \sum_{v \in V} f(s, v)$$

That is, the total flow out of the source. In the maximum-flow problem, a flow network G is given with a source s and a sink t, and a flow of maximum value is to be found. Figure 2-1 shows a simple flow network along with the maximum flow through the network. The first number above each edge denotes the capacity of that edge and the second number denotes the amount of flow through that edge. Maximum amount flow that can be passed trough this network is 10.

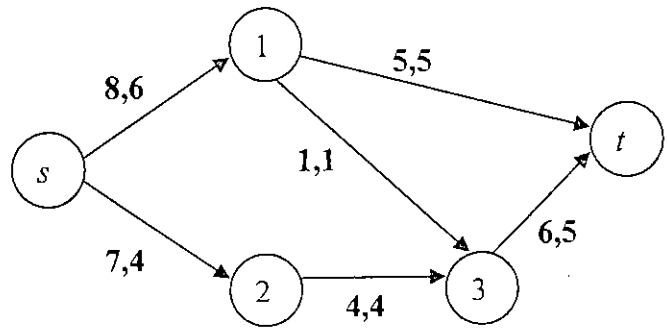


Figure 2-1 A simple example of a flow network with source *s* and sink *t*.

2.4.2. Networks with Multiple Sources and Sinks

A maximum flow problem may have several sources and sinks, rather than just one of each. Let us consider a network with a set of m sources $\{s_1, s_2, \ldots, s_m\}$ and a set of n sinks $\{t_1, t_2, \ldots, t_n\}$. This network can be reduced to an ordinary network by adding a *supersource* s and a *supersink* t along with the following edges:

- Directed edge (s, s_i) with capacity $c(s, s_i) = \infty$ for each $i = 1, 2, \ldots, m$.
- Directed edge (t_i, t) with capacity $c(t_i, t) = \infty$ for each $i = 1, 2, \ldots, n$.

Any flow in the new network corresponds to a flow in the original network and vice versa. The supersource s simply provides as much flow as desired for the multiple

sources s_i, and the supersink t consumes as much flow as desired for the multiple sinks t_i.

2.4.3. Residual Networks

Given a network and a flow, the residual network G_f consists of edges through which more flow can be sent. If f is a flow in G, then the *residual capacity* of an edge (u, v), where $u, v \in V$ is,

$$c_f(u, v) = c(u, v) - f(u, v).$$

When the flow $f(u, v)$ is negative, the residual capacity $c_f(u, v)$ is larger than the original capacity $c(u, v)$. This is because negative flow implies a flow in the opposite direction which can be cancelled by sending a flow of the same amount in the original direction.

2.4.4. Network Flow Algorithms

The first pseudo polynomial algorithm for the maximum flow problem is the augmenting path algorithm of Ford and Fulkerson [25], [26], [27]. Since then several more-efficient algorithms have been developed. Edmonds and Karp [28] obtained a polynomial version of the original augmenting path algorithm. The push relabel methods originally developed by Goldberg [29] [30], [31] solve the network flow problem from a different perspective other than the augmenting path methods. An overview of these network flow algorithms can be found in [29]. Some of the most frequently used network flow algorithms are described in the next few sections.

2.4.4.1. Ford-Fulkerson Method

The Ford-Fulkerson method is iterative. At each iteration, an augmenting path is found which is simply a path from the source s to the sink t through which more flow can be sent, that is all the edges on the path have positive residual capacities. Then the maximum possible flow is augmented along this path. This process is repeated until

no augmenting path can be found. Upon termination this process yields a maximum flow.

The Ford-Fulkerson method is as follows:

```
PROCEDURE Ford-Fulkerson(G,s,t)
    Initialize flow f to 0
    WHILE there exists an augmenting path p
        Augment flow f along path p
    END WHILE
    RETURN f
END PROCEDURE
```

The time to find an augmenting path is $O(E)$ if either Depth-First Search (DFS) or Breadth-First-Search (BFS) is used. The WHILE loop is executed at most $|f^*|$ times, where f^* is the maximum flow found by the algorithm. Each execution of the WHILE loop takes $O(E)$ time. Thus the total running time of the Ford-Fulkerson algorithm is $O(E|f^*|)$.

2.4.4.2. Edmonds-Karp Method

This method implements a Breadth-First-Search for computing the augmenting path in the Ford-Fulkerson method. That is, the augmenting path here is the shortest path from s to t in the residual network, where each edge has unit distance. Now the while loop of the Ford-Fulkerson method is executed at most $O(VE)$ time. With $O(E)$ time for each iteration, the total running time of the algorithm is $O(VE^2)$.

2.4.4.3. Push-Relabel Method- Goldberg's Algorithm

The original Push-Relabel algorithm is due to Goldberg et.al. [30] and runs in $O(V^2E)$ time, thus improving over the $O(VE^2)$ time of Edmonds-Karp algorithm. A modified version of the Push-Relabel algorithm is used to solve the optimization

problem proposed in this thesis. That's why the Push-Relabel algorithm will be described in more details than the other network flow algorithms.

The Push-Relabel algorithm does not compute an augmenting path at each iteration. Rather it examines each vertex to find whether a flow can be sent from that vertex to any of its neighboring vertices. The flow conservation property is not also maintained through out the execution. A *preflow*, which is a function f: $V \times V \rightarrow R$, is maintained at all time, that satisfies the capacity constraint, skew symmetry and the following relaxation of the flow conservation:

$$f(V, u) \geq 0, \text{ for all } u \in V - \{s\}$$

The total net flow at a vertex u is called the *excess flow* into u. A vertex is called overflowing if it has an excess flow greater than zero. Each vertex is associated with a height, where, height of source, $h[s] = |V|$ and height of sink, $h[t] = 0$. For all other vertices the height is initially zero, and gradually increases as flow is pushed from a vertex. Flow can be pushed downhill only, meaning that a lower vertex can not push its excess to a higher vertex.

Basic Operations:

As the name implies, the Push-Relabel algorithm has two basic operations:

Push: This operation is applied to an overflowing vertex u if for any other vertex v, $c_f(u, v) > 0$, and $h[u] = h[v] + 1$.

Flow of amount Minimum $(e[u], c_f(u, v))$ is pushed from u to v. After each push the flow and excess at each vertex are updated correspondingly. No residual edges exist between two vertices whose heights differ by more than 1, so, excess flow is pushed downhill only by a height differential of 1.

A push operation is further distinguished as a saturating push and a non-saturating push. If $c_f(u, v)$ becomes zero after a push operation, then it is a saturating push, otherwise it is a non saturating push.

Relabel: This operation is applied to an overflowing vertex u if $h[u] \leq h[v]$ for all $v \in V$ where $(u, v) \in E_f$. Relabeling is done when a vertex has excess flow but all of its adjacent vertices are higher than it. Since flow can only be pushed downhill, the height of the overflowing vertex must be increased to get rid of the excess flow.

The height of such an overflowing vertex u is increased to,

$$h[u] = 1 + \min\{h[v] : (u, v) \in E_f\}$$

The generic Push-Relabel algorithm is as follows:

```
PROCEDURE Generic_Push_Relabel(G, s, t)
    Initialize height and excess flow of each vertex to 0
    Initialize a zero flow through each edge
    h[s] = |V|
    FOR each vertex u ∈ Adj[s]
        Push a flow of value c(s, u) from s to u
        Update excess flow at u as, e[u] = c(s, u)
    END FOR
    WHILE a push or relabel operation is possible
        Select an operation and perform it
    END WHILE
END PROCEDURE
```

The excess flow of each vertex that can not be pushed to sink eventually comes back to the source. When the algorithm terminates, no vertex has excess flow, thus the preflow becomes a flow that maintains the flow conservation property.

The complexity of the push relabel algorithm depends on the number of relabel, saturating push and non-saturating push operations. Each of the three types of operations has separate bounds. Since the maximum height of any vertex can be $2|V| - 1$, total number of relabel operations is $O(V^2)$ where each relabel operation takes $O(V)$ time. Total number of saturating pushes is $O(VE)$ and total number of non-saturating pushes is $O(V^3 + V^2 E) = O(V^2 E)$. Each push operation takes $O(1)$

time. Thus the total complexity of the Push-Relabel algorithm is $O(V^2E)$ and is dominated by the number of non-saturating pushes.

2.4.5. Applications of Network Flow

Some combinatorial problems can easily be seen as maximum flow problems. The maximum Bipartite Matching Problem [25] can easily be solved by the Ford-Fulkerson algorithm on a graph $G = (V, E)$ in $O(VE)$ time. The problem is defined as follows:

Given an undirected graph $G = (V, E)$, a matching M is a subset of edges E, such that for all vertices v belongs to V, at most one edge of M is incident on v. A vertex v is said to be matched by a matching M, if some edge in M is incident on v; otherwise v is unmatched. A maximum matching is a matching with maximum cardinality. The problem can be reduced to a network flow problem by creating a supersource and a supersink along with their corresponding edges and assigning unit capacity to each edge.

Other applications of network flow maximization problem include Baseball Elimination Problem, Matrix Rounding and Matching Supplies and Demands [25], [26].

2.5. Chapter Summary

Some reliability optimization models for component based software systems have been described in this chapter. Most of these models have been solved using metaheuristics like GA, SA or TS. A brief introduction to the network flow maximization problem and a few methods for solving the problem have also been presented here. A modified version of Goldberg's Push-Relabel network flow algorithm described in Section 2.4.4.3 has been used to optimize the model proposed in this thesis. The mapping of the proposed model to a network flow graph along with the modifications to the original algorithm has been described in the next chapter.

Chapter 3. Proposed New Model

This chapter describes the proposed new web service system. A mathematical model of the proposed system has also been presented here. The optimization technique employed to solve the model is a heuristic algorithm that maps the model to a network flow maximization problem. But any network flow algorithm can not be used as the proposed model is significantly different from a typical network flow model. In this thesis a modified version of Goldberg's Push-Relabel method has been used to solve the proposed model. The modifications to the original algorithm have been presented in this chapter. The chapter concludes by providing a worst case complexity analysis of the proposed network flow heuristic algorithm.

3.1. The Distributed Web Service System

The proposed model is for a web service system where a set of web service components are distributed among multiple servers and are accessible to the clients through a broker that takes the decision of admission or rejection of a client into the system. The decision process is based on the maximization of total satisfaction of all clients. The broker notifies each server after a decision has been reached and the servers then execute the requested components and sends the results to appropriate clients. Figure 3-1 depicts the architecture of the Distributed Web Service System.

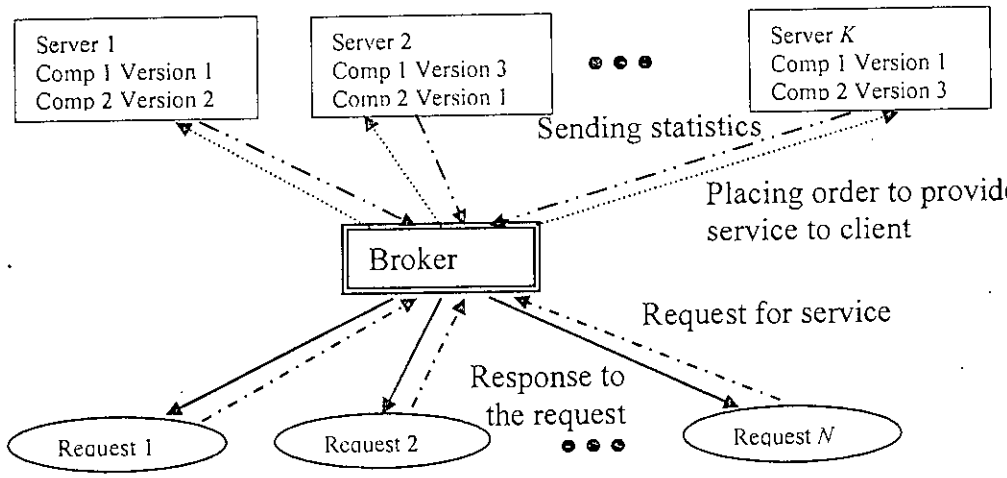


Figure 3-1 A typical architecture of the distributed web service system.

3.1.1. Working Principle of the Proposed Distributed Web Service System

The following points demonstrate the working principle of the proposed distributed web service system:

- Each web service component is capable of performing a separate task requiring some input parameters and producing some outputs. In response to the request of a client, a component is executed in the server using server's resources and the outputs are passed to the client.
- To perform a specific task, a client requests a component to the broker. Considering the available resources of the servers the broker takes the decision of acceptance or rejection of a client.
- Each component has multiple alternative versions, each performing the same task, but providing separate reliability and having different multidimensional resource requirements. The difference in reliability and resource requirements arises from the use of N-Version Programming which states that the versions should be developed using separate platform and/or technology by separate developers. While a single version of a component suffices, providing more than one version of the same web service component to a client increases the reliability of the service by means of redundant allocation. This difference in reliability brings the variation of client satisfaction definitely.
- The versions of the components are distributed among multiple servers. Each server contains a set of versions for a set of components. The sets of versions of a component stored in different servers are not disjoint, that is, more than one server can contain the same version of a component. Allocating the same version of a component multiple times from multiple servers does not improve total satisfaction of a client, since these versions will fail identically.
- The servers have multidimensional resources. Possible resource types are memory, CPU cycles, I/O bandwidth etc. Each server notifies the broker about the amount of available resource of each type and this information varies from server to server.

- In response to the request for a component of a client, the broker either rejects it or allocates a set of versions from different servers to the client. The set of versions are allocated to clients in such a way that maximizes total satisfaction of all clients while maintaining the resource constraint of the servers.
- When a decision has been made for a set of requests, the broker delegates the rest of the work to the servers, i.e., it redirects the requests to the assigned servers and starts the decision making process for a new set of requests. Each server processes the requests sent to it, and sends the results to the clients over the network.

3.1.2. Assumptions of the Model

The following points are assumed to simplify the model in the context of real or practical situation.

- Server reliability is assumed to be very high for each server and is not incorporated in the model.
- Multidimensional resource requirement of each version of each component is known to the broker. Also, the set of versions of a component hosted by a server is known to the broker.
- The versions of a component are independent of each other. This independency can be achieved by developing the versions separately from the same specification.
- Clients do not want to know which version is allocated from which server. Only the allocated set of versions for a component and the total satisfaction is of concern to them.
- Though a particular version of the requested component can be served to a particular request multiple times from multiple servers, but this sort of redundancy will not increase total satisfaction of that client.
- Satisfaction of a service can be a function of reliability, cost, performance and security. In the distributed web service system satisfaction of the client for a particular service will increase if more than one component is allocated to the client. The satisfaction value might be the reliability of that service, if the

client thinks so. In this model it is assumed that satisfaction of the service is a user-defined input parameter and it must be different for different set of versions used for a particular service.

- Providing reliable web services to the clients requires the maximization of satisfaction of the clients.

3.1.3. Mathematical Formulation

For the formulation of the model for reliable distributed web service system the following parameters are defined:

m_i = Number of versions of Component i

L = Maximum number of versions per component = $\max_i(m_i)$

n = Number of Requests

S = Number of Servers

R = Dimension of Resources

N = Number of Components

C_j^i = Version j of Component i

w_{jr}^i = Amount of Resource Type r required *by* C_j^i

a_{kr} = Amount of Resource Type r available at the Server k.

A_{qjk}^i = 1, if C_j^i is allocated for request q at Server k

= 0, otherwise.

x_{qj}^i = 1, if C_j^i is allocated for Request q

= 0, otherwise.

$= \sum_k A_{qjk}^i$

y_q^i = 1, if Component i is required by Request q

= 0, otherwise.

$u^i_{j_1,j_2,\ldots,j_L}$ indicates the satisfaction of a combination of versions for Component i. Each of the subscripts indicates the inclusion of a version. $j_l = \{0,1\}$, implying the absence or presence of the l-th version in the combination of the versions of Component i.

$u^i_{j_1,j_2,\ldots,j_L} = 0$, where $\sum_{l=m_i+1}^{L} j_l > 0$ as Component i has only m_i components.

Here,

$i = 1, 2, \ldots, N$

$j = 1, 2, \ldots, m_i$

$k = 1, 2, \ldots, S$

$q = 1, 2, \ldots, n$

$r = 1, 2, \ldots, R$

The objective is to maximize total satisfaction of all clients served, which is computed as the sum of the satisfaction values of the requested components. Thus, the objective function is:

Maximize, $\sum_{q=1}^{n}\sum_{i=1}^{N} y^i_q u_{x^i_{q1},x^i_{q2},\ldots,x^i_{qL}}$ subject to the following constraints:

Constraint 1: $\forall r \forall k \left[\sum_{q}\sum_{j} w^i_{jr} A^i_{qjk} \le a_{kr} \right]$, indicating the limitations of resources of different types in the servers.

Constraint 2: $\sum_{i=1}^{N} y^i_q = 1$, indicating that a client can request for at most one component.

3.2. Solving the Model by Mapping to the Network Flow Maximization Problem

A heuristic using the network flow maximization algorithm has been presented to solve the proposed model of resource allocation problem in a distributed web service system. If there were no limitation of resources of the servers then all the requests could have been served with the highest possible satisfaction, which is not a possible case in reality. Thus the assigned web service components must not overuse the amount of available resources at the servers. That is, the consumption of resources must not exceed the server capacity, which forms the basic idea of using a network flow model in the proposed scenario. But the typical flow maximization approach will not be suitable in this case, since flow in this model means a resource flow and maximization of resource consumption from each server does not always guarantee maximization of total satisfaction. Thus the original network flow approach is modified so as to allow resources to flow through the servers, which are represented as vertices of the network flow graph in as much quantity as possible. But when the server capacity is not sufficient the resources are allowed to make room for them by pushing back other flows from the servers, thus trying different combinations of allocations. Whenever an allocation is modified in which total satisfaction may increase, a snapshot of the current allocation is taken and is saved if it is better than the last best allocation.

3.2.1. Mapping of the Proposed Model to Network Flow Graph

The maximization problem defined in the model can be mapped to a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges as shown in Figure 3.2. A brief description of the mapping is presented below:

- The versions of the web service components (C_j^i) and the servers (Srv_k) form the vertex set V.

- An edge is created from the version node to the server node if the version is hosted by that server.
- All the resource requirements of the versions (w^i_{jr}) and the resource constraints of the servers (a_{kr}) are mapped to the capacities of the edges.
- Thus a flow through the network indicates a flow of resources constrained by the server capacities.

An example of a network flow graph for the model is presented in Figure 3-2 and the detailed description of the mapping is given in the next few sections.

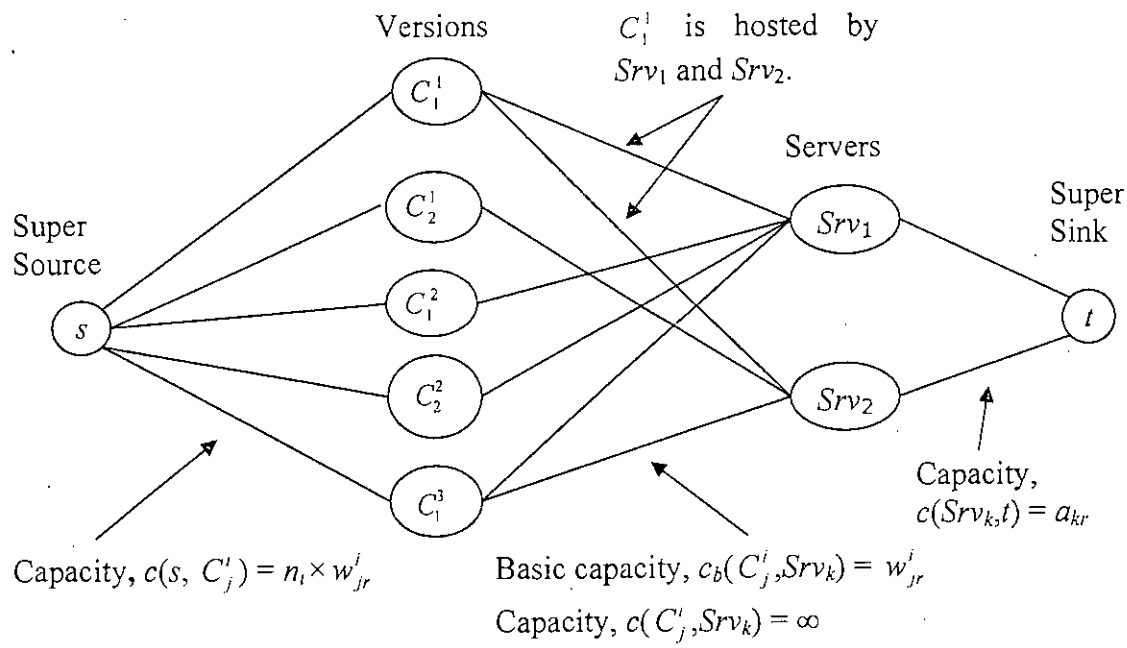


Figure 3-2 An example of a network flow mapping of the proposed model.

3.2.2. The Set of Vertices in the Mapped Network

The set of vertices of the network flow graph do not include the set of requests. Number of requests will play a role when declaring the capacities of the (source, version) edges. Thus the flow network size ($|V|$) and hence the complexity of the algorithm does not depend on the number of requests.

The vertices are divided into two sets:

The version set: Each vertex of this set represents a version of a component. Since component i has m_i versions, total number of vertices in this set is $M = \sum_{i=1}^{N} m_i$. Each version is denoted as C_j^i meaning Version j of Component i for $i = 1, 2, \ldots, N$ and $j = 1,2, \ldots, m_i$.

The server set: A vertex is created for each server. So, total number of vertices in this set is S. Each server is denoted as Srv_i, $i = 1, 2, \ldots, S$.

Here the version set represents the set of sources and the server set represents the set of sinks. To convert this multiple-source multiple-sink network to a single-source single-sink network, a supersource and a supersink must be added.

3.2.3. The Set of Edges in the Mapped Network

There are three sets of edges, among which two sets of edges are created along with the supersource and supersink. The remaining set of edges represents the availability of the versions in the servers. If version C_j^i is available at server Srv_k then there will be an edge from vertex C_j^i to vertex Srv_k. If a flow through this edge can be passed to sink, then it is guaranteed that the amount of resource required by the version C_j^i is available at server Srv_k. Since there is no meaning of pushing flows from one version to another or from one server to another, there are no such edges in the network flow graph. Thus the graph represented by the versions and the servers is bipartite.

3.2.4. Capacities of the Edges in the Mapped Network

Since a flow represents a resource flow, the capacity of each edge represents resources and is multidimensional. Let $c(u, v)$ denotes the capacity of the edge (u, v). Each capacity $c(u, v)$ represents the resource values in all dimensions. Since resource dimension is R in the proposed model, each $c(u, v)$ represents an R-tuple.

The capacities of the edges are defined as follows:

$c(s, C_j^i) = n_i \times$ (resource requirement of C_j^i)

$c(C_j^i, Srv_k) = \infty$

Basic capacity $c_b(C_j^i, Srv_k)$ = resource requirement of C_j^i

$c(Srv_k,t)$ = amount of resource available at server k

Here,

i = component index, $i = 1, 2, \ldots, N$

j = version index, $j = 1, 2, \ldots, m_i$

k = server index, $k = 1, 2, \ldots, S$

n_i = total number of request for Component i

The capacity constraint $c(s, C_j^i)$ is imposed so that Version C_j^i is allocated no more than n_i times, where n_i is the total number of requests of Component i. This is because there is nothing to gain by allocating the same version twice to a request.

The edge (C_j^i, Srv_k) has two types of capacities. The actual capacity $c(C_j^i, Srv_k)$ is infinity, whereas the basic capacity $c_b(C_j^i, Srv_k)$ represents the resource requirement of C_j^i. Any flow through this edge must be a multiple of the basic capacity $c_b(C_j^i, Srv_k)$ since a flow through this edge means allocation of a version and to allocate a version the resource requirement must be fulfilled completely. A multiple flow through this edge indicates multiple allocation of the version C_j^i for multiple requests. For example, basic capacity, $c_b(C_j^i, Srv_k) = 5$ and amount of flow through this edge, $f(C_j^i, Srv_k) = 15$ means that version C_j^i is allocated 3 times from Server k, and there are at least 3 requests for this version. If there were less than 3 requests then the version C_j^i would not have been allocated 3 times.

The capacity constraint $c(Srv_k, t)$ is imposed so that the total amount of resource consumption of all allocated versions from Server k does not exceed the resource available at Server k.

3.3. A Simple Example of the Proposed System

Let there be 2 components ($\boldsymbol{C}^1$ and $\boldsymbol{C}^2$) with 2 alternative versions for Component 1 (C_1^1 and C_2^1) and a single version for Component 2 (C_1^2). Consider 2 servers in the system with Server 1 (Srv_1) containing versions C_1^1 and C_1^2 and Server 2 (Srv_2) containing versions C_2^1 and C_1^2. Here the Version C_1^2 is hosted by both servers. Figure 3-3 depicts the architecture of this example. Specification of the components is presented in Table 3-1.

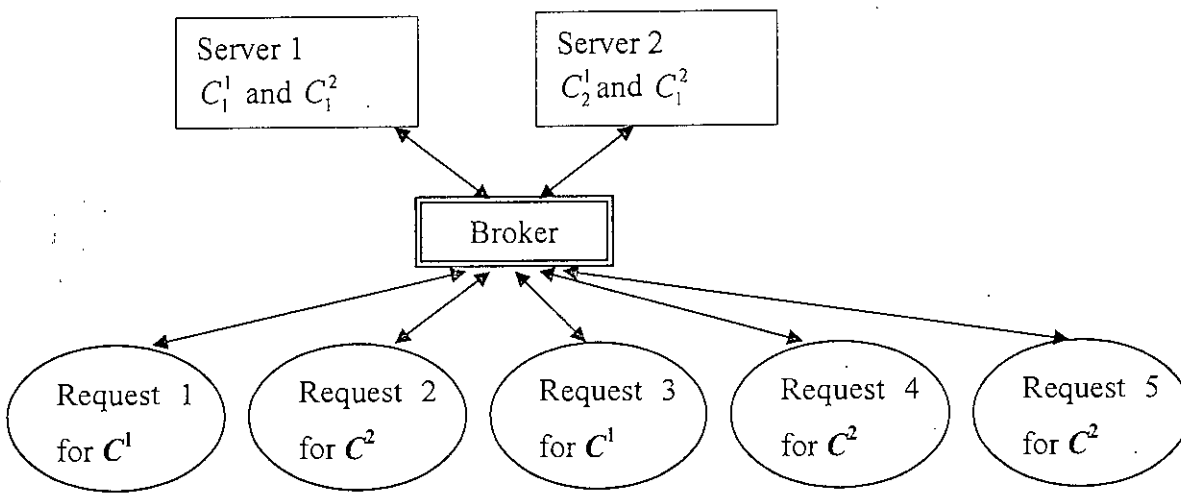


Figure 3-3 An example of the proposed system with 2 servers and 2 components.

Table 3-1 Resource requirements and containing servers of each of the versions of the two components.

C_1^1	C_2^1	C_1^2
Resource requirement: $w_{11}^1 = 4,\ w_{12}^1 = 3$ Hosted By: Srv_1	Resource requirement: $w_{21}^1 = 4,\ w_{22}^1 = 3$ Hosted By: Srv_2	Resource requirement: $w_{11}^2 = 4,\ w_{12}^2 = 3$ Hosted By: Srv_1, Srv_2

Here w_{jr}^{i} denotes the r-th resource requirement of C_j^i. In the above example $r = \{1, 2\}$ since there are two resource types, namely Resource 1 and Resource 2. C^1 has two versions, thus each request for C^1 can be served in one of the four ways. The satisfaction values for the four combinations along with the allocations are given in Table 3-2.

Table 3-2 An example of satisfaction values and corresponding allocations for the components.

Satisfaction (Random Value)	Corresponding Allocation
$U_{0,0}^1 = 0.0$	$\{\}$
$U_{0,1}^1 = 8.0$	$\{C_2^1\}$
$U_{1,0}^1 = 9.0$	$\{C_1^1\}$
$U_{1,1}^1 = 3.2$	$\{C_1^1, C_2^1\}$
$U_0^2 = 0.0$	$\{\}$
$U_1^2 = 8.8$	$\{C_1^2\}$

The network flow graph for this system requires 3 vertices to represent the versions and 2 vertices to represent the servers. The network flow mapping is shown in Figure 3-4. The first set of parentheses for each edge denotes the capacity of that edge where the second set denotes the amount of flow through that edge in the optimal allocation of versions.

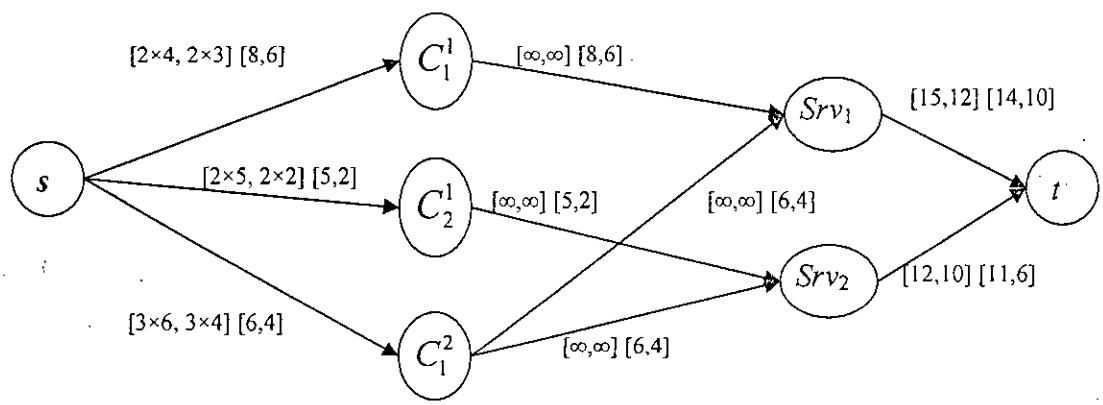


Figure 3-4 Network flow graph for the example stated above.

The number of allocations of each version of each component is found by executing the proposed network flow algorithm on the above graph. Table 3-3 shows the number of allocations (Allocation Count) of each version:

Table 3-3 Allocation count and the associated servers for each version of each component.

Component	Allocation Count	Allocated From
C_1^1	2	Server 1
C_2^1	1	Server 2
C_1^2	2	Server 1, Server 2

Now, these allocated components can be distributed among the five requests in many different ways. In the proposed system the optimal allocation of the requests to the allocated versions is one that maximizes total satisfaction of the service. The optimal allocation for the above example is given in Table 3-4.

Table 3-4 The optimal allocation and the satisfaction of the reqnests in the optimal allocation.

Requests	Allocated Versions	Satisfaction
Request 1	$\{C_1^1, C_2^1\}$	3.2
Request 2	$\{C_1^2\}$	0.88
Request 3	$\{C_1^1\}$	0.90
Request 4	$\{C_1^2\}$	0.80
Request 5	Rejected	0.0

Since the server capacities are limited all the requests can not be accepted. In the above example Component 2 is allocated twice from two different servers though there were three requests for it. So, one request for Component 2 must be rejected.

3.4. The Push Relabel Algorithm to Solve the Mapped Network Flow Graph

In the original push relabel algorithm there were two types of operations: Push and Relabel. A significant difference of the proposed algorithm from the typical flow algorithm is that, in the typical method, there is no need to push back any flow from the sink. But in the proposed method, since it is not known previously which allocation is better, at each iteration a new allocation is tried and compared with the last best allocation. Now to try a new allocation a previous allocation may have to be cancelled, since the capacities of the servers are limited. To perform the cancellation process, a new type of operation is defined, called the *Backward Push Operation* and the normal push is called the *Forward Push Operation*. These push operations are described in detail in Section 3.4.2 and in Section 3.4.3. Each vertex is associated with a height and a push can only take place from a higher vertex to a lower vertex. The purpose of the height function is to allow pushes in new directions rather than using the same direction again and again. Initially the height of all vertices except the source is zero. This height increases gradually with each relabel operation described later in Section 3.4.4.

3.4.1. Initial Flow

A typical flow algorithm starts with a zero flow through each edge of the network flow graph. But the proposed algorithm performs significantly better if an initial flow rather than the zero flow is assigned to each edge. That is, the proposed network flow algorithm starts from an initial allocation of components and moves towards a better allocation by performing push and relabel operations whenever possible. The initial allocation presented in this thesis is a greedy solution of the problem. In the first phase of initialization all possible requests are served with the best available version at that moment. In this phase at most one version is allocated to each request. In the next phase, if there are still available resources the requests are served with a redundant version that maximizes total satisfaction of a request. This process is continued until no more versions can be allocated to any request. To make the

complexity of this initialization process independent of the number of requests, the initial flow is computed separately for each component. For each component the requests for that component are processed group by group and for each group a version is selected and is assigned from one of its containing servers. A queuing strategy described later in Section 3.5 is used to find the allocation for each component.

3.4.2. Forward Push

All normal push operations are called the forward push operations. Even the pushes from the versions to the source are also forward push operations. The forward push operations are:

Source → *Version*: This push operation takes place in the initialization phase. A push from the source to each version is performed such that the (source, version) edge is saturated for all versions.

Version → *Source*: This push operation is performed when the excess flow of a version can not be passed to sink. To maintain the flow conservation property the excess flow must be returned to the source when the algorithm terminates. With the iterations, the height of each overflowing vertex will be increased and eventually will be greater than the height of the source. At this time, the excess of the version will be pushed to the source.

Version → *Server*: Since the versions are not directly connected to the sink, the excess of a version must be pushed to sink via a server meaning the allocation of that version from that server. A version can reside in more than one server, so the requests for that version can be served from a set of multiple servers. Thus a version tries to push its excess flow (which includes the number of requests for that version) to multiple servers to find a path to sink. To push the excess, a version must be raised to a height that is greater than at least one of its adjacent servers. When the version's height becomes greater than the source's height, it is assumed that the remaining excess of

that version can not be passed to sink and is returned to the source, thus completing the push operations from that version.

Server → *Sink*: A push from the server to the sink implies the allocation of one or more components that reside in that server and it is possible only when the server has sufficient resources to execute those versions. In terms of the flow network, it can be said that the excess flow of a server that can be pushed to sink must be less or equal to the residual capacity of the (server, sink) edge. After each push to sink, the number of allocations of each version is updated.

Server → *Version*: All the excess flows of the servers can not be passed to the sink, because of the limited capacities of the (server, sink) edges. When there is an overflowing server, but the residual capacity of the (server, sink) edge is not enough to push the excess, the excess must be returned to the version from where it has come, so that the version can try a new sever to push its excess.

Sink → *Server*: This push operation is performed for the de-allocation of a version. Since a version is not directly connected to the sink, the de-allocation of a version requires two pushes one after another: one from Sink → Server and the other from Server → Version.

The push operation from vertex u to vertex v is performed as follows:

Push $(u, v, d(u, v))$
// $d(u, v)$ = Amount of flow that can be pushed from vertex u to vertex v
// $d(u, v)$ = minimum of $e[u]$ and $c_f(u, v)$
// $c_f(u, v)$ denotes the residual capacity of the edge (u, v)
Applies when: u is overflowing and $h[u] > h[v]$
Action:

$e[u] = e[u] - d(u, v)$	// Decrease excess of vertex u
$e[v] = e[v] + d(u, v)$	// Increase excess of vertex v
$f(u, v) = f(u, v) + d(u, v)$	// Increase flow from vertex u to vertex v
$f(v, u) = f(v, u) - d(u, v)$	// Decrease flow from vertex v to vertex u

$c_f(u, v) = c_f(u, v) - d(u, v)$ // Decrease residual capacity of edge (u, v)

$c_f(v, u) = c_f(v, u) + d(u, v)$ // Increase residual capacity of edge (v, u)

3.4.3. Backward Push

Backward push means the cancellation of the allocation of one or more components. It is actually performed by using a combination of forward pushes. When there is excess flow in a server, but the residual capacity of that (server, sink) edge is not enough to push the excess flow, a backward push takes place in which one or more previously allocated versions are de-allocated. De-allocation of a version means that the resource flow for that version is pushed back from the sink to the version via the server. The backward push thus results in an increase of the residual capacity of the corresponding (server, sink) edge.

Applies when: There is excess flow in Server k but the residual capacity of the (Server k, sink) edge is less than the excess flow of Server k.

Action:

REPEAT

$v \leftarrow$next adjacent vertex of Server k

$d(v) = \min(e[k]$, amount of flow that can be pushed back to version $v)$

//Push $d(v)$ amount of flow from the sink to the version v via Server k

//These push operations are normal forward push operations

Push$(t, k, d(v))$

Push$(k, v, d(v))$

UNTIL $c_f(k, t) \geq e[k]$ OR all the adjacent vertices have been checked

3.4.4. Relabel

Each vertex is associated with a height as in the original push relabel algorithm described in Section 2.4.4.3. The excess flow of a vertex can only be pushed to a

lower adjacent vertex. When there is an overflowing vertex but all of its adjacent vertices are higher than it, then a relabel operation takes place.

Action:

Update height of overflowing vertex u to $h[u]$, such that,

$$h[u] = 1 + \min_{v}\{h[v] : v \in ADJ[u]\}$$

3.4.5. Solving a Knapsack Problem at Each Server

Since the capacity of each server is limited all the requested versions hosted by that server can not be allocated. Determining the optimal set of versions to allocate from a server considering all other requests and all other servers become a Multi Knapsack Problem which is computationally very expensive. Rather, a Knapsack Problem at each server is solved to find the optimal set of versions for that specific server without considering allocations from other servers. To solve the Knapsack Problem greedily all the versions hosted by a specific server are sorted in decreasing order by their (satisfaction/aggregate resource requirement) values, where aggregate resource requirement is computed as follows:

Aggregate resource requirement = $\sqrt{\sum_{r=1}^{R} (w^{i}_{jr})^{2}}$

Thus when an overflowing server will be pushed (that is when an allocation is found) the versions with greater satisfaction values and smaller resource requirements will be considered first. Though this process does not guarantee optimal allocation from a server, it considers both satisfaction and resource constraints. The best part of this optimization technique is that it requires some additional work only in the graph construction phase. After the graph has been constructed in the above mentioned sorted order, each server will automatically consider the versions for allocation in that order. The graph even need not be updated for different set of requests.

3.4.6. Snapshot

When the network flow algorithm terminates it is not guaranteed that the allocation at the termination time is the best allocation. Consider the following example with two components, a single server and two requests. Satisfaction and resource requirements of the components are given in Table 3-5.

Table 3-5 Satisfaction and resource requirements of Component 1 and Component 2.

Component 1 (C^1)	Component 2 (C^2)
Satisfaction = 0.8	Satisfaction = 0.9
Resource Requirement = 5	Resource Requirement = 4
Number of request for C^1= 1	Number of request for C^2 = 1

The network flow mapping for this problem is as follows:

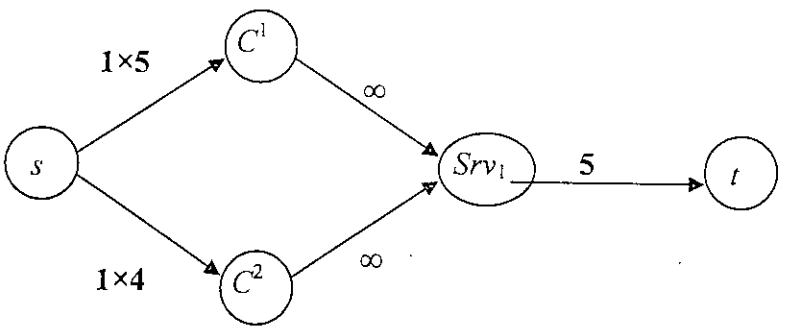


Figure 3-5 Flow network for the snapshot example.

Here, resource available at server Srv_1: 5.

Since the server is able to execute either C^1 or C^2, but not both, the system should chose to serve C^2, since it will maximize the satisfaction of the client admitted to the system. When the network flow algorithm is run in the above scenario, at first a flow through C^1 will be passed to t via Srv_1. Then C^2 will make its way to sink by a backward push, de-allocating C^1. Now again C^1 will try to reach sink by pushing back C^2 and after some oscillations, excess of C^2 will come back to source, which means that C^1 is served to Request 1, which is not a desirable situation.

In this simple example, the final allocation depends only on the indexing of the version nodes, but in a large network, backward pushes explore different allocations and it can not be said conclusively that the allocation that is found at the time of termination is the best. That is why, after each push to the sink, a snapshot of the current allocation is taken and the allocation that gives the maximum total satisfaction is saved.

Operation Snapshot:

Applies when: A forward push from Server → Sink is performed.

Action:

Find an allocation of the selected versions to the request set.
Current satisfaction = total satisfaction of the current allocation.
IF current satisfaction > the last best satisfaction THEN
    Save current satisfaction as the best satisfaction
    Save current allocation as the best allocation.

A snapshot is to be taken whenever a push may result in a better allocation. Since none of the pushes other than the Server → Sink forward push results in the allocation of a new component, the snapshot is taken only after the Server → Sink forward push operation.

3.5. The Allocation of Versions to Requests

Whenever there is a push to sink, a new component is allocated and the total satisfaction of the service is to be computed considering this new allocation. The network flow algorithm computes only the number of allocation of each version of each component. Since, the request set is decoupled from the flow network, at each snapshot to compute the total satisfaction an allocation of the selected versions to the requests is to be found. Since redundancy is allowed each request is served with a set

of versions, where the empty set denotes the rejection of the request. If a request for Component i is not rejected then at least one version and at most m_i versions are allocated to it, where m_i is the number of versions of Component i.

The computation for the allocation of selected versions of a component to requests is totally independent to the allocation of other components. So the allocation for the requests of each component can be considered separately.

Let us consider a simple case of two requests (q_1 and q_2) for Component 1, where Component 1 has three different versions namely C_1^1, C_2^1 and C_3^1. Some of the possible allocations of these three versions to two requests are shown in Table 3-6.

Table 3-6 Some possible allocation of three versions to two requests.

Allocation 1	Allocation 2	Allocation 3
$q_1 = \{ C_1^1, C_2^1, C_3^1 \}$	$q_1 = \{ C_1^1 \}$	$q_1 = \{ C_1^1, C_3^1 \}$
$q_2 = \{\}$	$q_2 = \{ C_2^1, C_3^1 \}$	$q_2 = \{ C_2^1 \}$

There are many other such allocations. Among the all possible allocations, the allocation that maximizes total satisfaction should be chosen in the proposed model.

In the proposed algorithm, a greedy strategy is implemented to find the allocation. In this greedy strategy, the allocation for each component is computed separately. The allocation thus computed may be sub-optimal but its complexity is less than that of the optimal allocation and is thus capable to allocate in real time. Since the model is proposed for real time web services, the clients should not be kept waiting too long for the responses. And since the allocation is to be computed for each snapshot and a snapshot is to be taken for each push to sink; total complexity of the algorithm will be increased significantly with the increased complexity of the allocation.

Let $allocationCount_{ij}$ denote the number of allocations of Version j of Component i. Number the requests of Component i is n_i. Clearly, $allocationCount_{ij} \leq n_i$.

A queue structure (Q) is implemented to store the n_i requests of Component i. At each iteration, a set of requests is popped from the front of Q, assigned a new version and sent back to the rear of Q for further allocation of versions. Processing a set of requests Q_{front} from the front of Q includes the following steps:

- The current allocation of the set Q_{front} is determined.
- The version that will yield maximum increase in satisfaction considering the current allocation of the requests of Q_{front} is computed. Let this version be C_j^i.
- Now C_j^i is allocated to the first $allocationCount_{ij}$ requests of Q_{front}. These requests are now moved to the rear of the Q.
- Rest of the requests of Q_{front} remains at the front of Q.
- The whole process is repeated until no more allocation is possible.

Thus the n_i requests for Component i are processed group by group. Before allocating a redundant version to a group, it is ensured that each of the n_i requests has got at least one version of the requested component. This restriction is imposed to avoid allocations like "Allocation 1" in Table 3-6.

An example: Consider the allocation for Component 1 with 4 alternative versions where 5 requests have been placed for this component. Table 3-7 shows the data needed for finding the allocation and Figure 3-6 shows the steps to find the allocation from the given data.

Table 3-7 Data for the allocation of Component 1 to five requests.

Versions of Component 1:	C_1^1, C_2^1, C_3^1 and C_4^1
Requests for Component 1:	$q_1, q_2, \ldots, q_5$
Number of allocations of each version:	$allocationCount_{11} = 3$
	$allocationCount_{12} = 2$
	$allocationCount_{13} = 2$
	$allocationCount_{14} = 1$

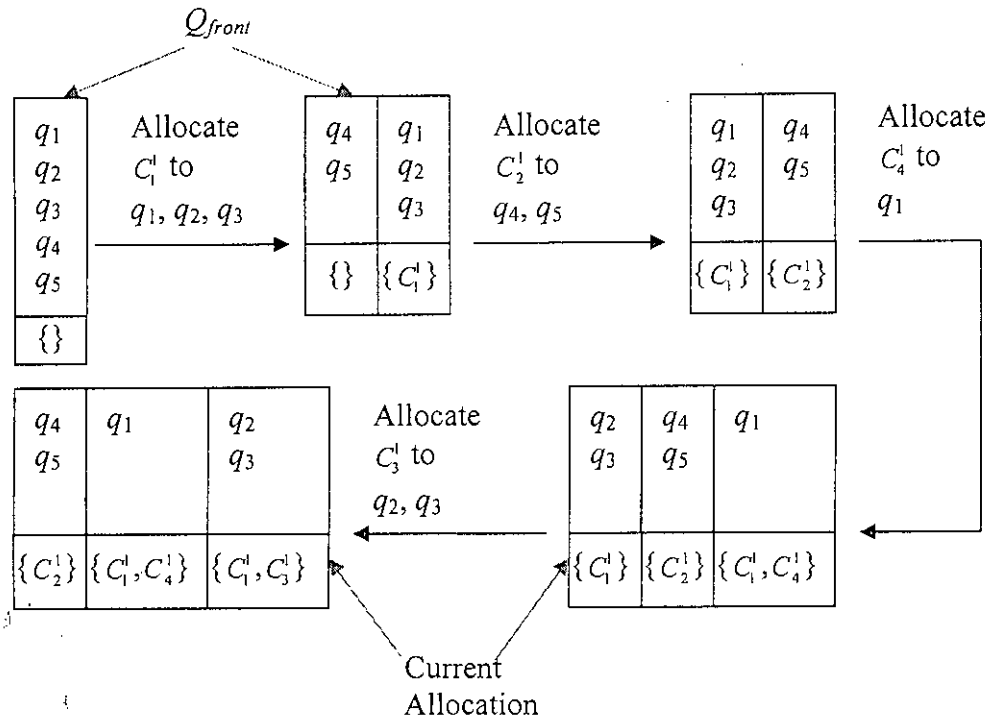


Figure 3-6 An example of allocation of versions to requests.

Requests are shown separately in the queue for better understanding of the process. Actually the queue holds only the number of requests of each group. In this example the following is assumed.

$$\text{Satisfaction}(C_1^1) \geq \text{Satisfaction}(C_2^1) \geq \text{Satisfaction}(C_3^1) \geq \text{Satisfaction}(C_4^1)$$

Thus the allocation for component 1 has been started from the highest satisfaction version that is from Version 1. Also at the third iteration C_4^1 is allocated before C_3^1, because the following is assumed in this example.

$$\text{Satisfaction}\{C_1^1, C_4^1\} > \text{Satisfaction}\{C_1^1, C_3^1\}$$

Since Component i has m_i alternative versions, the maximum number of groups that can reside in the Queue is 2^{m_i} for Component i. Now, each group can be pushed to the Queue only once and so can be popped only once. At most $O(m_i)$ time is required to process each group. Thus finding the allocation of Component i requires no more

than $O(m_i 2^{m_i})$time. For N components the complexity of finding an allocation is at most $O\left(\sum_{i=1}^{N} m_i 2^{m_i}\right)$.

3.6. Stopping Criteria

In the typical flow maximization algorithm, the iteration stops when no more flow can be pushed to the sink, i,e, when the heights of the vertices become greater than the height of the source. In the original Push Relabel algorithm by Goldberg, the height of the source is maintained at $|V|$, the number of vertices in the flow network. The excess of any vertex that can not be pushed to sink ultimately comes back to the source.

In a typical flow network it is sufficient to determine whether a push can be possible, that is whether more flow can be pushed through a potential path. But in the proposed model, it also has to be determined whether a push is possible by canceling a previous push. The flow that is pushed back, now tries to find another path to sink. Because of these backward push operations some flow can always be pushed to the sink. Since the maximum number of push operations is directly related to the height of the source, the height of source must be chosen in such a way so that no excess should come back to the source immaturely, that is without exploring other possible options. On the other hand, since the number of iterations of the proposed algorithm increases with the increase in source's height, source height can not be raised too high. This is why the source height is considered as a user defined parameter in the proposed algorithm. Figure 4-7 and Figure 4-8 of Chapter 4 show the variation in performance and running times of the proposed algorithm with the variation in the height of the source. While executing the proposed algorithm the source height is maintained at $2|V|$, because after that a small increase in total satisfaction causes a very large increase in total running time of the algorithm.

3.7. The Algorithm

The proposed network flow heuristic algorithm is as follows:

```
PROCEDURE Push_Relabel(G, s, t)
    FOR each vertex v other than s
        h[v] = 0 // Height of vertex v is initialized to 0
        e[v] = 0 // excess flow at vertex v is initialized to 0
    END FOR
    h[s] = 2 × (M+S+2) // Total number of vertices = M+S+2
    e[s] = ∞
    FOR each vertex v adjacent to s
        f = c(s, v)
        Push(s, v, f) //Push f amount of flow from s to v
    END FOR
    Assign an initial flow to each edge of the network
    WHILE there is an overflowing vertex
        u ← lowest height vertex among all overflowing vertices
        Relabel(u)
        IF Type(u) = "Version" THEN
            v ← Lowest height vertex among all adjacent vertices of u
            f = e[u]
            Push(u, v, f)
        ELSE // Type(u) = "Server"
            f = min (e[u], c_f(u,t))
            Push(u, t, f)
            Update the number of allocations of each version
            Snapshot()
            IF u is still overflowing THEN //perform backward push
                FOR each v where h[u] > h[v]
                    f = amount of flow that can be pushed back to v
                    Push(t, u, f)
```

```
                    Push(u, v, f)
                END FOR
            END IF
            f = min (e[u], c_f(u,t)
            Push(u, t, f)
            Update the number of allocations of each version
            Snapshot()
            IF u is still overflowing THEN
                FOR each v
                    f = amount of flow that can be pushed back to v
                    Push(u, v, f)
                END FOR
            END IF
        END IF
    END WHILE
    Return the best allocation saved by Snapshot operation
END PROCEDURE
```

The assignment of initial flow to each edge and the Push, Relabel and snapshot operations are described in detail in Section 3.4.

3.7.1. Complexity Analysis

The complexity of the proposed network flow heuristic algorithm depends on the network flow graph construction process and on the Push-Relabel algorithm. Since the requests set is decoupled from the flow network, the graph need not be reconstructed for each set of requests. Only the (source, version) edge capacities vary with the number of requests and need to be updated for each set of requests.

For N components each with m_i (i = 1, 2, … , N) alternative versions, there are a total of $M = \sum_{i}^{N} m_i$ vertices in the versions set. Thus total $O(M)$ time is required to update the capacities of the M edges of (source, version) type.

Before applying the network flow algorithm an initial flow is assigned to the network. This initialization procedure uses the queuing strategy described in Section 3.5. However in the initialization phase, processing each group of request requires $O(m_i S)$ time to find the current best version and the server from which to assign the version. The complexity of the initialization procedure is thus $O\left(\sum_{i=1}^{N} m_i S 2^{m_i}\right)$.

The complexity of the Push-Relabel network flow algorithm depends on the number of push and relabel operations. A worst case complexity analysis of the algorithm is presented in the next section.

3.7.1.1. Complexity of the Push Operations

In the worst case scenario the following will happen in several iterations:

- Total excess flow of a version will be pushed to the server, but may not be pushed to the sink due to the unavailability of server resources.
- A Server → Version push will take place because of this lack of server capacity which will increase the excess flow of the corresponding version.
- The version now will look for other servers to push its excess flow.

In each iteration the version height is increased by at least 2. This height increase is required for the next iteration so that the version can perform further pushes to the servers. A simple example makes it clear. Initially the height of a version is 0. In the first iteration it will be 1 to make a Version → Server push effective. Then, the server height must be 2 to push the excess flow to the version that can not be pushed to sink. Now, to perform another push the version height must be 3. Thus to attain the maximum height of a version which is $2|V|+1$, at most $|V|+1$ iterations will be needed.

The number of push operations for different types pushes is computed as follows:

Source → Version Push:

Since there are total M versions, the number of pushes of this type is $O(M)$.

Version → Source Push:

In the worst case some amount of excess from each version will be returned to source, thus requiring a total of $O(M)$ pushes.

Version → Server Push:

In each iteration at most S Version → Server push operations are possible, since a version of a component can reside in at most S servers. Thus the total number of push operations for $(|V|+1)$ iterations is $(|V|+1)S$. For M versions, the total complexity becomes $M(|V|+1)S = O(M|V|S)$.

Server → Sink Push:

In each iteration a Server → Sink push might not be possible due to the unavailability of resources. But the complexity does not change since resource availability checking need to be performed in each iteration. For each Server → Sink push $O(M)$ search is required to update the number of allocations of each version. Thus the total complexity of all pushes from all servers is $O(S|V|M)$ for the $O(|V|)$ iterations. After each push to sink, a snapshot is taken whose complexity is $O\left(\sum_{i=1}^{N} m_i 2^{m_i}\right)$ as described in Section 3.5. Multiplying with this factor, the Server → Sink push complexity becomes $O\left(S|V|M\sum_{i=1}^{N} m_i 2^{m_i}\right)$.

Server → Version Push:

A push from a server to a version can be part of either a forward push or a backward push. Total complexity of all *Server → Version* push operations is the same as Server → Sink push operations and is $O(S|V|M)$. Complexity of backward push operations also include the number of pushes from the sink to the server which is also $O(S|V|M)$.

Total complexity of all push operations

$$= O(M) + O(M) + O(M|V|S) + O\left(S|V|M\sum_{i=1}^{N} m_i 2^{m_i}\right) + O(S|V|M).$$

$$= O\left(S|V|M\sum_{i=1}^{N} m_i 2^{m_i}\right)$$

3.7.1.2. Complexity of Relabel Operations

The relabel operations are applied to the versions and to the servers only. The source and sink can not be relabeled. The maximum number of relabel operation per server and per version is $O(|V|)$.

At each relabel operation, the height of a vertex is raised to (1+ the minimum height of all of its adjacent vertices). Thus each server relabel operation performs $O(1+M)$ search and each version relabel operation performs $O(1+S)$ search to find the minimum height adjacent vertex.

Total complexity of all server relabel operations is $O(S|V|M)$ and

Total complexity of all version relabel operations is $O(M|V|S)$.

Total relabel complexity = $O(SM|V|)$

$$= O(SM(S+M+2)) = O(S^2M + SM^2)$$

3.7.1.3. Total Worst Case Complexity of the Algorithm

Total complexity of the algorithm includes the time for graph update process, for initialization and for push and relabel operations. Thus total complexity of the algorithm is:

$$= O(M) + O\left(\sum_{i=1}^{N} m_i S 2^{m_i}\right) + O\left(S|V|M\sum_{i=1}^{N} m_i 2^{m_i}\right) + O(SM|V|)$$

The total complexity is dominated by the term

$$O\left(S(|V|+2)M\sum_{i=1}^{N} m_i 2^{m_i}\right) = O\left(S(M+S)M\sum_{i=1}^{N} m_i 2^{m_i}\right) = O\left((SM^2 + S^2M)\sum_{i=1}^{N} m_i 2^{m_i}\right)$$

Here, m_i, the number of alternative versions of Component i, is assumed to be very small, since developing alternative versions require more effort and cost. In this thesis, m_i is assumed to be at most 5, thus $2^{m_i} = 32$, for the maximum value of m_i. This is why the snapshot complexity does not contribute much to the complexity of the proposed algorithm. The total complexity of the algorithm as computed above depends on the number of versions and on the number of servers and is independent of the number of requests, n.

3.8. Chapter Summary

In this chapter the mapping of the proposed model to the network flow maximization problem along with the heuristic algorithm to solve the model has been described. The optimization techniques that have been implemented to improve the performance and/or running time of the proposed algorithm are also presented in this chapter. The chapter concludes by providing a worst case complexity analysis of the proposed algorithm. The next chapter confirms this complexity analysis by testing the running times of the proposed heuristic for randomly generated data sets. Experimental results showing the improvement of the proposed algorithm over a Greedy algorithm have also been presented in the next chapter.

Chapter 4. Result Analysis

The proposed network flow heuristic algorithm will not always produce the optimal solution. To compare the output of the proposed algorithm a Brute Force program is written that explores all possible solutions and picks up the optimal one. The proposed heuristic has also been tested with a Greedy algorithm. These algorithms along with the experimental results are presented in the next sections.

4.1. Brute Force Algorithm

Since the Brute Force algorithm explores all possible solutions the solution space of this algorithm grows exponentially with the problem size. Thus the optimal solution using Brute Force algorithm can not be found even for moderate size of problems. For example, if there are 3 alternative versions of each component, then each request can be served in 2^3 ways, since multiple versions of a component can be served to the same request. This information can be encoded in 3 bits. Now, for 10 requests the size of the solution will be 30 bits. And the total number of all possible solutions among which an optimal one is to be selected is thus 2^{30}. This is why the proposed algorithm is compared with the Brute Force algorithm for only smaller data sets. To compare the performance of the proposed algorithm a Greedy algorithm is presented in this thesis and is described in detail in Section 4.2. The results obtained from the proposed algorithm, the Brute Force algorithm and the Greedy algorithm are presented in Section 4.4.

4.2. Greedy Algorithm

The greedy algorithm takes decision locally based on the requests. At the first iteration the algorithm assigns no more than 1 version to each request. The greedy algorithm always tries to allocate the version with the highest satisfaction. Since each version can reside in multiple servers, from the list of containing servers the first one that has sufficient resource to execute the version is selected for that request. If the

highest satisfaction version is already allocated to the same request or no server has the resource to execute it, then the version with the next highest satisfaction is considered. After scanning all the requests once, the algorithm gives a second pass and allocates a second version to the requests whenever possible. This phase increases the satisfaction of any request by means of redundant allocation of versions. The procedure is continued until no more versions can be allocated to any request. The inherent idea here is to maximize satisfaction by serving more requests and allowing redundancy when the total number of accepted requests can no more be increased.

The Greedy Algorithm:

```
REPEAT
    FOR each request q_i
        c = Component requested by q_i
        v = Find_Version_To_Allocate(c, q_i)
        IF v != NIL
            Add v to the set of allocated versions for q_i
        END IF
    END FOR
UNTIL (there is no change in Total Satisfaction)
```

The Find_Version_To_Allocate(c, q_i) procedure finds a version v of the component c such that,

- v is not already allocated to q_i
- Resource requirement of v can be satisfied by any of its containing servers, and
- Allocation of v to q_i results in the maximum increase of satisfaction among all other versions of component c that satisfies the above two constraints.

If no such version can be found the Find_Version_To_Allocate(c, q_i) procedure returns *NIL*.

4.3. Data Generation

Random data are generated to compare the performance of the proposed algorithm. To generate data the following parameters are defined:

MAX_COMP = Maximum number of components

MIN_COMP = Minimum number of components

MAX_SERVER = Maximum number of servers

MIN_SERVER = Minimum number of servers

MAX_VER = Maximum number of versions per component

MIN_VER = Minimum number of versions per component

MAX_REL = Maximum reliability

MIN_REL = Minimum reliability,

MAX_DIM = Maximum resource dimension

MAX_REQ = Maximum number of requests

MIN_REQ = Minimum number of requests

MAX_RS = Maximum resource requirement per version

MIN_RS = Minimum resource requirement per version

Let $U_{0,x}$ denote a random floating point number in the range [0, x].

Let $I_{a,b}$ denote a random integer in the range [a, b].

Random data are generated as follows:

Number of Requests, $n = MIN\_REQ + I_{0,(MAX\_REQ-MIN\_REQ)}$

Number of Components, $N = MIN\_COMP + I_{0,(MAX\_COMP-MIN\_COMP)}$

Number of Servers, $S = MIN\_SERVER + I_{0,(MAX\_SERVER-MIN\_SERVER)}$

For each component i, the number of alternative versions m_i is generated at random using the following formula:

$$m_i = MIN\_VER + I_{0,(MAX\_VER - MIN\_VER)}$$

For each version j of each component i (C_j^i) the following data are generated:

Reliability of C_j^i, $R_{ij} = MIN\_REL + (MAX\_REL - MIN\_REL) * U_{0,1}$

Resource requirement of $C_j^i = MIN\_RS + I_{0,(MAX\_RS - MIN\_RS)}$

Number of servers containing C_j^i: A random number in the range [30% of S, 70% of S]

The servers that contain the version C_j^i are generated as random numbers in the range [1, S].

The server capacities a_{ik} are generated so as to satisfy 30%-100% of the requests n, from a total of S servers. Thus,

a_{ik} = (Average resource (of type k) requirement of the versions stored in server i) × (x% of n)/S, Here x = {25, 50, 80,100}

The satisfaction values of the combinations of allocations are generated from the reliabilities of the versions as follows:

- When a single version is allocated the satisfaction is the same as the reliability.
- When exactly two versions (C_{j1}^i and C_{j2}^i) are allocated,

  Satisfaction of the combined allocation,

  = Sum(R_{j1}^i, R_{j2}^i)+ Average(R_{j1}^i, R_{j2}^i) × $U_{0,0.5}$
- When more than two versions ($C_{j1}^i, C_{j2}^i, \ldots, C_{jl}^i$) are allocated,

  Satisfaction of the combined allocation,

  = Sum ($R_{j1}^i, R_{j2}^i, \ldots, R_{jl}^i$) - Average ($R_{j1}^i, R_{j2}^i, \ldots, R_{jl}^i$) × $U_{0,0.5}$

The observation here is that the satisfaction of a client increases by a great amount when the first redundant version is allocated to the client. After that the allocation of more redundant components increases satisfaction by a very small amount.

The following quantities are assumed for all randomly generated data sets.

MAX_COMP=20	*MAX_SERVER*=20	*MAX_VER*=5
MIN_COMP=10	*MIN_SERVER*=10	*MIN_VER*=1
MAX_REL=0.99	*MAX_RS*=20	*MAX_DIM=2*
MIN_REL=0.70	*MIN_RS*=5	

The other parameters are defined separately for different data sets and will be introduced in the next section.

The generated data sets are fed to the proposed network flow heuristic, to the Brute Force algorithm and to the Greedy algorithm. Table 4-1 provides a comparison of all three algorithms for randomly generated small data sets. Figure 4-1 and Figure 4-2 compares the proposed algorithm and the Greedy algorithm for randomly generated large datasets. Figure 4-3 shows how the difference in total satisfaction of the proposed algorithm and the Greedy algorithm varies with varying server capacities. Figure 4-4 compares the same for varying number of requests. Figure 4-5 shows that the maximum number of iterations needed by the proposed algorithm does not increase with increasing number of requests. However this number varies with the number of different web service components and servers and this variation is depicted by Figure 4-6. Figure 4-7 and Figure 4-8 shows how the performance of the proposed algorithm varies with different source heights.

4.4. Result Analysis

Table 4-1 shows the results obtained from the Brute Force algorithm, Greedy algorithm and the proposed algorithm for 10 randomly generated data sets. Here,

M = Total number of Version nodes

S = Total number of Servers

n = Total number of Requests

$Accpt$ = Total number of Accepted Requests

Sat = Total Satisfaction

Efficiency denotes the Total Satisfaction of the proposed algorithm with respect to the Brute Force algorithm and is computed as follows:

$$Efficiency = \frac{Sat_{proposed\ alg.} \times 100}{Sat_{Brute\ Force\ alg.}}$$

Table 4-1 Comparison of the proposed algorithm with Brute Force and a Greedy algorithm.

Data Sets			*Greedy Algorithm*		*Brute Force Algorithm*		*Proposed Algorithm*		*Efficiency (%)*
M	*S*	*N*	*Accpt*	*Sat*	*Accpt*	*Sat*	*Accpt*	*Sat*	
7	2	8	8	8.00	6	9.7	7	9.4	97
8	2	10	9	7.70	8	10.9	9	10.5	96
9	2	10	10	12.47	7	13.23	8	13.01	98
9	4	15	13	11.50	13	16.72	13	16.61	99
10	2	12	12	17.15	12	18.21	12	18.21	100
11	4	14	13	10.99	9	13.22	10	11.97	91
12	5	12	8	7.18	7	8.81	8	7.94	90
12	3	15	12	10.54	14	13.00	12	12.48	96
12	4	11	10	8.75	9	12.19	9	11.92	98
13	8	14	11	9.64	10	14.67	12	11.68	80

The *efficiency* of the proposed algorithm is more than 90% of the Brute Force algorithm for almost all input data sets. Here *efficiency* 90% means that the Total Satisfaction obtained from the proposed algorithm is 90% of that obtained from the Brute Force algorithm which provides the optimal solution. Though these input data sets are not realistic, they provide a measurement of the optimality of the solution of the proposed algorithm. The Brute Force algorithm can not be applied in real life scenarios, because it can not produce a solution even for moderate size of data sets, for example, datasets with more than 15 requests and more than 15 versions can not be handled by the Brute Force algorithm. On the contrary, the proposed network flow heuristic algorithm can handle thousands of requests in a distributed system with hundreds of versions of web service components. Thus the applicability of the proposed algorithm makes up for the fact that it produces suboptimal solutions.

For larger data sets the proposed algorithm is compared with the Greedy algorithm and the results are presented below.

Figure 4-1 shows the Total Satisfaction values for both the algorithms. Here server capacities are maintained so as to accept almost 50% of all requests and the total number of requests is varied from 300 to 800. Each data point in the graph represents the average output of 50 randomly generated data sets.

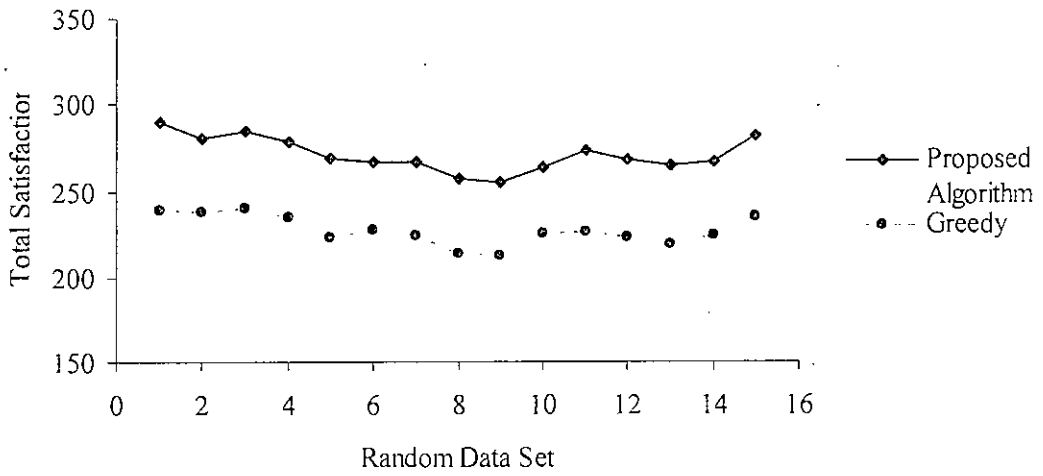


Figure 4-1 Comparison of Total Satisfaction for randomly generated data sets.

As expected, the proposed heuristic provides much better satisfaction values than that of the greedy algorithm. For most of the cases Total Satisfaction obtained from the proposed algorithm is almost 20% better than the Greedy algorithm. However this percentage varies greatly with the amount of available resources at the servers. This variation is depicted by Figure 4-3.

Figure 4-2 compares the Total Number of Accepted Requests generated by both the proposed algorithm and the Greedy algorithm for the same Random Data Sets of Figure 4-1.

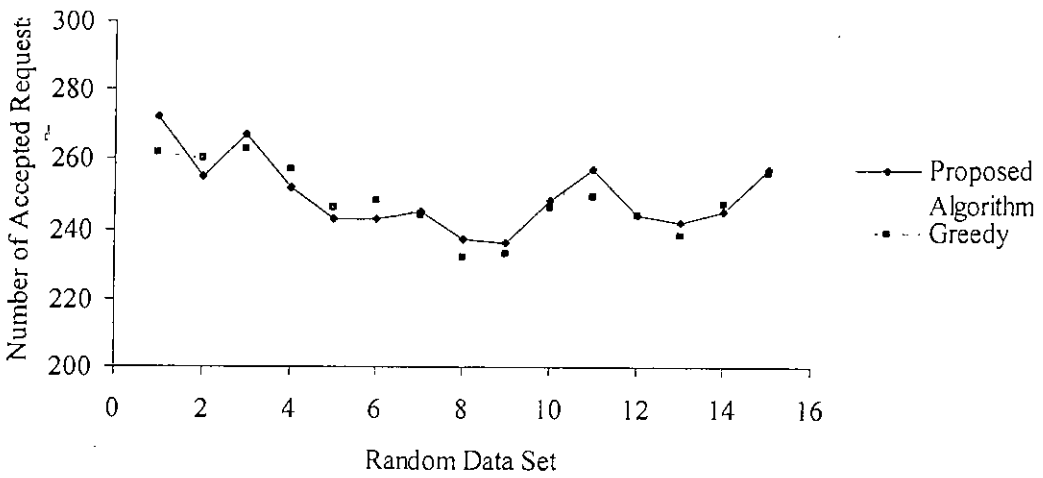


Figure 4-2 Comparison of Total Number of Accepted Requests for randomly generated data sets.

For a few cases the proposed algorithm provides better satisfaction values by rejecting more requests than the Greedy algorithm. Such scenarios are not unexpected since the proposed algorithm is designed to maximize total satisfaction. It can also be noticed that for more than 50% of the input cases the proposed algorithm accepts more requests as well as provides better satisfaction values compared to the Greedy algorithm.

Figure 4-3 shows the variation of percentage increase in Total Satisfaction of the proposed algorithm from the Greedy algorithm with respect to the variation of the amount of available resources at the servers. The percentage increase is computed as follows:

$$\% \text{ increase} = \frac{(Satisfaction_{proposed\ alg.} - Satisfaction_{Greedy\ alg.}) \times 100}{Satisfaction_{Greedy\ alg.}}$$

The % increase is computed for randomly generated 15 data sets for the following two cases:

High Resource: Amounts of available resources at the servers are generated so as to satisfy almost 80% to 100% requests.

Low Resource: Amounts of available resources at the servers are generated so as to satisfy almost 25% requests.

For each data set only the amounts of available resources are varied while all other parameters are held constant. Each data point in the graph represents the average output of 50 randomly generated data sets.

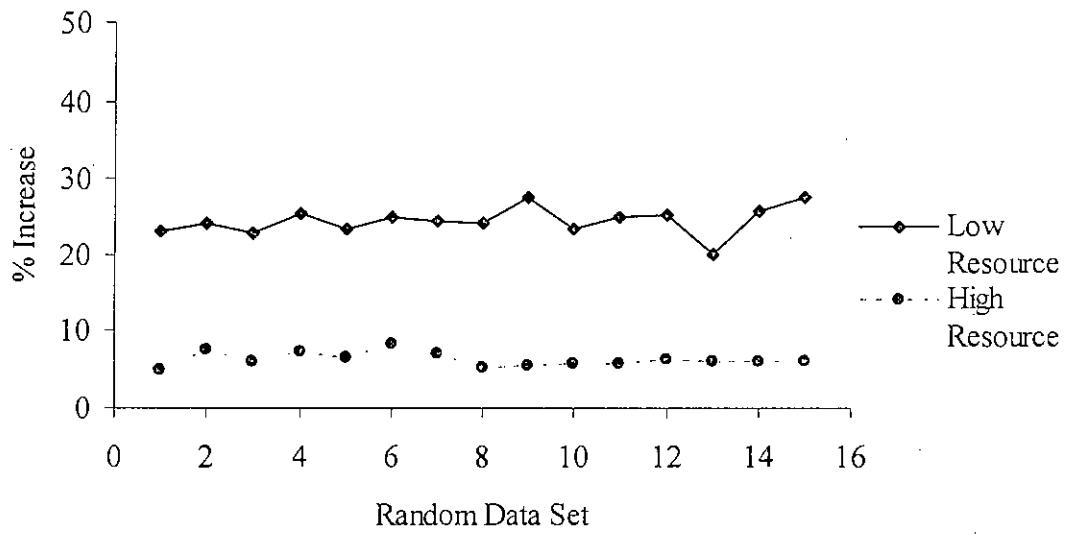


Figure 4-3 Percentage increase in Total Satisfaction for different server capacities.

From Figure 4-3 it is clear that when the servers have very limited resources but there are many requests, the allocation found by the proposed algorithm is much better than that found by the Greedy algorithm. This is because now there are many different options to choose and the proposed algorithm performs a much better search than the Greedy algorithm. For most of the cases 20% to 30% improvement from the Greedy solutions have been achieved. On the other hand when the servers have huge resources and almost 80% to 100% requests can be satisfied; there is not much scope

for the proposed algorithm to find much better solutions than the Greedy algorithm. Still in this case the solutions obtained from the proposed algorithm are almost 8%-10% better than the Greedy solutions.

Figure 4-4 shows how Total Satisfaction varies with increasing number of requests provided all other parameters are held constant.

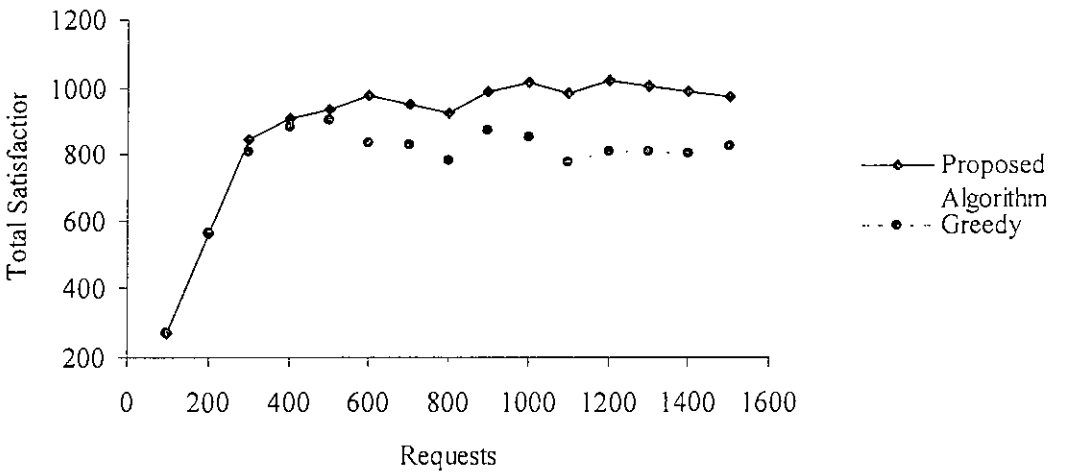


Figure 4-4 Comparison of Total Satisfaction for randomly generated data sets with increasing requests.

Since the numbers of requests are small for the first few data sets, all the requests are accepted and both the proposed algorithm and the Greedy algorithm produce similar results. The difference in Total Satisfaction values obtained from the proposed algorithm and the Greedy algorithm increases with increasing number of requests. This is because number of options increases with increasing requests and the proposed algorithm can explore the options more efficiently than the Greedy algorithm.

Complexity of the proposed algorithm depends on the number of push operations which in turn depends on the network flow graph size and on the height of the source. The most attractive part of the proposed algorithm is that its worst case complexity does not depend on the number of requests. The proposed algorithm has been executed for 12 data sets with increasing number of requests while keeping all other

parameters of the data sets constant. Figure 4-5 shows the total number of push operations in the main while loop of the proposed algorithm for the input data sets. When the number of requests is small, all the excess flows of most of the servers are passed to the sink. Thus there are no pushback operations for most of the servers and total number of iterations is small. Though the total number of push operations increases with increasing requests, its upper bound is determined by the size of the network and not by the number of requests.

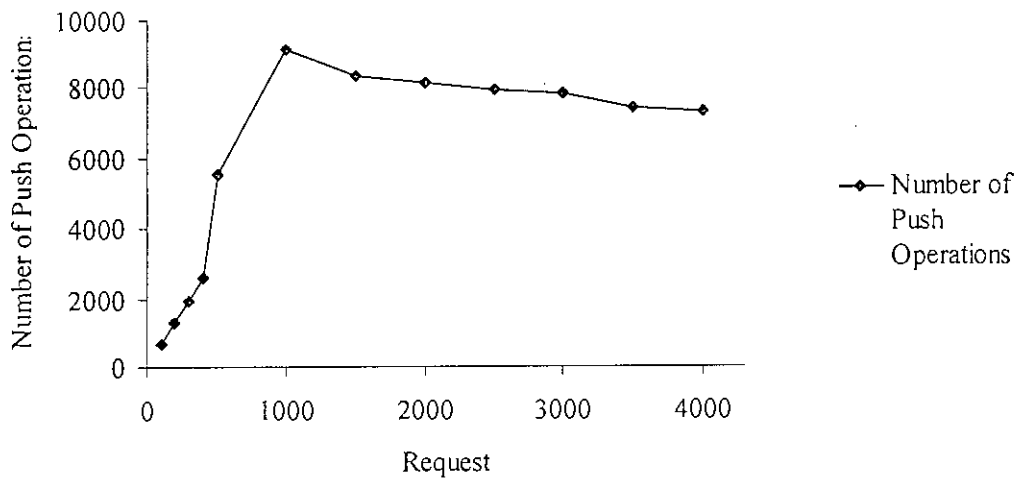


Figure 4-5 Total number of push operations performed by the proposed algorithm.

Figure 4-6 shows the running times of the proposed algorithm for three different network sizes. For all three networks the height of the source is held constant at 2|V|, where |V| denotes the number of vertices in the network. The algorithm has been run on a Personal Computer having Intel Pentium- 4 processor and 512 Mb of RAM. Each data point in the graph represents the average output of 50 randomly generated data sets of same network size. As indicated by Figure 4-5 the total number of push operations and hence the running time of the proposed algorithm is small when the number of requests in the system is smaller compared to the total capacity of the system. With increasing requests running time of the algorithm increases because the system capacity becomes insufficient and finding an optimal allocation requires more and more pushback operations. However the running time becomes constant when the

total number of push operations reaches its maximum and after that point the running time does not increase with increasing number of requests.

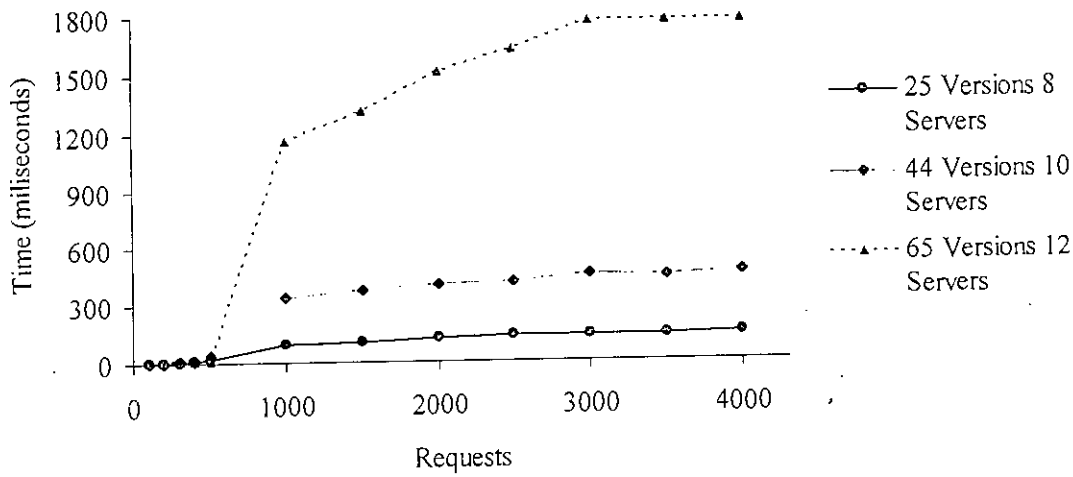


Figure 4-6 Running times of the proposed algorithm for three different network sizes.

With the increasing height of the source, total number of push operations increases and the proposed network flow heuristic algorithm explores more options before returning the excess flow of any version to the source. Thus the algorithm provides better satisfaction values if the source height is increased. However increasing number of push operations also increases the running time of the algorithm. So, a trade off between the performance and the running time of the algorithm becomes necessary. Figure 4-7 and Figure 4-8 show how the performance and the running time of the algorithm varies with increasing source height. For each randomly generated data set the algorithm is run for four different source heights: $|V|$, $2|V|$, $3|V|$ and $4|V|$, where $|V|$ is the total number of vertices in the flow network. Figure 4-7 shows the percentage increase in Total Satisfaction for source height $2|V|$, $3|V|$ and $4|V|$ with respect to the Total Satisfaction obtained for source height $|V|$. If source height is increased from $2|V|$ to $3|V|$, Total Satisfaction increases by 0.6% on the average. On the other hand, time required by the algorithm increases by almost 50% for the same increase in source height. In the proposed algorithm the source height is maintained at $2|V|$, this is

because the proposed model deals with real time services and so more emphasis has been given for reducing the running time of the proposed heuristic algorithm.

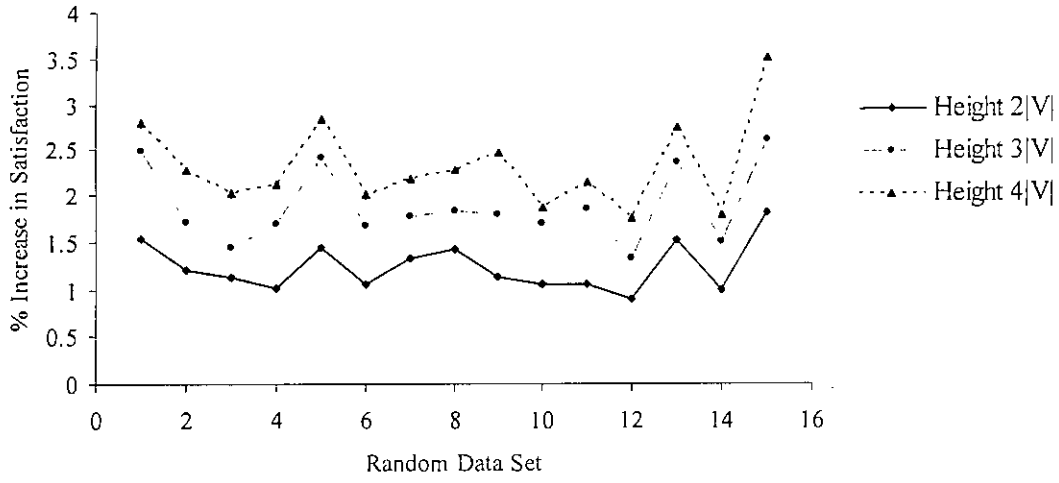


Figure 4-7 Percentage increase in Total Satisfaction for three different source heights with respect to source height $|V|$.

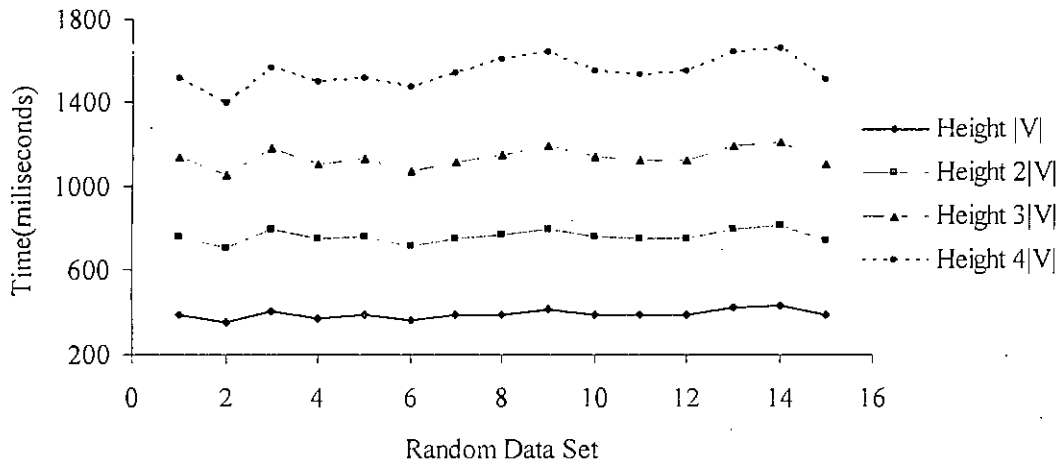


Figure 4-8 Running times of the proposed algorithm for the four different source heights.

4.5. Chapter Summary

In this chapter the proposed network flow heuristic algorithm has been compared with a Brute Force algorithm and a Greedy algorithm. The results show that the proposed algorithm provides near optimal solutions when compared with the Brute Force algorithm and also provides far better solutions than the Greedy algorithm. It is also clear from the result analysis that the running time of the proposed network flow heuristic algorithm does not depend on the number of requests since after a certain level the running time becomes almost constant for a given network size. The following chapter concludes this thesis and also presents some suggestions to further improve this research.

Chapter 5. Conclusion

This chapter starts with describing some major contributions of this thesis and finally presents some options for future research in this area.

5.1. Major Contribution

The previous reliability optimization models aim at the optimal "*integration*" of components in developing a single component-based software application. In this thesis a new optimization model is presented that aim at the optimal "*allocation*" of web service components to multiple clients in a distributed web service system. In the optimal allocation, not only the total satisfaction of all clients will be maximized, but also the number of clients admitted to the system will be maximized. This will happen automatically, since the allocation strategy implemented here assigns at least one version of a specific component to each client before assigning a redundant version to any client already admitted to the system.

Also, this thesis explores the idea of applying network flow models to optimization problems, which has not been done previously. The network flow approach presented here not only finds the optimal solution in polynomial time, but also is independent of the number of requests, the varying parameter of the system. The worst case complexity depends only on the total number of versions of all components and on the total number of servers in the system. These parameters define the size of the network flow graph and are fixed and known previously. Since running time increases heavily with increasing network sizes, the proposed network flow heuristic algorithm is best suitable for a small number of web service components and a small number of servers with huge capacity to accept a large number of requests.

Since the algorithm proposed in this thesis solves the resource allocation problem in a distributed system, it can be implemented in any system that incorporates distributed resources, for example, a server firm. The model on which the proposed algorithm is built perfectly fits today's web services architecture. Nowadays all applications are

moving towards the zero-installation web services based approach. Besides regular web applications like webmail and search engines, there are now web services based applications such as office suites (Google Docs and Spreadsheets), online anti-virus checks and custom information systems (Google Earth, Google Maps). To ensure the best possible service with limited resources, the resource allocation must be optimized for maximum gain. This is exactly what the proposed algorithm is designed to solve.

5.2. Future Works

Pushing the excess from a server to the sink means the allocation of a set of versions from that server. To give higher priority to versions that have higher satisfaction values and lower resource requirements, the adjacent versions of a server are sorted in decreasing order by their (satisfaction/aggregate resource requirement) values. This Greedy Knapsack strategy does not always guarantee the optimal allocation from a server. Some research work can be directed in this area to find the optimal set of versions for each Server $\rightarrow$ Sink push. The same can be applied for the backward pushes as well. Though the complexity of the proposed algorithm does not depend on the number of requests, it depends heavily on the total number of web service components along with their alternative versions. Reducing the complexity even more requires further research in this area. The network flow approach can also be used for existing reliability optimization models and its performance can be compared with that of the existing reliability optimization algorithms.

References

[1] Gottschalk, K., Graham, S., Kreger, H. and Snell, J., "Introduction to web services architecture", IBM Systems Journal, Vol-41, No. 2, 2002.

[2] Kreger, H., "Fulfilling the web services promise", Communications of the ACM, Vol-46, No. 6, pp 29-34, June 2003.

[3] Tanenbaum, A. S. and Steen, M., "Distributed Systems", pp 362-364, 2002.

[4] Xie, M., "Software reliability modelling", World Scientific Publishing Co., Singapore, 1991.

[5] Mura, J. D., "A Theory of software reliability and its applications", IEEE Transaction on Software Engineering, SE-1(3), 1975.

[6] Zo, H., Nazareth, D. L. and Jain, H. K., "Measuring reliability of applications composed of web services", Proceedings of the 40th Annual Hawaii International Conference on System Sciences, pp 278-287, 2007.

[7] Zo, H., "Supporting intra- and inter-organizational business processes with web services", Ph.D. Thesis, the University of Wisconsin-Milwaukee, August 2006.

[8] Chen, L. and Avizienis, A., "N-Version Programming: a fault tolerance approach to reliability of software operation", IEEE Proceedings of FTCS-25, Vol-3, pp 113-119, 1996.

[9] Belli, F., and Jedrzejowicz, P., "Fault tolerant programs and their reliability", IEEE Transactions on Reliability, Vol-39, No. 2, pp 184–192, 1990.

[10] Randell, B., "System structure for software fault tolerance", IEEE Transaction on Software Engineering, Vol-SE-13, pp 582-592, 1987.

[11] Berman, O. and Ashrafi, N., "Optimization models for reliability of modular software systems", IEEE Transaction on Software Engineering, Vol-19, No. 11, pp 1119-1123, 1993.

[12] Wattanapongsakorn, N. and Levitan, S., "Reliability optimization models for fault tolerant distributed systems", In Proceedings of Annual Reliability and Maintainability Symposium, pp 193-199, 2001.

[13] Jung, H. and Choi, B., "Optimization models for quality and cost of modular software systems", European Journal of Operational Research, Vol-112, No. 3, pp 613–619, 1999.

[14] Belli, F. and Jedrzejowicz, P., "An approach to the reliability optimization of software with redundancy", IEEE Transactions on Software Engineering, Vol-17, No. 3, pp 310–312, 1999.

[15] Caserta, M., and Uribe, A. M., "Tabu search-based metaheuristic algorithm for software system reliability problems", submitted to Computers and Operations Research, July 2007.

[16] Caserta, M. and Ryoo, H. S., "Efficient tabu search-based procedure for optimal redundancy allocation in complex system reliability". In Proceedings of the 5th International Conference on Optimization: Techniques and Applications, Vol-2, pp 592–599, 2001.

[17] Chang, W. C., Wu, C. S. and Chang, C., "Optimizing dynamic web service component composition", Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence, 2005.

[18] Tsai, W. T., Zhang, D., Chen, Y., Huang, H., Paul, R. and Liao, N., "A software reliability model for web services", the 8th IASTED International Conference on Software Engineering and Applications, pp 144–149, 2004.

[19] Kuo, W. and Prasad, V. R., "An annotated overview of system-reliability optimization", IEEE Transactions on Reliability, Vol-49, No. 2, pp 176–187, 2000.

[20] Kim, J. H. and Yum, B. J., "A heuristic method for solving redundancy optimization problems in complex systems", IEEE Transactions on Reliability, Vol-42, No. 4, pp. 572-578. 1993.

[21] Painton, L. and Campbell, J., "Genetic algorithms in optimization of system reliability", IEEE Transactions on Reliability, Vol-44, Issue. 2, pp 172-178, 1995.

[22] Marseguerra, M. and Zio, E., "System design optimization by genetic algorithms", In Proceedings of Annual Reliability and Maintainability Symposium, pp 222-227, 2000.

[23] Glover, F., "Tabu search – part 1", ORSA Journal on Computing, Vol-1, No. 3, pp 190-206, 1989.

[24] Glover, F., "Tabu search and adaptive memory programming-advances, applications and challenges", Interfaces in Computer Science and Operations Research, 1996.

[25] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C., "Introduction to algorithms, 2nd edition", pp 651–668, 2001.

[26] West, D. B., "Introduction to graph theory, 2nd edition", pp 183–187, 2001.

[27] Ford, L. R. and Fulkerson, D. R., "Maximal flow through a network", Can. J. Math, Vol-8, pp 399-404, 1956.

[28] Edmonds, J. and Karp, R. M., "Theoritical improvements in algorithmic efficiency for network flow problems", J. Assoc. Comput. Mach., Vol-19, pp 248-264, 1972.

[29] Goldberg, A. V., Tardos, E. and Tarjan, R. E., "Network flow algorithms", Algorithms and Combinatorics, Vol-9, pp 100-164, 1990.

[30] Goldberg, A. V., "A new max flow algorithm", Technical Report MIT/LCS/TM-291, Laboratory for computer science, M.I.T., 1985.

[31] Goldberg, A. V., Tarjan, R. E., "A new approach to the maximum flow problem", Proceedings of the 18th ACM STOC, pp 136-146, 1986.