`ABSTRACT

GANDLUR, KARTHIK SATYANARAYANA. Implementation of Adaptive Routing in Public Logistics Networks. (Under the direction of Michael G. Kay.)

A public logistics network is proposed as a means to extend many of the features associated with public warehouses to the entire supply chain. In addition to providing traditional warehousing and storage functions for hire, a public logistics network would make it possible to negotiate with multiple firms on a load-by-load basis in order to determine the most efficient means of providing the resources needed to complete each stage of a load's transit through the network. The report gives an overview of the supporting technologies that make Public Logistics Network a reality. It is intended to provide a preliminary outlook at the various issues related to implementation and could be used for future research as a basis for building the infrastructure required for the new model. An effort has been made to provide an overview of the Web Services, which will be a logical extension to the implementation of such a network. In addition to that, this report will describe the use of Adaptive Routing Protocol using Java Programming for building a hypothetical Public Logistics Network covering the southeastern United States. The network consists of 36 nodes representing the public distribution centers and 59 routes, which represents various interstate highways connecting them. The model helps identify the different scenarios that may occur, including in-transit trade and cost variations. The scenario representations of the model will be used as a benchmark for future research and development associated with building an actual negotiating agent model.

IMPLEMENTATION OF ADAPTIVE ROUTING IN PUBLIC LOGISTICS NETWORKS

KARTHIK SATYANARAYANA GANDLUR

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

INDUSTRIAL ENGINEERING

Raleigh

2002

APPROVED BY:

To

Amma, Appa & Murali

BIOGRAPHY

Karthik Satyanarayana Gandlur was born in Bangalore, India in 1977. He received his Bachelor's degree in Industrial Engineering from the Bangalore University in 1999. He joined Indian Institute of Science (IISc) as a Research Assistant. Later he worked at Maini Precision Products, Bangalore as an Industrial Engineer. He then joined the graduate program in the Industrial Engineering department at North Carolina State University in the fall of 2000 and is currently working towards the completion of a M.S degree in Industrial Engineering.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my graduate advisor, Dr. Michael Kay for his guidance, support, and for being my mentor. Working with him has been a highly rewarding experience.

I would like to thank Dr. Robert Young and Dr. Robert Handfield for showing interest in my research and agreeing to be on my committee.

Special thanks to my friend Balaji Gopinathan, for his patience, guidance and support, in helping me update my programming skills and using the same in my thesis.

I would like to thank all the members of the WISDEM (Industrial Engineering) group for their support and encouragement. Thanks to all at PS 209 for putting up with my antics for last two years and making my stay at North Carolina State University very enjoyable.

Most of all I would like to thank my family their love and support, and for standing by me when I needed them most. This work is dedicated to my parents for motivating me to come so far and for teaching me to be successful in life.

Contents

List of Figures

List of Tables

Chapter 1: Introduction

1.1 Introduction

Experts predict the approaching e-business era will require massive changes in global supply chain operations, the activities in any organization that make and deliver goods and services to customers. Greatly increased customer expectations, coupled with enormous opportunities for performance improvement, will compel this redesign [1].

The redesign will employ reengineered processes, optimizing systems, and channel-linked organizations to produce improvements in order cycle time, total delivered cost, order fill rates, and on-time performance that are multiple times better than today. Crafting today's supply chains into the advanced e-chains of the future will not be easy. It will require a rapid evolution of supply chain competence from functional focus through enterprise integration to channel collaboration to virtual connectivity.

The trading environments offered by electronic commerce (eCommerce) will be radically different from the traditional market models in existence today. Users in this new rapidly changing market will require ever more sophisticated services to assist them. Buyers will need to identify and purchase the best from the myriad of products and services available on the Internet; suppliers will want to target potential customers in a more efficient manner, increasing revenue and reducing costs. Software too needs to keep up with rapid change and increasing complexity. The eCommerce market will not tolerate slow response or delayed decision-making. New ways of developing channels for interactive relationships with customers and building customer communities must be built [2]. It is this business model which is forcing businesses to take note of software agents.

A software agent is a software entity that can act in an autonomous manner, can learn, is proactive, and has the ability to interact with other entities, be they human or software based. Software agents are predicted to play an increasing role in electronic commerce as they exhibit much of the behavior considered to be desirable or even essential. Agents endowed with these capabilities can significantly reduce the amount of human effort required, therefore providing real value [3].

Logistics services for e-commerce have proved to be an area that requires major improvements if e-commerce is to achieve its full potential. The failure of many e-businesses, for example in the United States, to fulfill orders during peak demand periods and the reluctance of some sellers to engage in international e-commerce because of the complex logistics requirements clearly demonstrate the critical role of e-logistics [4].

The existing e-logistics problems arise largely from the fact that e-commerce and the demand for related logistics services have grown at a much faster rate than that at which suitable logistics services and solutions have been developed. Traders have responded to the increased demand for logistics services that arises from e-commerce by adopting a variety of methods. These include handling of order fulfillment by companies themselves using in-house logistics services, outsourcing fulfillment to third-party logistics service providers (3PLs), drop shipping, and various combinations of these methods. Concurrently, considerable efforts have been made to develop software applications in order to automate logistics functions such as order management, cargo and equipment tracking, transportation management and planning, customer service management, and returns management. Technology plays a critical role in providing systems that can enhance the ability of logistics service providers to satisfy customer demands. The main weakness of the efforts to develop applications for improving logistics is the general lack of integration between the various applications used for different logistics functions. Many of the applications are designed to handle different types of logistics functions, and this tends to lead to the existence of incompatible systems being applied to related logistics functions.

To achieve more efficient e-logistics and e-fulfillment, it is desirable to have a trading environment in which there is sufficient information about goods as regards their description and origins, and destinations. Sellers and buyers should be able to monitor and track goods at every point along the way from the supplier to consumer. All stakeholders should be able to check on the Internet the availability and status of orders. All this can be achieved if trade information is simplified automated and fully harmonized. It also requires sophisticated supply chain management systems for compiling and enabling global end-to-end monitoring of trade information.

To accomplish these broad objectives we have to take advantage of the great potential provided by the Internet technology in order to capture, transfer and monitor trade information over global networks of supply chains in an open fashion. Also we have to promote greater integration of software applications for logistics functions, including the use of such systems as Web Services, XML (extensible Markup Language), SOAP (Simple Object Access Protocol), and UDDI (Universal Description, Discovery and Integration).

Yantra, a leading provider of enterprise software for distributed commerce management, describes in their white paper [5] the impact of the Internet on supply chain logistics providers. The paper describes the changing attitude of clients of customers and that real time sharing of information has become the "norm." Clients expect their logistics partners to provide them and their customers with accurate order and inventory status. This expectation places significant strain on logistics providers that often utilize fulfillment systems not designed to work in a real-time environment. Most logistics providers lack a single system that provides real-time status of orders and inventory across their entire fulfillment network. A study published in *Logistics Information Management* [6] predicted that the number of private warehouses will decrease in future and the number of transshipment warehouses will increase. This shows that the idea of public logistics networks goes along with the recent trends in the logistics industry.

1.2 Public Logistics Networks

A "public logistics network" is proposed as a means to extend many of the features associated with public warehouses to the intelligent supply chain. In addition to providing traditional warehousing and storage functions for hire, a public logistics network would make it possible to negotiate with multiple firms on a load-by-load basis in order to determine the most efficient means of providing the resources needed to complete each stage of a load's transit through the network. Items could continuously negotiate with the logistics resources of the network using simultaneous auctions in order to determine the best route and cost and schedule. Similar to the dynamic pricing used to sell airline seats, a price for each available space on a truck and storage space at a distribution center (DC) could be negotiated in real time for each individual item [4].

A unique capability of such a network is that a third party can search the network for any type of item in transit. Once located, negotiations can take place and the item might be resold to the third party and redirected to a new destination. The potential utility of this search and negotiate capability depends on the characteristics of items being transported: it is not likely to be needed to locate low-cost, ubiquitous items like toothbrushes because they can be expected to be available at every local store; nor is it needed to locate custom-made, one-of-a-kind products because there are so few of the items available, of uncertain quality, that the use of traditional private logistics networks is likely to be the most efficient. A public logistics network is likely to be most suitable for managing the multitude of commodity-like items (replacement parts, etc.) that fall in the middle ground between ubiquitousness and uniqueness.

The idea of Public Logistics Network in comparison with private, single owned monopolies like FedEx and UPS can be summarized very well below [4]:

> Currently, it is common for a single logistics firm to handle a load throughout its transport. Although companies like FedEx and UPS have very sophisticated proprietary tracking and control infrastructures, the control of the logistics network is highly centralized. The most notable feature of these private logistics networks is that a single firm controls the network and much of the technology used to coordinate the operation of the network is proprietary. As a result, the principal competitive advantage that a private logistics company has is the barrier to entry due to the very large scale of operation (national or international) required in order to be able to underwrite the development of private facilities and propriety technologies. Nevertheless, a single firm, unless it becomes a monopoly, is ultimately limited in the scale of its operation, resulting in the use of single-firm "hub" DCs. With a limited number of large-scale hub DCs, a load can make many circuitous "hops" before it reaches its destination.

The most salient impact of such a network is likely to be that it would make it possible to separate the different functions of the network so that a single firm is not required for coordination. This would enable economies of scale to be realized in performing each logistics function since each element of the network has access to potentially all of the network's demand. The increase in scale might make it economical to ship in full truckloads throughout the network as opposed to more costly less-than-truckload shipments. This could be possible because a single truck could be used to transport all of the demand associated with a lane (or link) in the network. Many of the long-haul, single-product full-truckload shipments between private facilities can be replaced by sequences of short-distance hops between public DCs. Links in the network could be served by trucks that are owned and operated by different firms, and each transshipment point (i.e., public DC) in the network could be an independently operated facility. Due to the increase in scale, it would be economical to have many more DCs. Public DCs (which would operate like existing private highly automated hub DCs, but on a smaller scale) could be established in small cities and towns that would never have such facilities if they were served as part of a proprietary, private logistics network.

1.3 A Snapshot

This section presents a brief snapshot of the possible scenario to help better understand the proposed system.

Every manufactured item will have an RFID attached to it. The manufacturer generates a GUID and an item specific XML code derived from the Document Type

Definition (DTD) [7] relevant to the UNSPSC code of the class of products associated with the part. The RFID will be programmed with the GUID given to the item. The XML data will be stored in the PackageIP Server at the manufacturer's premises. If an order has been placed for the item, it will be shipped to the customer; otherwise, it will be stored in the warehouse. PackageIP servers at the warehouse/truck will update inventory data (update XML files). The item is shipped across the public logistics network to the customer [8].

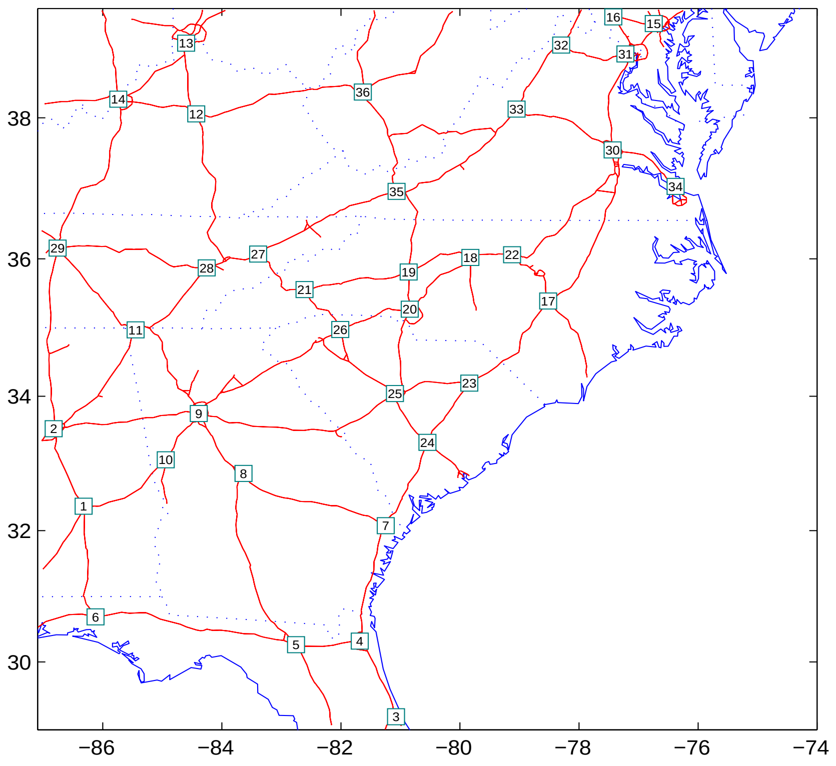


Figure 1.1: Hypothetical public logistics network showing 36 public DCs covering the southeastern portion of the USA and connected via interstate highways [9].

Now let us suppose that a customer, who owns a 1965 Ford Mustang, has a broken transmission and wants to buy a replacement for the critical component. He shops around on the Internet and searches for the part by using the UNSPSC number of the part got from the website. Placing an order with the manufacturer has a lead-time of two months before he can get the replacement. But he finds a similar part was shipped the previous day from Jacksonville, FL and is on a truck near Benson, NC heading north on I-95, to be delivered in Richmond, VA. He also notices from his query that the part is available for in-transit trade. He starts the negotiation immediately with the part's agent running on a computer onboard the truck, by giving his intent to buy and his offer terms. After an agent-mediated negotiation process, which in case is successful, i.e., both the buyer and the seller (part) agree on the trade terms, the part is immediately unloaded at the closest DC and loaded onto to a truck heading towards the destination. All the necessary information about the part can be stored in the "smart tag" attached to the item. The information will be stored in a standard XML format that can be queried and accessed easily by any commonly used browser. When the truck transporting the car on nears the I-85–I-40 interchange, the item would be unloaded at the interchange's public DC (DC 15 in Figure 1.1) and loaded onto the next truck heading to Raleigh along I-40. Within two hours, the part could be in Raleigh as compared to an earlier wait period of over two months. At the same time, the customer at Richmond could locate an identical replacement part and, with the revenue from the sale of the part to the Raleigh customer minus the cost of the replacement, may be able to realize a net gain.

Currently, without a public logistics network, the customer in Raleigh would most likely have to either shop around with local dealers or wait till he receives the part from the manufacturer. Information about any item can be tracked effortlessly using this system, which facilitates seamless information flow throughout the supply chain. As the product finds its way to the customer, it reduces overheads from other components and parties of the supply chain. As all negotiations and settlements are done automatically, the system will work with better efficiency and flexibility.

Intelligent agents representing each package will negotiate with agents representing each manufacturer, customer, truck, and distribution center in the logistics network in order to minimize its individual transport cost. This makes it possible to focus on the pure transport-related arbitrage opportunities that a public logistics network can provide. In particular, it will be determined whether, in equilibrium, the logistics network does operate in a least cost manner and, most importantly, whether the network can re-optimize through self organization after being subject to a variety of disturbances, ranging from the simple breakdown of a truck to the logistical challenges associated with a major natural disaster (e.g., a hurricane) [9].

1.4 Implementation and Coordination of PLN

This research will focus on implementation of software agents, and how they are helpful in the management of issues that are critical to the coordination of a public logistics network. Also it further looks at different ways of this can be implement over the Internet, using the Web Services.

The two specific issues that are critical to the coordination of a public logistics network are Adaptive Routing and In-transit trade [9].

- **Adaptive routing:** When a package is to be transported through the network, intelligent software agents would be spawned and sent ahead to each DC along the route to the package's destination. Based on the local price of transport and storage at each DC, the "best" route for the package is determined. All of the package agents that are at DCs not along the intended route would remain active until the package reaches its destination in order to be available to determine alternative routes for the package in case of any disruption (cost increase) along its intended route or a significant cost decrease along an alternative route.

- **In-transit trade:** All of a package's agents that are located at the DCs along both its intended and alternative routes are available as trading agents for the possible resale of the package and its subsequent re-routing to a new destination. Triggered updates will provide each trading agent with the minimum cost required to transport the package from its current location to the DC where the trading agent is located; as a result, the exact, accurate transport cost can be added to the FOB (free-on-board) price of the package from its origin, making it possible to instantaneously negotiate using price quotes that represent the delivered price at the DC where the trading agent is located.

These issues can be managed successfully by use of software agents. An Agent is a software entity that has a knowledge base, an inference mechanism and an explicit model of the problem to solve – including a model of other agents with which it must interact. Agent technologies are well suited for applications having the following special characteristics: modular, decentralized, changeable, ill structured, and complex. The tasks of design, simulation, and scheduling and control manifest many of these characteristics, particularly under the impact of modern manufacturing trends listed above, suggesting that agents are a natural way to address them.

Agents are pro-active objects, and share the benefits of modularity that have led to the widespread adoption of object-oriented methods, and are thus best suited to applications that fall into natural modules. They monitor their own environment and take action, as it deems appropriate. This characteristic of agents makes them particularly suited for applications that can be decomposed into stand alone processes, each capable of doing useful things without continuous direction by some other process. Public logistics network involves items distributed over a wide geographic region, and decentralized control via an agent-based architecture is an ideal fit to such an organizational strategy.

Modularity and independence combine to make agents especially valuable when a problem is likely to change frequently. This type of complex dynamic behavior is a perfect fit to our model, where items are routed dynamically over the supply network. Transportation and storage prices vary with time and availability. Items are to be re-routed to different customers when demand arises. Space, cost, and time are to be continuously negotiated for as the item makes its way through the distribution network. For these reasons, the use of intelligent agent systems would be the best approach to tackle this complex system.

Web Services act as a nice complement to the use of these agents over the network. Web services, in the general meaning of the term, are services offered via the Web. In a typical Web services scenario, a business application sends a request to a service at a given URL using the SOAP protocol over HTTP. The service receives the request, processes it, and returns a response. Since all these services make use of software programs and protocols, software agents can be built using the object-oriented programming languages, which are compliant with these standards and further the use of them over the Internet.

Web services depend on the ability of enterprises using different computing platforms to communicate with each other. This requirement makes the Java platform, which makes code portable, the natural choice for developing Web services. This choice is even more attractive as the new Java APIs for XML become available, making it easier and easier to use XML from the Java programming language.

CHAPTER 2: ROUTING

2.1 Introduction

Routing is the process of finding a path from a source to every destination in the network. It allows the users in the remotest part of the world to get information and services provided by computers anywhere in the world [10].

Routing is a term usually associated with the main process used by Internet hosts to deliver packets. Internet uses a hop-by-hop routing model, which means that each host or router that handles a packet examines the Destination Address in the IP header, computes the *next hop* that will bring the packet one step closer to its destination, and delivers the packet to the next hop, where the process is repeated. To make this work, two things are needed. First, *routing tables* match destination addresses with next hops. Second, *routing protocols* determine the contents of these tables.

This is similar to the various routing strategies that are associated with the movement of Automated Guided Vehicles (AGV). The aim of routing AGVs is to find an optimal and feasible route for every single AGV. The routing decision includes three aspects. Firstly, it should detect whether there *exists* a route, which could lead the vehicle from its origin to the destination. Secondly, the route selected for an AGV must be feasible. Thirdly, the route must be optimal. This is very similar to what we are trying to achieve over the public logistics network. The package in a PLN can be related to an AGV and the movement of the package in the network is similar to the routing strategy employed for an AGV.

The evaluation of alternative routes and the actual routing of an AGV based on this evaluation can be either static or dynamic. When routing is *static*, the path taken by an AGV between any two given nodes is always the same—that is, the route does not vary over time as a function of the current congestion in the system. The most natural solution is always to select the path with the shortest travel distance. When routing is *dynamic*, different paths can be taken by an AGV at different times when moving between two given nodes. Taking into consideration the current status of the system, the vehicle router selects a path for the AGV at the time that the vehicle is dispatched and if there is a communications link between the router and the vehicle, then the router modifies the vehicle's path during travel [11].

We can compare this scenario with the routing strategy used by a package moving in a public logistics network, with constant and variable route costs. If the costs do not change, for any of the routes during the period the package leaves the origin and reaches the destination, the routing can be considered to be static. But if the route costs keep changing over time as a function, and the objective of the package is to move between the origin and the destination with least transportation cost, the route taken by it constantly keeps changing from its intended journey path.

Routing is accomplished by means of *routing protocols* that establish mutually consistent routing tables in every router in the network. A routing table contains at least two columns: the first is the address a destination endpoint or a destination network, and the second is the address of the network element that is the next hop in the "best path" to this destination. When a package arrives at a router, the router consults the routing table to decide the next hop for the packet. Any routing protocol must communicate global topological information to each routing element to allow it to make the local routing decisions. Yet global information, by its very nature, is hard to collect, subject to frequent change and voluminous. A routing protocol asynchronously updates routing tables at every router.

2.2 Comparison of Routing Protocols

The various choices that we have in using the different types of protocols can be classified based on the type of control we desire to have, nature of hop, and state of the system. In centralized routing, a central processor collects information about the status of each link and processes this information to compute a routing table for every node. It then distributes these tables to all the routers. In distributed routing, routers cooperate using a distributed routing protocol to create mutually consistent routing tables. Centralized routing is reasonable when the network is centrally administered and the network is not too large. However it suffers from the same problems as a centralized name server: creating a single point of failure, and the concentration of routing traffic to a single point.

A package agent can carry the entire route (that is, the addresses of every router on the path from the source to destination), or the agent can carry just the destination address, and each router along the path can choose the next hop. These alternatives represent extremes in the degree to which a source can influence the path of a packet. A source route allows a sender to specify a package's path precisely, but requires the source to be aware of the entire network topology. If a link or a router along the path goes down, a source-routed package will not reach the destination. Also if the path is long, the information contained by the agent can be fairly large.

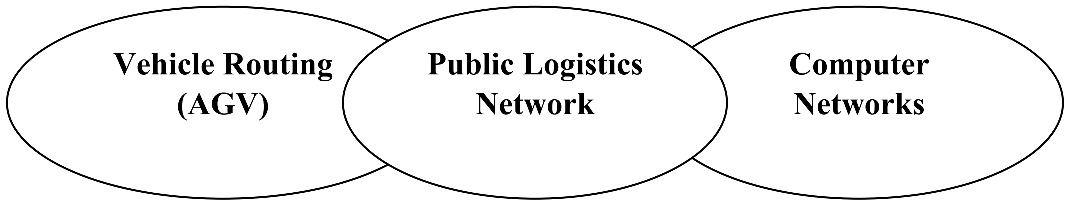


Figure 2.1: Comparison of Routing in Networks

The routing protocol in a public logistics network can shown to share some characteristics of the routing which occurs in vehicle routing (AGV) and computer networks, but also have some major differences. It is an intermediate between the vehicle routing and the way routing occurs in a computer network.

Some of similarities and the differences can be listed, as shown in the table below.

Table 2.1: Comparison of VR, PLN and CN

Vehicle Routing	**Public Logistics Networks**	**Computer Networks**
Network structure is assumed to be 100% reliable and accurate.	Network structure is assumed to be 100% reliable and accurate.	Networks may or may not be 100% reliable and accurate.
Centralized routing, where a central processor, which computes all the routing tables, collects all the information.	Distributed routing, routers cooperate by using a protocol which creates mutually consistent routing tables.	Distributed routing, routers cooperate by using a protocol which creates mutually consistent routing tables.
Usually state independent, where the route taken is not dependent on the current network state, but can also have dynamic routing.	State dependent, where dynamic routing occurs and the choice of a route depend on the current network state.	Both state dependent and state independent depending on the type of protocol used.
Network is small, usually confined to shop floor of a manufacturing facility.	Network is large, e.g. South eastern portion of USA as shown in Figure 1.1	Network can be of any size and scalable.
Usually a single central processor controls all the AGVs.	Usually one router is associated with one package or a small number of packages.	Every router handles millions of packets that are routed through it.

Comparing the routings above, we notice that routing in public logistics networks shares characteristics of both vehicle routing and routing in computer networks, but also is different from both. We can say following are the characteristics that would be ideal for a routing protocol to be implemented in a public logistics network.

- Entire network topology is assumed to be reliable, robust and known to all the entities in the network. This assumption is based on the fact that a logistics network does not dynamically change over short intervals of time and DCs are not subject to frequent shut downs or change in location.
- Every node (DC) knows the address of all its neighbors and also the cost of reaching there.
- It follows a distributed routing and is hop-by-hop, where each DC may act as a router. Here the package knows the destination address and each router (DC) along the path will choose the next hop.
- It follows a multiple path and state-dependent way of computing the optimal path. This means that it is sensitive to the dynamic changes in the route costs and can still find a path to the destination even if a few routes along its journey are shut down or have an exhorbitantly high cost of transportation.

Based on the assumptions and the desired characteristics for a public logistics network, distance vector algorithm (also known as Bellman-Ford Algorithm) was found to be the most suitable. It allows a router (DC) to find routing information, that is, the next hop to reach every destination in the network by the shortest path, by exchanging routing information with only its neighbors (neighboring DCs). Roughly speaking, in a distance-vector algorithm, a node tells its neighbors its distance to every other node in the network. Thus it is distributed and is suitable for large inter-networks controlled by multiple administrative entities. It follows a multiple path and state-dependent way of computing the optimal path.

2.3 Distance-vector Routing

In distance-vector routing we assume that each router knows the identity of every other router in the network but not necessarily the shortest path to it. Each router maintains a distance-vector, that is, a list of (destination, cost) tuples, one tuple per destination, where cost is the current estimate for the sum of the link costs on the shortest path to that destination. Each router initializes the cost to reach all non-neighbor nodes to a value higher than the expected cost of any route in the network, commonly referred as infinity [10].

A router periodically sends a copy of its distance vector to all its neighbors. When a router receives a distance vector from a neighbor, it determines whether its cost to reach any destination would decrease if it routed packets to that destination through that neighbor. It can easily do so by comparing its current cost reach a destination with the sum of the cost to reach its neighbor and its neighbor's cost to reach that destination. If the nodes asynchronously update their distance vectors, the routing tables will eventually converge. The intuition behind the proof is that each router knows the true cost to its neighbors. This information is spread one hop with the first exchange of distance vectors, and one hop further on each subsequent exchange. With continued exchange of distance vectors, the cost of every link is eventually known throughout the network. The distance vector algorithm is also known as the Bellman-Ford algorithm after its creators.

The interval between exchanges of the distance vectors represents a trade-off between sensitivity to link finding the best path and the overhead in exchanging and processing routing information. The longer the update interval, the longer it takes for a router to detect a failed peer. Thus, choosing a good update interval is nontrivial.

RIP (Routing Information Protocol) suggests that vectors be exchanged once every 30 seconds, and if a router does not hear from its neighbor for six consecutive intervals, the neighbor is assumed to be down. Thus, a node can be down for three minutes before anyone notices. This information will then take its time to be propagated throughout the network. However, if the routing protocol triggered updates, convergence after a failure can be made much more rapidly. Thus, a good compromise may be to use a slow update interval (around 30 seconds) in the common case, but to trigger rapid distribution of distance vectors in the case of link or router failure.

The Figure 2.1 below represents the Distance-Vector algorithm at node A. Node A receives a distance vector from its neighbor B. It uses this information to find that it can reach nodes C and D at a lower cost. It therefore updates its own distance vector and chooses B as its next hop to nodes C and D.

In the Figure 2.1, if router A has an initial distance vector of ({A, 0}, {B, 1}, {C, 4}, {D, ∞}), we see that the arrival of a distance vector from B results in updating its costs to C and D. If a neighbor's distance vector results in a decrease in a cost to a destination, that neighbor is chosen to be next hop to get to that destination, for example, the distance vector from B reduced A's cost to D. Therefore, B is the next hop for packets destined for D. A router is expected to advertise its distance vector to all its neighbors every time it changes.

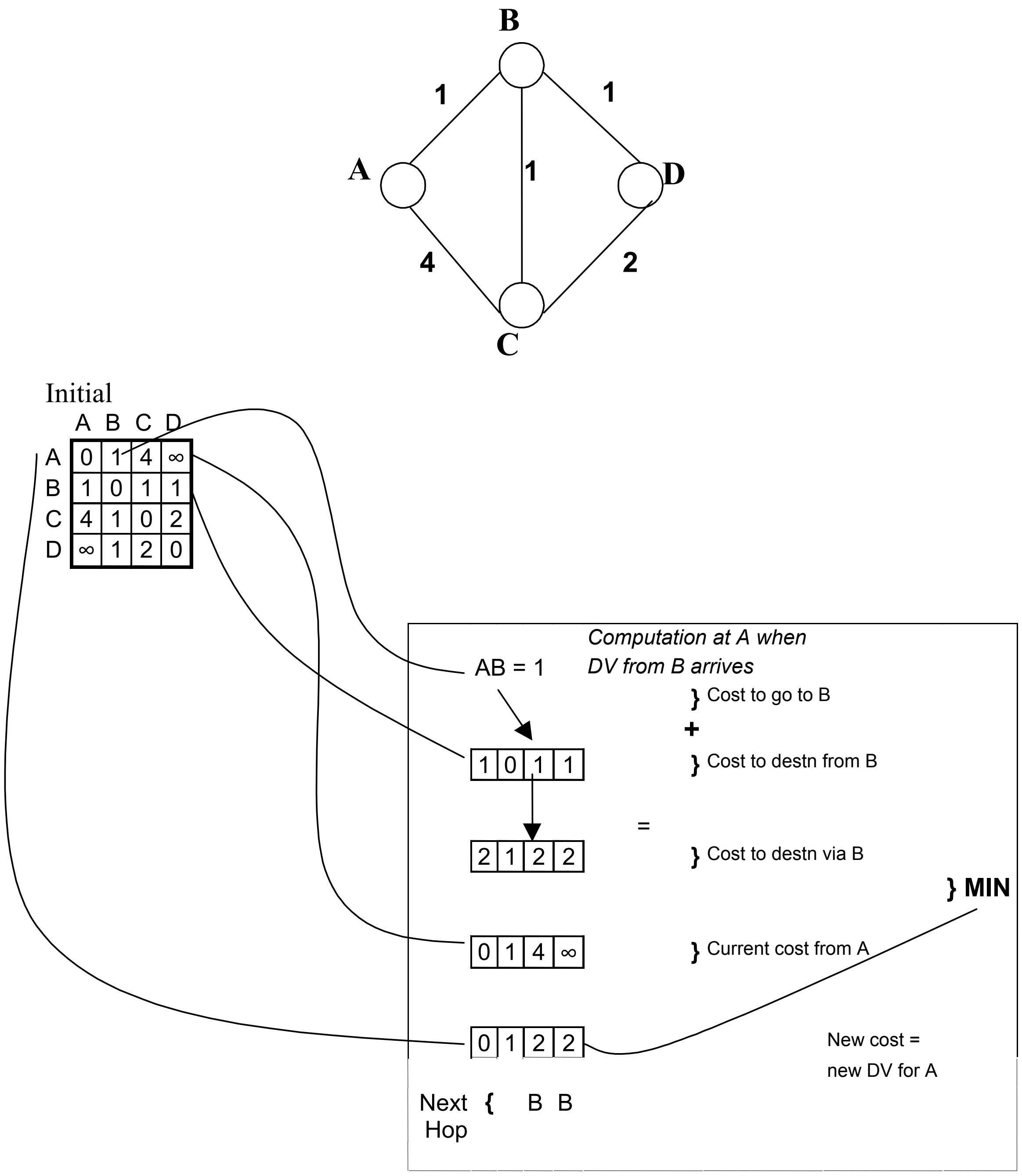


Figure 2.2: Example of Distance-Vector Algorithm

We can show that even if nodes asynchronously update their distance vectors, the routing tables will eventually converge. The intuition behind the proof is that each router knows the true cost to its neighbors. This information is spread one hop with the first exchange of distance vectors, and one hop further on each subsequent exchange. With the continued exchange of distance vectors, the cost of every link is eventually known throughout the network.

CHAPTER 3: AGENTS

3.1 Introduction

An agent is a system/computer program that can perform a task with a certain amount of “intelligence” due to some specialized skill (learning systems) or knowledge (knowledge bases). Consistent with the requirements of a particular problem, each agent might possess such intelligence to a greater or lesser degree [12].

An Agent can also be described as a software entity that has a knowledge base, an inference mechanism, and an explicit model of the problem to solve, including a model of other agents with which it must interact [13].

The modern manufacturing environment has taken a huge leap in the past two decades. The most important trends can be classified into increased product complexity, supply networks, and increased product variety over time.

Agent technologies are well suited for applications having the following special characteristics: modular, decentralized, changeable, ill structured, and complex. The tasks of design, simulation, and scheduling and control manifest many of these characteristics, particularly under the impact of modern manufacturing trends mentioned above, suggesting that agents are a natural way to address them.

Agents are well suited for use in applications that involve distributed computation or communication between components. Agent technology is well suited for use in applications that reason about the messages or objects received over a network. This explains why agent-based approaches are so popular in applications that utilize the

Internet. Multi-agent systems are also suited for applications that require distributed, concurrent processing capabilities.

An Agent can be autonomous. It monitors its own environment and takes action, as it deems appropriate. This characteristic of agents makes them particularly suited for applications that can be decomposed into stand alone processes, each capable of doing useful things without continuous direction by some other process. Public logistics networks are an expression of decentralized approach, and agent-based architectures are an ideal fit to such an organizational strategy.

Transportation and storage prices vary with time and availability. Items are to be re-routed to different customers when demand arises. Space, cost, and time are to be continuously negotiated for as the item makes its way through the distribution network. For these reasons, the use of intelligent agent systems would be the best approach to tackle this complex system.

3.2 Agent Based Negotiation Model

Intelligent agents representing each entity in the logistics network will negotiate with a common objective of transporting items from the manufacturer to the customer with minimum transportation cost. Trucks, distribution centers, and the factory constitute the set of resources in the logistics network. Other players in the network include the manufacturers, customers and the packages. The manufacturer can be represented by a single entity (the manufacturer agent will coordinate internally with the factory agent for resources required, production plans, forecasts, etc.). The model will consist of the following agents:

1. Package (P)
2. Manufacturer/Supplier (S)
3. Customer (C)
4. Truck (T)
5. Distribution Center (D)

The package agent will be the central control agent having multi-party negotiating capabilities. The distribution center agents will negotiate with transportation agents for obtaining the cheapest price for space in a truck. Keeping the transportation agents away from the mainstream item negotiation helps to keep the model simple and will reduce the load on package agents. Every agent will have an objective, to maximize the profit of its owner, potentially leading towards the maximization of profit for the whole supply chain.

The distribution center agent *D* is a shopbot-like agent, whose functions are very similar to that of any shop. It acts as a shopbot agent as it searches for the best deal available among the trucking agents. It becomes a pricebot agent as it puts up quotes for storage spaces, which changes dynamically over time. The factors determining the price of storage are (a) price quoted by competing DCs, (b) availability of storage over time, (c) demand for storage space, (d) cost of transportation as negotiated with the truck agents, and (e) other operational costs. The trucking agent *T* is a price-bot agent. It quotes the price for transporting items, which changes over time. Factors similar to those affecting the cost of storage at the DCs affect the price quoted by the truck agents. A brief snapshot of interactions between the various agents is shown in the Figure 3.1

Package agents carry information about its configuration, transportation data, price, owner, etc. These agents work for the current owner of the package associated with it. The main goal of the agent is to minimize the cost of ownership of the package. The ownership of the package changes as it flows through the supply chain. It depends on the terms of sale adopted by the trading parties (FOB/CIF). More information regarding the terms of sale can be found in the “Glossary of Shipping Terms” [14]. Therefore, the owner agent *O* can either be the manufacturer *F* or a customer *C*.

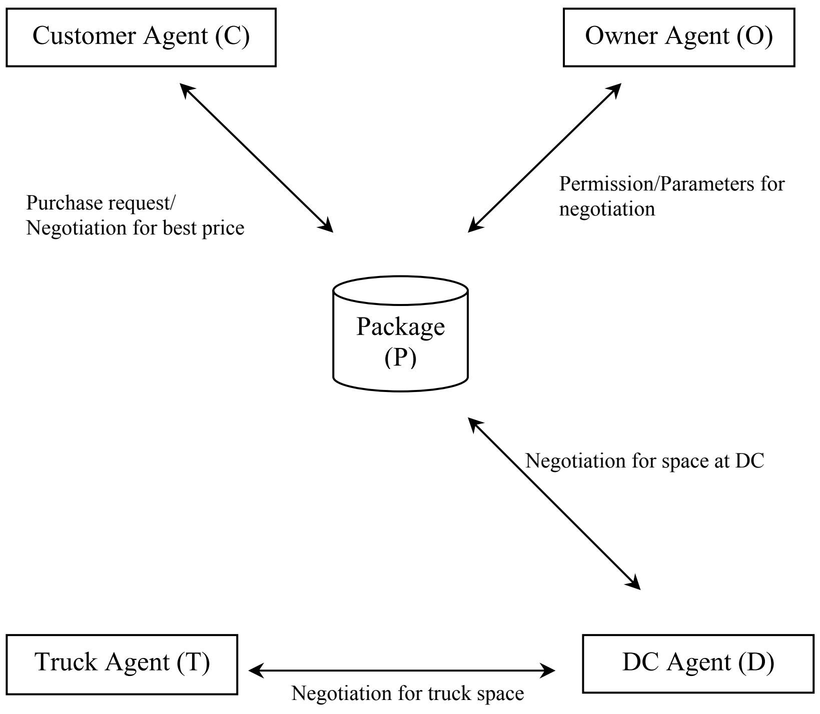

Customer Agent (C)
Owner Agent (O)
Purchase request/
Negotiation for best price
Permission/Parameters for
negotiation
Package
(P)
Negotiation for space at DC
Truck Agent (T)
DC Agent (D)
Negotiation for truck space

Figure 3.1: Agent based negotiations

Negotiations start when a customer initiates a search for a particular item. The search returns results that contain possible candidates for negotiation. The customer selects the packages of his choice and creates agents C_1, C_2... C_n, where *n* packages are selected for negotiation.

When package agent P_i receives a request for negotiation, it performs the following steps:

1. Obtain permission for negotiation from the owner agent O_j. This is extremely important since the owner might not be willing to negotiate due to policy restrictions.
2. Obtain parameters for negotiation such as price range, shipping policy, time for shipping etc.
3. P_i searches for distribution centers that it could use to ship itself to the new customer in order to obtain the best price for transportation/storage.
4. The package selects the least cost path using distance-vector algorithm and calculates the cost incurred in re-routing itself to the new customer. With this as the base, it negotiates with C_i for the best price that it can get.
5. The customer agent C_i receives quotes from all the packages and negotiates with them for the best deal.

The customer receives a comprehensive report of all the negotiations that took place and is given the option of selecting the best deal. The report could also be used as an input to a decision support system, which could analyze and select the best option.

As the package moves along the public logistics network, it is constantly engaged in negotiations with the resources to obtain the least cost path to the customer. According to the above model, only the package agent has multi-party negotiation capabilities. All the other agents negotiate only with the package agent. This structure provides the model with a better integration and security, and minimizes complexity.

Agent descriptions provide an ability to specify both static and dynamic characteristics of various supply chain entities. Each agent is specialized according to its intended role in the supply chain (e.g., manufacturer agents, transportation agents, supplier agents, distribution center agents, retailer agents, and end-customer agents).

3.3 Agent Development Platforms

Agent development platforms are an integrated tool suite for constructing intelligent software agent. It includes tools for managing the agent-based software development process, analyzing the domain of agent operations, designing and developing networks of communicating agents, defining behaviors of individual agents, and debugging and testing agent software.

There are various agent development platforms that are presently being used, with most of them still in the process of being evolved with newer protocols and technologies. The tools are categorized as either commercially available products or academic and research projects. They include some of the popular platform like SWARM [15], which use Objective C and Java and is a multi agent simulation. One of the primary criteria which differentiates most of these development platforms, is the language used in there construction. Java is the most widely language used, in agent development as it works across different platforms and is also compliant with the use of new technologies like XML which is very essential for a software agent. Considering these factors, and after a brief evaluation of the various packages that are being developed, ZEUS [16], an agent development tool by British Telecom, was found to be most suitable for agent development in logistics network.

Zeus is a “collaborative” agent building environment and component library written in Java. Each ZEUS agent consists of a definition layer, an organizational layer and a co-ordination layer. The definition layer represents the agent's reasoning and learning abilities, its goals, resources, skills, beliefs, preferences, etc. The organization layer describes the agent's relationships with other agents. The co-ordination layer describes the co-ordination and negotiation techniques the agent possesses. Communication protocols are built on top of the co-ordination layer and implement inter-agent communication. Beneath the definition layer is the API.

The main advantages of using ZEUS were use of Java as the development language, which was very essential to make it platform independent. Also it had a very good Graphic User Interface and automatic code generation mechanism. It provided with a good documentation of different case studies including a negotiation based model in a market scenario.

But the main disadvantage of using ZEUS was its incompatibility with the implementation of the adaptive routing protocol. The use of the distance vector algorithm required extensive coding, and most of the features offered by ZEUS became irrelevant. Any change in building an agent, other than the way it was done in the case studies, required a lot of coding, making the utility of ZEUS more as a GUI tool.

CHAPTER 4: SUPPORTING TECHNOLOGIES

4.1 Web Services

Web services, in the general meaning of the term, are services offered via the Web. In a typical Web services scenario, a business application sends a request to a service at a given URL using the SOAP protocol over HTTP. The service receives the request, processes it, and returns a response. The core of Web services uses standards like XML, SOAP, UDDI, and WSDL. An often-cited example of a Web service is that of a stock quote service, in which the request asks for the current price of a specified stock, and the response gives the stock price. This is one of the simplest forms of a Web service in that the request is filled almost immediately, with the request and response being parts of the same method call [17].

Another example could be a service that maps out an efficient route for the delivery of goods. In this case, a business sends a request containing the delivery destinations, which the service processes to determine the most cost-effective delivery route. The time it takes to return the response depends on the complexity of the routing, so the response will probably be sent as an operation that is separate from the request.

Web services and consumers of Web services are typically businesses, making Web service messages predominantly a business-to-business (B2B) transactions. An enterprise can be the provider of Web services and also the consumer of other Web services. For example, a wholesale distributor of spices could be in the consumer role when it uses a Web service to check on the availability of vanilla beans and in the provider role when it supplies prospective customers with different vendors' prices for vanilla beans.

Web services depend on the ability of parties to communicate with each other even if they are using different information systems. XML (Extensible Markup Language), a markup language that makes data portable, is a key technology in addressing this need. Enterprises have discovered the benefits of using XML for the integration of data both internally for sharing legacy data among departments and externally for sharing data with other enterprises. As a result, XML is increasingly being used for enterprise integration applications, both in tightly coupled and loosely coupled systems. Because of this data integration ability, XML has become the underpinning for Web-related computing.

Web services also depend on the ability of enterprises using different computing platforms to communicate with each other. This requirement makes the Java platform, which makes code portable, the natural choice for developing Web services. This choice is even more attractive as the new Java APIs for XML become available, making it easier and easier to use XML from the Java programming language [18].

In addition to data portability and code portability, Web services need to be scalable, secure, and efficient, especially as they grow. The Java Platform is specifically designed to fill just such needs. It facilitates the really hard part of developing Web services, which is programming the infrastructure. This infrastructure includes features such as security, distributed transaction management, and connection pool management, all of which are essential for industrial strength Web services. And because components are reusable, development time is substantially reduced.

Because XML and the Java platform work so well together, they have come to play a central role in Web services. In fact, the advantages offered by the Java APIs for XML and the J2EE platform make them the ideal combination for deploying Web services.

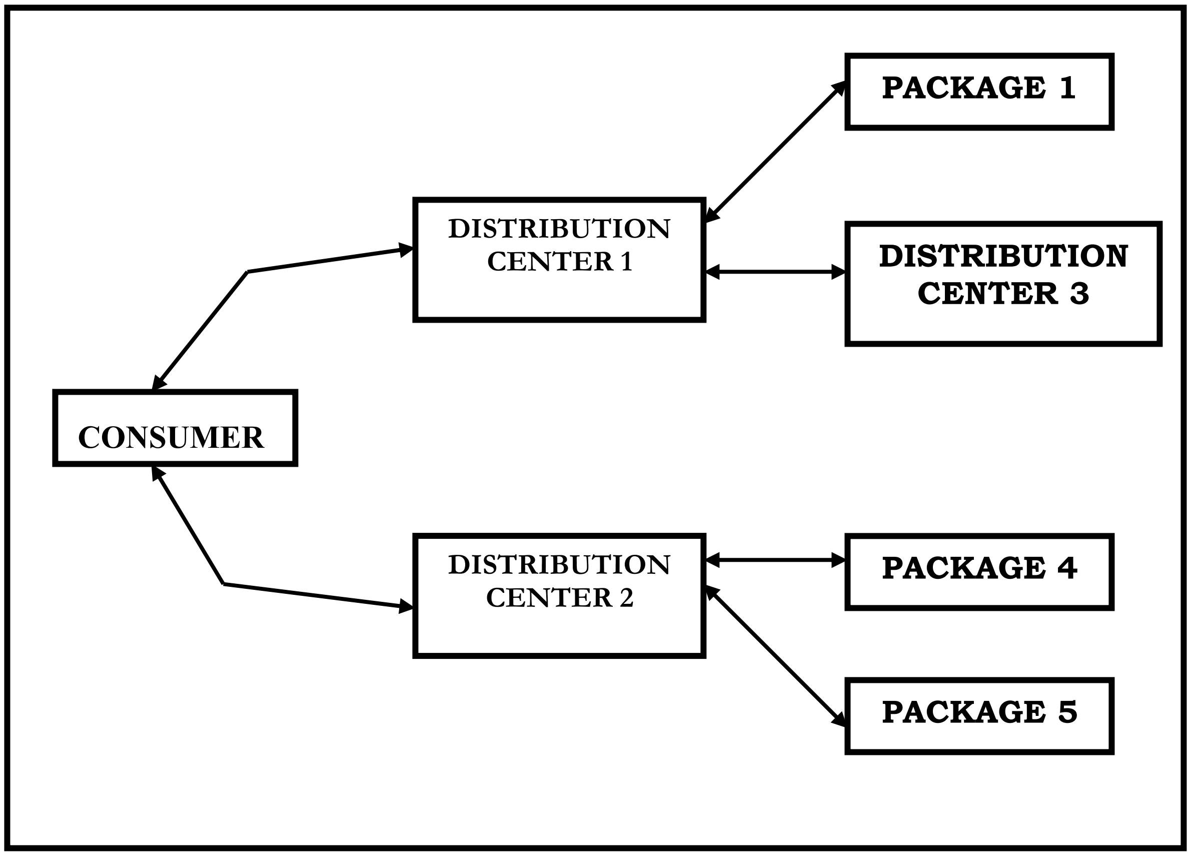


Figure 4.1: Example of Implementation of Web Services

In the following example, we can show how the web services can be implemented in a Public Logistics Network through a system of buying and selling agents (representing Distribution Centers, consumers, and packages) that engage in multi-parameter negotiation and run on wireless mobile devices. Agents representing well-informed consumers and participating suppliers interact one to one on equal footing to seek agreement on the terms of a consumer purchase. Here DC's can act both as a supplier as well as a customer, depending on whether it is interacting with a consumer who actually requires the product or with the package, which is in a nearby DC or route.

Although net-based product information helps consumers make educated buying decisions from their homes and offices, consumers lack the resources to perform comparison-shopping activities at the point of purchase. This is further compounded by how critical the component is for them and how quickly they can get the product. The consumer's personal digital assistant (PDA) can serve this immediate need, finding web-based product reviews and alternative offerings. In this setting, the consumer may direct the PDA to begin a negotiation with the nearest DC's (and others including actual suppliers) or simply add the product to a "want list" to enable ongoing negotiations and a later purchase.

At any time, the consumer may add items, noting preferences such as warranty terms, merchant reputation, availability, time limit for the purchase, and preferred price. As shown in the Figure 4.1 above, we can consider that packages nearby meet the requirements of the customer, and are at nearby DC's or on a nearby route on its intended journey. The DC will engage nearby DC's or the package agents in a silent exchange seeking the items on the "want list" and opening negotiations on the terms of the sale, alerting its owner if a deal is reached.

Behind the scenes, the selling agents (either the DC's depending on their role or the packages') negotiate sales terms, considering such factors as availability, probability of an immediate sale, and the age and shelf-life of goods on hand, following guidelines set by the merchant. Through the negotiation process leading up to a successful or lost sale, the merchant gains valuable information about customers' purchasing decisions. Questions such as recurrence of sale of a particular product, the price a customer is willing to pay even if required urgently or why a sale didn't occur can be answered using information obtained or inferred during the agent negotiation. We can anticipate that merchants will learn and respond to localized consumer preferences through the analysis of aggregate data acquired through long-term negotiations with large numbers of customers and potential customers.

The ability of a DC's to engage a package in a negotiation depends upon the supplier's willingness to incorporate the necessary network and software infrastructure into a package to make such an exchange possible. In this research I do not attempt to solve this limitation, but hypothesize that as this level of consumer-DC-package interaction becomes possible, consumers will value these capabilities equally with traditional values such as reputation, price, and variety. Consumer behavior will lead merchants to participate in information exchange and negotiation, as customer demand has previously led them to adopt fax machines, e-mail, and web sites.

4.2 Extensible Markup Language (XML)

Extensible markup language (XML) has attracted considerable attention from enterprises interested in employing dynamic information exchange over the Internet between trading partners. XML is a more advanced language than the Hypertext markup language (HTML), which is currently used to exchange data on the Internet. XML allows for the use of special tags before a message that would allow a browser to identify the message. Through the use of those tags, the search agent could immediately recognize the identity of the product and its characteristics [19].

Java technology and XML are a natural match for the creation of applications that exploit the web of information where different classes of clients, from a traditional phone to the latest smart refrigerator, consume and generate information that is exchanged between different servers that run on varied system platforms. The following information should be available in the XML tag [8]:

- Item Specific Information

Item specific negotiation code Manufacturer & Model	Physical Description Other relevant Information

- Instance Specific Information

Globally Unique Serial Number Location Price Current Owner	Parameter showing the willingness to negotiate Other relevant information

For smooth transfer of information between different types of computer systems, it is essential to set a standard format the tags should conform to.

The tree diagram shown in Figure 4.2 below represents the XML structure.

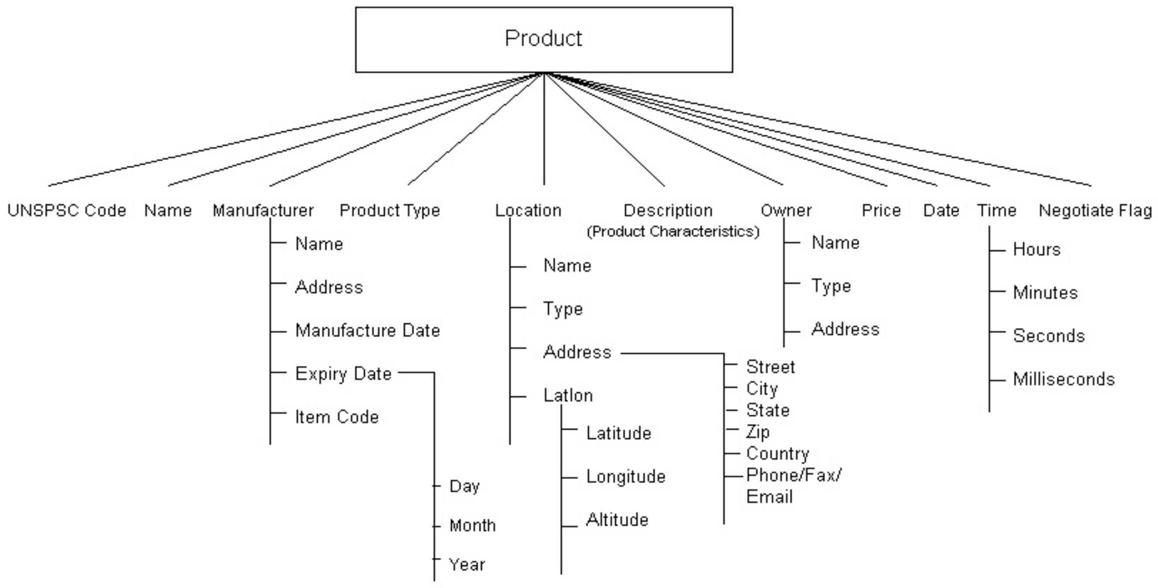


Figure 4.4: XML structure [8]

4.3 Simple Object Access Protocol (SOAP)

SOAP [20] is a protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses. SOAP can potentially be used in combination with a variety of other protocols; but usually SOAP is used in combination with HTTP and HTTP Extension Framework.

In the example below, a customer sends a PriceQuote SOAP request to a DC service. The request takes a string parameter, ticker symbol, and returns a float in the SOAP response. The SOAP Envelope element is the top element of the XML document representing the SOAP message. XML namespaces are used to disambiguate SOAP identifiers from application specific identifiers. The Figure 4.3 also illustrates the HTTP bindings [20] .

```
POST /TransportPriceQuote HTTP/1.1
Host: www.DC1server.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<m:GetTransportPrice xmlns:m="Some-URI">
        <name>FORD MUSTANG CAR</name>
     </m:GetTransportPrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 4.3: SOAP Message Embedded in HTTP Request

Figure 4.4 is the response message containing the HTTP message with the SOAP message as the payload.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn


<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
     <m:GetTransportPriceResponse xmlns:m="Some-URI">
        <Price>120</Price>
     </m:GetTransportPriceResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 4.4: SOAP Message Embedded in HTTP Response

4.4 Universal Description, Discovery and Integration (UDDI)

UDDI is a Web-based distributed directory that enables business to list them on the Internet and discover each other, similar to a traditional phone book's yellow and white pages. UDDI uses the XML Schema Language to formally describe its data structures. Web services based on UDDI are an evolution in e-business applications that will help reach these goals and take B2B to the next level. UDDI is a specification for distributed Web-based information registries of Web services. UDDI is also a publicly accessible set of implementations of the specification that allow businesses to register information about the Web services they offer so that other businesses can find them. Web services are the next step in the evolution of the World Wide Web (WWW) and allow programmable elements to be placed on Web sites where others can access distributed behaviors [21].

UDDI registries are used to promote and discover these distributed Web services. The core component of the UDDI project is the UDDI business registration, an XML file used to describe a business entity and its Web services. Conceptually, the information provided in a UDDI business registration consists of three components: “white pages” including address, contact, and known identifiers; “yellow pages” including industrial categorizations based on standard taxonomies; and “green pages” the technical information about services that are exposed by the business. Green pages include references to specifications for Web services, as well as support for pointers to various file and URL based discovery mechanisms if required.

UDDI-based registries will help businesses, in our case the Distribution Centers to take advantage of Web services:

- Distribution Centers will have a means to describe their services and business processes in a global, open environment on the Internet thus extending their reach.
- Potential trading partners (DC and consumer or DC and packages) will quickly and dynamically discover and interact with each other on the Internet via their preferred applications thus reducing time to market.

CHAPTER 5: Adaptive Routing Representation in PLN using Java

5.1 Basic Assumptions

1. This is being done for a Public Logistics Network (PLN). The network has 36 nodes and 59 routes. Each node represents a Distribution Center (DC). The intent of this model is to find the best route from any given origin to a destination, using the Distance Vector Routing algorithm.
2. The costs between all the nodes are known. Each DC is aware of the identity of every other DC, and is also aware of the cost of the routes to its immediate neighbors. After several iterations of propagations, each DC is aware of the (best) cost to every other DC in the network, and the next (neighboring) DC it has to take to reach the destination DC.
3. The packages are transported by trucks, which ply between two DCs. And at every DC the package has to be unloaded and loaded on to a new truck, going along the intended route. As a result, passing through a DC has an associated cost.
4. When a package is introduced to the network, it has a designated origin and a designated destination. It may also have a designated intermediate hop. It is also assumed that the package in transit is available for In-transit trade.
5. Upon introduction of the package, it sends out Package Agents representing itself to all DCs in the network. These package agents contain the package's description, and the cost required for the package to travel from its Home DC (the current DC where the package is present) to the DC of the package agent.

6. As the package travels through the network, it records the DCs it has passed through, and the sunk route cost so far, and is aware of the next DC on its intended path.
7. The cost of the route may change randomly. This change in cost is propagated by the relevant DCs (the DCs at either ends of the route). This may lead to a change in the intended route of a package. When the package is in transit between DCs, any change in cost does not affect the intended route. It is only when the package reaches a DC, it re-evaluates the intended route for its validity as the best route.

5.2 Model Initialization and Operations

The basic initializations that take place once the Java programs are executed are the Route initialization and the DC initialization. It basically means that data from the AllRoutes.txt (Appendix B.1) and AllDCs.txt (Appendix B.2) are read by the system and initialized for further execution.

In Route Initialization, it loads all routes information in the network. Route information for a route consists of the two end DCs of the route, and the initial cost between them. The cost of a route is same in either direction, i.e., for example cost from DC 7 to DC 24 is the same as cost from DC 24 to DC 7.

In DC Initialization, all DCs are created. Each DC knows the identity of every other DC in the network. Each DC is aware who its neighbors are, and the cost to each of these neighbors. At the creation of each DC, a routing table is created for each DC, which consists of the path and cost to every other DC in the network. At the time of initialization, the cost and path to non-neighboring DCs are not known. In such cases, the cost is considered infinity (a cost greater than the sum of the longest path in the network), and the path as unknown.

Once the initializations occur, the next step in the sequence of events is the Package Creation. It is manual, done by the supplier when the item is created and its information is released into the network. A package in this model contains the following information: Package Id, Content Id, Origin, and Destination.

When created, a package registers itself at the origin. The DC where a package is registered is called its “home” base. It them spawns package agents and registers these agents in all the neighbors of the current Home (initially the Origin) DC. It also maintains three types of cost information: sunk cost, cost from current home to destination, and cost from origin to destination. Sunk cost is the transportation cost already incurred by the package to reach its current location in the network. Cost from current home to destination, is the cost that the package is supposed to incur to travel from its current home to the desired destination at that instance of time. This may or may not change, depending on the route costs of its intended journey. Cost from origin to destination is the sum cost of the “sunk cost” and “cost from home to destination”.

Package Agent, is an agent of that particular package for which it holds the reference. It also holds the information as to the cost required to divert the package to the DC, where it is residing. A package hops from its current home DC to a neighbor of the current home DC, if that neighbor is in the best path to the destination of the package. This information is supplied by the routing table present in the current home DC. As the hop occurs, the package changes its home DC from the previous DC to current DC. It also moves its agents along with it. The cost of the route between the first and the next

DC is considered as sunk cost of the package. Also when a hop occurs, it records the journey it has undertaken so far.

As the package moves from current DC to the next DC, it un-registers all agents in the neighbors of the current DC, and registers agents for itself in all the neighbors of the next DC. The agents now hold updated cost of the package, if it has to come to the DC where the agent resides. This takes into account the sunk cost incurred by the package in its journey so far. When an In-transit trade occurs, the destination of the package is changed, with the new destination being the new destination DC. The next hop is now recalculated for the new destination.

When the route cost of any route in the network is changed, it is propagated throughout the network. All packages are asked to recalculate their expected cost to their destinations, and are also asked to update their journey information, if necessary. Each DC has a routing manager created for it. This routing manager runs in a separate thread created specifically for each DC. Its responsibility is to propagate the routing table of that DC to each of its neighbors, every time a change in cost occurs.

5.3 **Pseudo-code for Routing Table Propagation**.

A variation of Distance Vector algorithm called Path Vector Algorithm is used in this model. Every DC in the network has a routing table associated with it. The routing table consists of a Vector. Each element in the Vector consists of the following:

1. The Id of the destination DC.
2. The path vector to that DC. The path vector consists the following information: Destination DC Id, the cost to the destination DC and the Ids of all DCs in the route to the destination DC.

Initialization of Routing Table has the following steps associated with it:

1. Each DC is initialized with a routing table, which consists the identity of all DCs in the network.
2. In the initial routing table, only the costs to neighbors are known. The routes to the neighbors are also known. It is the direct route to the neighbor.
3. The costs to all other DCs are initialized as INFINITY. This is the network's infinity, which is a unit greater than the cost of the longest route in the network. The longest route possible in the network includes all the routes present in the network. The routes to all non-neighbor DCs are empty at initialization.
4. Each DC has its own Routing Manager, which manages routing for that DC in a concurrent thread of its own.

Routing table propagation has the following steps associated with it:

1. A DC receives the following information from a neighbor – Id of the neighboring DC and neighbor's routing table
2. The cost to the neighboring DC is queried from home DC's routing table.
3. For a given destination DC, the cost to that destination is queried from home DC's routing table.
4. The cost to and from the neighbor to the same destination DC (as present in the neighbor's routing table) is added.
5. If a DC is not in the neighbor's route to the destination DC, and if the cost in step 4 is less than the cost in step 3, the routing table of this DC is updated as explained below.

Routing table updation has the following steps:

1. The current cost to a destination from this DC is replaced with the new cost.
2. The current route to the destination is discarded. A new route is formed, starting from the route from this DC to its neighbor (this may not be the direct route, if an intermediate DC offers a lower route than the direct route), and then the neighbor’s route to the destination DC (which resulted in the lower cost).

5.4 UML Representation of the Model

The Unified Modeling Language (UML) is the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. It simplifies the complex process of software design, making a "blueprint" for construction [22].

I have used the UML from the Rational Rose Enterprise Edition to design the Sequence Diagrams for this particular model. Sequence diagrams show in detail the messages dispatched between participating objects along with their time ordering. The objects displayed are known from the software architecture of the model. Each column represents an object. The object names are given at the head of each column. The vertical dotted lines below the object names are time lines for each object. The time is increasing from top to down. Horizontal lines with single sided arrows are asynchronous messages. Rectangles at the end of message arrows show that the receiving object processes the message [23].

There are four sequence diagrams described below, which shows how the Java model works for Package Creation, Package Hop, Destination Change and Cost Change.

Sequence flow when a package is created:

1. When a new package is created and introduced into the system by the PackageCreatorPanel, the Package factory is notified about it.
2. Package properties are then recorded, based on which the journey is initialized. This depends on the origin and the destination of the package in the network.
3. Package agents are first created, which propagate in the network, to find the best route to reach the desired destination, and they register in all the neighbors of the Home DC.
4. Package registers at the Home DC and its agents at all its neighbors.
5. Based on the current costs in the network, the best route for the intended journey and next hop is also calculated.

Sequence Flow when Package hops to the next DC:

1. The PackagePanel class calls the hop () method of the selectedPackage object of the Package class.
2. In the next step, the current DC un-registers itself as the current Home of the package and also the package agents from its neighboring DC's.
3. The next DC where the package is presently located registers itself as the 'Home' DC and propagates the package agent to all its neighbors.
4. Meanwhile the Hop event is recorded and process for evaluating the best route along its intended journey is started.

Sequence flow when there is a change in destination:

1. Object packagePanel of the class PackagePanel calls destinationChanged method of the object selectedPackage of the Package class.

2. The selectedPackage object calls the destinationChanged method of the journey object.

Sequence flow when cost change occurs in the network:

1. Any change in cost in the network is read by the RoutePanel class and notified to the Route factory.
2. The new cost for the selected route is taken into consideration and concerned DCs and Package are notified about the change.
3. It triggers a re-initialization of the route costs if there is a package moving along the route. This may or may not result in change in journey of the package depending on the amount of change in cost.
4. If the cost change is large enough to result in change in route, the journey of the affected packages are changed.
5. The new journey and cost are re-calculated.

5.5 Model Scenarios

The various scenarios that the model handles are:

1. Route initialization, for a package, given the origin and the destination using Adaptive Routing Algorithm
2. Cost change, for a route, which may or may not result in route change for a package journey in the network.
3. In-transit trade, which may or may not lead to change in destination and subsequent transportation cost.

Figure 5.1 shows an example of adaptive routing and in-transit trade for a single package being transported from DC 4 (Jacksonville, FL) to DC 30 (Richmond, VA). The current “Home Agent” for the package is located at DC 24. The Home Agent coordinates the interactions between all of a package’s other agents, routes the package, and, in the case of the possible resale of the package, forwards final offers to the current owner of the package (possibility an agent at a factory in Jacksonville, FL) and, if accepted, arranges for the re-routing of the package to its new destination.

In Figure 5.1, the package itself is onboard a truck traveling north along I-95 heading to DC 24. The package’s transport cost from DC 4 to DC 24 has already been paid via micro payments and is a sunk cost. The relevant cost for routing purposes is the future cost of transporting the package from DC 24 to its destination; this cost is currently \$0.58, assuming the package will travel via DCs 24, 23, 17, and 30. Once the package leaves DC 24, its Home Agent moves (as a mobile agent) to the next hop, DC 23.

The package agents at DCs 25, 20, 18, and 22 are available to determine alternative routes if necessary. The cost reaching the destination (DC 30) via these agents is greater than along the intended route. Each agent at each DC has a “via cost,” the cost reaching the destination via the DC. Table shows the effects of four possible disruption scenarios: Scenarios 1–3 correspond to decreases in the transport costs from DC 20 (Charlotte, NC) to DC 18 (Greensboro, NC), possibly due to an oversupply of trucks headed to Charlotte; decreases of \$0.05, \$0.12, and \$0.22 are shown in the table. Scenario 4 corresponds to an increase of \$1.00 in the transport cost from DC 23 to DC 17, possibly due to an accident along I-95 that will require trucks to be re-routed via non-interstate roads.

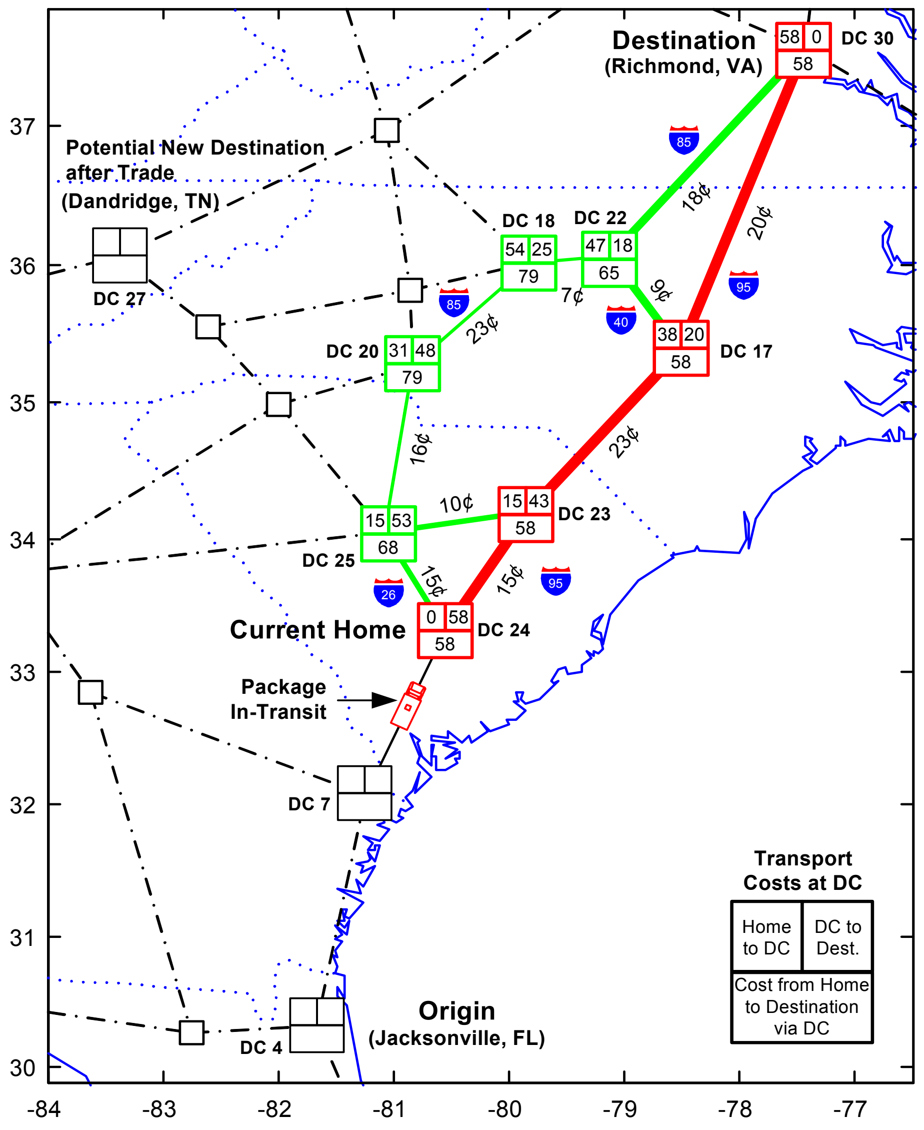


Figure 5.1. Example of Adaptive routing [9]

Scenarios 3 and 4 result in changes that are significant enough to trigger the re-routing of the package; Scenarios 1 and 2 terminate without re-routing. What is most important from an implementation point of view is that the disruptions are only propagated locally. The DC at the source of the disruption propagates its new via cost to all of its neighbors. Each neighbor DC needs to continue to propagate the change to its neighbors only if it would result in a change to its predecessor or successor.

Table 5.1 Adaptive Routing Scenarios [9]

Scenario		DC	24	23	17	25	20	18	22	30
0	Current	Via Cost (¢)	58	58	58	68	79	79	65	58
:		Pred. DC	–	24	23	24	25	20	17	17
		Succ. DC	23	17	30	23	18	22	30	–
1	23¢ → 18¢ = ↓5¢	Via Cost (¢)	–	–	–	–	74	74	–	–
:	along arc (20,18)	Pred. DC	–	–	–	–	–	–	–	–
	⇒ No change	Succ. DC	–	–	–	–	–	–	–	–
2	23¢ → 11¢ = ↓12¢	Via Cost (¢)	–	–	–	67	67	67	–	–
:	along arc (20,18)	Pred. DC	–	–	–	–	–	–	–	–
	⇒ No change	Succ. DC	–	–	–	20	–	–	–	–
3	23¢ → 1¢ = ↓22¢	Via Cost (¢)	57	–	–	57	57	57	57	57
:	along arc (20,18)	Pred. DC	–	–	–	–	–	–	18	22
	⇒ Re-route	Succ. DC	25	–	–	20	–	–	–	–
4	23¢ → $1.23 = ↑$1.00	Via Cost (¢)	79	89	90	79	79	79	79	79
:	along arc (23,17)	Pred. DC	–	–	22	–	–	–	18	22
	⇒ Re-route	Succ. DC	25	25	–	20	–	–	–	–

Any change along the intended route will reach the Home Agent, at which point the agent can be re-routed if necessary. As can be seen in the table, only a significant change will result in significant communications between the DCs; most minor changes will be terminated locally. Since hundreds of millions of agents could be active at any time coordinating tens of millions of packages, reducing communications requirements is an important feature for any routing protocol.

With respect to in-transit trade, any of the DC agents in Figure are available as trading agents. For example, if a customer at DC 27 (Dandridge, TN) is interested in purchasing the package, it can spawn search agents. These agents would first reach the package's agents at DCs 25, 20, and 18. The cost of re-routing the package from its current location at DC 24 to these DCs is immediately known to be $0.15, $0.31, and $0.54, and can be used as part of the delivered price in the resale negotiations.

5.6 Model Verification and Validation

Model verification is the process of determining that the model operates as intended. This was performed for this particular system, by changing the data in the input data files. The route costs were all set to zero as well as infinity and system behavior was verified. All the costs were initially set to zero, and the journey obtained. When a cost was added to one of the routes in this path and the system was re-initialized, this route was avoided as that represented a higher cost. Similarly, when all the costs were set to infinity and route costs were added one by one, these routes always formed part of the intended journey, as they represented a lower total cost.

Model validation is the process of determining whether a simulation model is an accurate representation of the real system, for the particular objectives of the study. The results obtained for a smaller network with 8 DCs and 20 routes were found to be true when done manually.

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

This report describes the idea of Public Logistics Network and provides an outline of the technology that could be used to support such a system. In addition to providing traditional warehousing and storage functions for hire, a public logistics network would make it possible to negotiate with multiple firms on a load-by-load basis in order to determine the most efficient means of providing the resources needed to complete each stage of a load's transit through the network.

The report gives an overview of the supporting technologies that make Public Logistics Network a reality. This research gives an overview of the various entities that make up the Public Logistics Network and proposes a model that will use the suitable routing protocols and latest technologies to provide the necessary visibility and flexibility to the system.

The Java model gives us a good insight of the various scenarios that are possible in a Public Logistics Network, including the in-transit trade and cost variations. After looking at various routing protocol requirements and different choices available, it implements the Distance Vector Algorithm. The main advantage of using this is that it makes use of distributed routing protocol. This lowers the overheads required and lesser memory for routing tables. This means huge savings in the band-with required over the Internet and faster computation.

The following conclusions can be drawn from the Java model:

- Use of Distance Vector Algorithm is the most optimum way of calculating the best path over the network. It is fast, requires less memory and is scalable very easily over large network topologies.
- Public Logistics Network handles in-transit trade and cost variations very effectively. This demonstrates that the operations that are presently handled by large monopolies can be achieved through many smaller players (DCs) with the same efficiency and reliability, with the use of new technologies like Web Services.
- Use of Software Agents and agent concepts, provide an ability to specify both static and dynamic characteristics of various supply chain entities. Their knowledge base, inference mechanism and an explicit model of the problem to solve ability enable us to perform specific tasks in the supply chain of which this logistics network is a part.
- In this era of the Internet, Web Services play an important role in our day-to-day life. Web Services can be implemented effectively over the entire network, to provide variety of services including mapping out an efficient route for the delivery of the goods, enable in-transit trade and use of latest technologies.

Use of latest technologies like XML, Java, SOAP and UDDI gives the ability of entities in the supply chain in general and public logistics network specifically, to communicate with each other even if they are using different information systems and across various platforms.

The present Java model has to be extended also to include economic analysis and detailed modeling of agent-based negotiations. The development of the search/track program would be the next big step forward. It would involve the design of PackageIP servers as well as the design of agent-based negotiation protocols, in addition to building the agents. Complementing this work, development of XML tags, SOAP and UDDI should also proceed along the guidelines provided in this report.

APPENDIX A

Java Program 1: DC.java

```
import java.util.Vector;
import java.util.Hashtable;
import java.util.Enumeration;

/**
 * This class represents a Distribution Center.
 * The DC uses a variation of Distance-Vector algorithm called Path Vector
 * algorithm to build its routing tables.
 * Each DC is aware of the identity of every other DC in the network. It knows who its
 * neighbors are, and the cost to each of its neighbor.
 * For any change in cost, it re-builds its routing table, and signals
 * each of its neighbors to build their routing tables. Likewise, a
 * neighboring DC can also signal this DC to re-build the routing table
 * with its updated routing table / route cost.
 * The DC also accepts registrations of Packages who have this DC as their home DC.
 * and, also for PackageAgents from neighboring DCs.
 */
 public class DC {
            // Private member variables.
            /**
             * To identify this DC uniquely
             */
            private String id;
            /**
             * The routing table for this DC.
             * Contains the cost to every DC in the network, and also the
             * path to each DC (which has this lowest cost).
             */
            private RoutingTable routingTable;
            /**
             * Holds all Package Agents currently registered at this DC.
             */
            private Vector packageAgents;
            /**
             * Holds all Packages, for which this DC is their current home.
             */
            private Vector homePackages;

             // Constructor
             /**
              * Create a DC.
              * The only constructor for the DC class. Initializes this DC's routing table.
              *
              * @param id the Id for this DC
              * @param costToNeighbours a hashtable containing all neighbouring DC Ids as
              *                               keys, and the costs to each of them as the values.
              *
              */
             public DC (String id, Hashtable costToNeighbours) {
                        Log.log("DC constructor for " + id + " with cost to neighbours as " +
costToNeighbours.toString(),
                                                    Log.FULL);
                        setId(id);
                        initializeRoutingTable(costToNeighbours);
                        packageAgents = new Vector();
                        homePackages = new Vector();
             } // end of constructor

            /**
             * Initialize the routing table of this DC.
             *
             * @param costToNeighbours a hashtable containing all neighbouring DC Ids as
             *                                  keys, and the costs to each of them as the values.
             *
             */
            private void initializeRoutingTable(Hashtable costToNeighbours) {
             Log.log("DC::initializeRoutingTable for id " + id, Log.FULL);
             routingTable = new RoutingTable(this.getId(), costToNeighbours);
            }

            // Accessors
            /**
             * Returns the unique identifier for this DC.
             *
             * @returns String the unique identifier
             */
            public String getId() {
                       return id;
            }

            /**
             * Returns a hash table containing the cost to each of its neighbours.
             *
             * @returns Hashtable a hashtable containing all neighbouring DC Ids as
             *                                  keys, and the costs to each of them as the values.
             *
             */
            public Hashtable getCostToNeighbours() {
                       return routingTable.getCostToNeighbours();
            }

            /**
             * Returns a Vector of all the packages which have this DC in the home.
             *
             * @returns Vector of all the packages which have this DC in the home.
             */
            public Vector getHomePackages() {
                       return homePackages;
            }

            /**
             * Returns a Vector of all the package agents which are currently registered
             * in this DC.
             *
             * @returns Vector of all the package agents which are currently registered
             * in this DC.
             *
```

```
         */
        public Vector getPackageAgents() {
                return packageAgents;
        }

        // Routing table related methods
        /**
         * Reinitialize the routing table of this DC. (A passthrough method to this
         * DC's routing table).
         * Used to reinitalize the state of this routing table, whenever there is a cost
         * change in the network's routes.
         *
         * @param newCostToNeighbours a hashtable containing all neighbouring DC Ids as
         *                            keys, and the (new, if updated) costs to each of them as the values.
         *
         */
        public void reinitialize(Hashtable newCostToNeighbours) {
                routingTable.reinitialize(newCostToNeighbours);
        }

        /**
         * Gets the cost to all the DCs, from this DC. (A passthrough method to this
         * DC's routing table).
         *
         * @returns Hashtable containg all the DC Ids as keys, and the cost to each one
         *                                    of them as the values.
         *
         */
        public Hashtable getCostToAllDCs() {
                return routingTable.getCostToAllDCs();
        }

        /**
         * Gets the next hop to all the DCs from this DC.
         *
         * @returns Hashtable containg all the DC Ids as keys, and the Id of the next
         *                                    DCs, which have to hopped to, in order to reach the DC
    specified
         *                                    as the key, as the values
         *
         * @deprecated
         *
         */
        public Hashtable getNextHopToAllDCs() {
                return routingTable.getNextHopToAllDCs();
        }

        /**
         * Returns the next hop to be taken towards the target DC.
         * (A passthrough method to this DC's routing table).Assumes that the
         * routing table is valid and current.
         *
         * @param dcId the Id of the DC which is the target of this journey.
         * @returns String the next DC's Id, which is the next hop to the
         *                            target DC.
         *
         */
        public String getNextHopToDC(String dcId) {
                return routingTable.getNextHopToDC(dcId);
        }

        /**
         * Returns the cost to the target DC. (A passthrough method to this
         * DC's routing table).Assumes that the routing table is valid and current.
         *
         * @param dcId the Id of the DC to which the cost is required.
         * @returns Double the current best cost to the DC specified.
         *
         */
        public Double getCostToDC(String dcId) {
                return routingTable.getCostToDC(dcId);
        }

        /**
         * Propogate the Path Vector for this DC to all its neighbours.
         *
         */
        public void propogatePV(boolean force) {
                Log.log("DC::propogatePV for id " + id, Log.FULL);
                Vector neighbours = DCFactory.getInstance().getNeighboursForDC(this.id);
                for (int i = 0; i < neighbours.size(); i++) {
                        DC dc =
DCFactory.getInstance().getDCForDCId((String)neighbours.elementAt(i));
                        if (dc != null) {
                                dc.updatePV(this, routingTable.getPathVectorsTable(), force);
                        }
                }
        } // end of method propogateDV

         // Update DV info
         public void updatePV(DC neighbour, Hashtable neighboursPVT, boolean force) {
                Log.log("DC::updatePV for id " + id + " from neighbour " + neighbour.getId(),
Log.FULL);
                routingTable.updatePathVector(neighbour.getId(), neighboursPVT, force);

                 // Whenever propogation occurs, let the Factory know, so that
                 // it can print the DV Matrix
                 DCFactory.getInstance().showDVMatrix("After propogation from "
                                                                              + neighbour.getId() + " to
neighbour " + this.id);
         } // end of method updateDV


        // Package : become and unbecome home
        public void becomeHomeForPackage(Package aPackage) {
                Log.log("At " + id + ", becoming home for package " + aPackage.getPackageId(),
Log.MIN);
                homePackages.add(aPackage);

                StringBuffer txtN = new StringBuffer();
                Vector neighbours = DCFactory.getInstance().getNeighboursForDC(this.id);
                for (int i = 0; i < neighbours.size(); i++) {
```

```
                              DC aDC = 
DCFactory.getInstance().getDCForDCId((String)neighbours.elementAt(i));
                              txtN.append(aDC.getId() + ", ");
                              PackageAgent agent = new PackageAgent(aPackage.getPackageId(),

          aPackage,

          this);
                              aDC.registerPackageAgent(agent);
                    }
                    Log.log("From " + id + ", registering package agents for package " + 
aPackage.getPackageId()
                                        + " at neighbours " + txtN.substring(0, txtN.length() - 2) + ".", 
Log.MIN);
          }
          public void unbecomeHomeForPackage(Package aPackage) {
                    Log.log("At " + id + ", unbecoming home for package " + aPackage.getPackageId(), 
Log.MIN);
                    homePackages.remove(aPackage);

                    StringBuffer txtN = new StringBuffer();
                    Vector neighbours = DCFactory.getInstance().getNeighboursForDC(this.id);
                    for (int i = 0; i < neighbours.size(); i++) {
                              DC aDC = 
DCFactory.getInstance().getDCForDCId((String)neighbours.elementAt(i));
                              txtN.append(aDC.getId() + ", ");
                              aDC.unregisterPackageAgentWithId(aPackage.getPackageId());
                    }
                    Log.log("From " + id + ", unregistering package agents for package " + 
aPackage.getPackageId()
                                        + " at neighbours " + txtN.substring(0, txtN.length() - 2) + ".", 
Log.MIN);
          }

          // PackageAgent : register and unregister
          public void registerPackageAgent(PackageAgent packageAgent) {
                    Log.log("At " + id + ", registering package agent for " + packageAgent.getAgentId(), 
Log.FULL);
                    packageAgents.add(packageAgent);
          }
          public void unregisterPackageAgentWithId(String packageId) {
                    Log.log("At " + id + ", unregistering package agent for " + packageId, Log.FULL);
                    for (int i = 0; i < packageAgents.size(); i++) {
                              PackageAgent agent = (PackageAgent)packageAgents.elementAt(i);
                              if (agent.getThePackage().getPackageId().equalsIgnoreCase(packageId)) {
                                        packageAgents.remove(agent);
                                        return;
                              }
                    }
          }

           // Override Object.equals
           public boolean equals(Object o) {
                     if (! (o instanceof DC)) {
                               return false;
                     }
```

```
                if (((DC)o).getId().equalsIgnoreCase(id)) {
                        return true;
                }
                return false;
        }

        // Override Object method toString
        public String toString() {
                return id;
        }

} // end class DC
```

Java Program 2: DCFactory.java

```
import java.util.*;
import java.io.*;

/**
 * This class is a DC Factory, used to create and
 * initialize DCs
 */

public class DCFactory {
	public static String NEIGHBOURS_FILE_NAME = "E:\\Thesis\\Stable\\AllNeighbours.txt";
		// Static members
		private static DCFactory theInstance;

		// Private members
		private Vector allTheDCs;
		private Vector allDCIds;
		private Vector neighboursData;

		// constructor
		private DCFactory() {
			Log.log("DCFactory constructor.", Log.FULL);

			theInstance = this;
			allTheDCs = new Vector();
			allDCIds = new Vector();
			neighboursData = new Vector();
			initialize();
		} // end constructor

		// Initializer for DC Factory
		private void initialize() {
			Log.log("DCFactory::initialize()", Log.FULL);
			initializeAllDCIds();
			initializeNeighboursData();
			createAllDCs();

			Log.log("DCFactory::initialize Complete.", Log.FULL);
		} // end initialize

		public static void createInstance() {
			if (theInstance == null) {
				Log.log("DCFactory : theInstance is null.. so creating and initializing it anew.",
Log.FULL);
				new DCFactory();
			}
		}

		// Get the Factory Instance.
		public static DCFactory getInstance() {
			createInstance();
			return theInstance;
		} // end getInstance

		// Override Object.finalize
```

```
protected void finalize() throws Throwable {
	Log.log("DCFactory::finalize. Setting theInstance to null", Log.FULL);
	theInstance = null;
}

private boolean initializeAllDCIds() {

	/*
	allDCIds.add("DC17");
	allDCIds.add("DC18");
	allDCIds.add("DC20");
	allDCIds.add("DC22");
	allDCIds.add("DC23");
	allDCIds.add("DC24");
	allDCIds.add("DC25");
	allDCIds.add("DC30");
	*/

	String dcs = "DC";
	for (int i = 1; i < 37; i++) {
		Log.log("Adding " + dcs + i + " to the DC list.", Log.FULL);
		allDCIds.add(dcs + i);
	}

	allDCIds.trimToSize();
	return true;
}

private boolean isValidDCId(String aDCId) {
	if (allDCIds.size() > 0) {
		return allDCIds.contains(aDCId);
	} else {
		Log.log("No DC list loaded.", Log.FULL);
		return false;
	}
}

private boolean initializeNeighboursData() {
	try {
		File inpFile = new File(NEIGHBOURS_FILE_NAME);
		LineNumberReader lir = new LineNumberReader(new FileReader(inpFile));
		while(lir.ready()) {
			String line = lir.readLine();
			StringTokenizer st =  new StringTokenizer(line, "|");
			DCNeighbours dcn = new DCNeighbours();
			dcn.setThisDCId(st.nextToken());
			Vector neighbours = new Vector(5);
			while (st.hasMoreTokens()) {
				neighbours.add(st.nextToken());
			}
			dcn.setNeighbours(neighbours);
			Log.log("Adding neighbours data " + dcn.toString(), Log.FULL);
			neighboursData.add(dcn);
		}
		lir.close();
	} catch (IOException ioe) {
				Log.log("IO Error", Log.MIN);
				ioe.printStackTrace();
				System.exit(-1);
			}
			return true;
		}

		public Vector getNeighboursForDC(String aDCId) {
			for (int i=0; i < neighboursData.size(); i++) {
				if
(((DCNeighbours)neighboursData.elementAt(i)).getThisDCId().equalsIgnoreCase(aDCId)) {
					return ((DCNeighbours)neighboursData.elementAt(i)).getNeighbours();
				}
			}
			return null;
		}

		// public Getters
		public Vector getAllDCIds() {
			return (Vector)allDCIds.clone();
		}

		public Vector getAllTheDCs() {
			return (Vector)allTheDCs.clone();
		}

		public DC getDCForDCId(String dcId) {
			if (allTheDCs.size() == 0) {
				Log.log("DC::getDCForDCId - allTheDCs is empty.", Log.FULL);
				return null;
			}
			for (int i = 0; i < allTheDCs.size(); i++) {
				if ( ((DC)allTheDCs.elementAt(i)).getId().equalsIgnoreCase(dcId)) {
					return (DC)allTheDCs.elementAt(i);
				}
			}
			return null;
		} // end of method getDCForDCId

		// Create a single DC object here
		private DC createDC(String dcId) {
			Log.log("DCFactory::createDC for " + dcId, Log.FULL);
			if (isValidDCId(dcId)) {
				Hashtable costToNeighbours = getDirectCostToNeighbours(dcId);
				return new DC(dcId, costToNeighbours);
			}
			return null;
		} // end of method createDC

		/**
		 * Direct cost to neighbours.. i.e., without an intermediate hop..
		 */
		private Hashtable getDirectCostToNeighbours(String dcId) {
			Hashtable costToNeighbours = new Hashtable();
			Vector neighbours = getNeighboursForDC(dcId);
			Log.log("\tFor DC " + dcId, Log.FULL);
		if (neighbours != null) {
			for (int i = 0; i < neighbours.size(); i++) {
				Log.log("\tGetting route between " + dcId + " and " +
(String)neighbours.elementAt(i),
						Log.FULL);
				Route r = RouteFactory.getInstance().getRouteBetween(dcId,
						(String)neighbours.elementAt(i));
				if (r != null) {
					Log.log("\tAdding costToNeighbours, " + r.toString(),
Log.FULL);
					costToNeighbours.put(r.getTheOtherDCId(dcId), new
Double(r.getCost()));
				}
			}
		}
		return costToNeighbours;
	}


	// Create all DCs here
	public Vector createAllDCs() {
		Log.log("DCFactory::createAllDCs.", Log.FULL);
		for (int i = 0; i < allDCIds.size(); i++) {
			allTheDCs.add(createDC((String)allDCIds.elementAt(i)));
		}
		Log.log("Initialized all " + allTheDCs.size() + " DCs.", Log.MIN);
		allTheDCs.trimToSize();
		return allTheDCs;
	}

	public void routeCostChanged() {
		for (int i = 0; i < allTheDCs.size(); i++) {
			DC aDC = (DC)allTheDCs.elementAt(i);
			aDC.reinitialize(getDirectCostToNeighbours(aDC.getId()));
		}
	}

	// Display the DV matrix upon request
	public void showDVMatrix(String message) {
		if (Log.getLogLevel() != Log.DEBUG) return;
		// Moved method implementation and helper methods to
		// helper class to reduce clutter here.
		DCMatrixDisplayer.showDVMatrix(message);
		//DCMatrixDisplayer.close();
	} // end of method showDVMatrix

} // end of class DCFactory



/**
 * This class is a helper class used to display the
 * DV Matricies
 */
class DCMatrixDisplayer {
	// Private members
```

```
private static FileWriter matrixOp;
private static final int FIELD_LEN = 8;

public static void showDVMatrix(String message) {
        try {
                if (matrixOp == null) {
                        matrixOp = new FileWriter("matrix.out", true);
                }

                matrixOp.write(message + "\n");
                matrixOp.flush();

                Vector allDCIds = DCFactory.getInstance().getAllDCIds();

                // Print matrix header
                StringBuffer header = new StringBuffer();
                header.append(pad("") + "|");
                for (int k = 0; k < allDCIds.size(); k++) {
                        header.append(pad((String)allDCIds.elementAt(k)) + "|");
                }
                matrixOp.write(header.toString() + "\n");

                for (int i = 0; i < allDCIds.size(); i++) {
                        String dcId = (String)allDCIds.elementAt(i);
                        DC dc = DCFactory.getInstance().getDCForDCId(dcId);

                        if (dc == null) {
                                Log.log("DC was null for id " + dcId, Log.FULL);
                        } else {
                                Hashtable dv = dc.getCostToAllDCs();

                                StringBuffer line = new StringBuffer();
                                line.append(pad(dcId) + "|");

                                for (int j = 0; j < allDCIds.size(); j++) {
                                        String aDCId = (String)allDCIds.elementAt(j);
                                        Double cost = (Double)dv.get(aDCId);
                                        if (cost == RoutingTable.INFINITY) {
                                                line.append(pad("--") + "|");
                                        } else {
                                                line.append(pad(cost.toString()) + "|");
                                        }
                                }
                                matrixOp.write(line.toString() + "\n");
                                matrixOp.flush();
                        }
                }

                matrixOp.write("\n\n");
                matrixOp.flush();

        } catch (IOException ioe) {
                Log.log ("IO Error occurred when message was " + message, Log.MIN);
                ioe.printStackTrace();
        }
} // end of method showDVMatrix
```

```
	private static String pad(String contents) {
		contents = contents.trim();
		int contLen = contents.length();

		if (contLen >= FIELD_LEN) {
			// no wrap. just return.
			return contents;
		}
		int padding = FIELD_LEN - contLen;
		StringBuffer field = new StringBuffer(" " + contents);
		for (int i = 1; i < padding; i++)
			field.append(" ");

		return (field.toString());
	} // end of method align

	public static void close() {
		Log.log("Close of DCMatrixDisplayer activated.", Log.FULL);
		try {
			if (matrixOp != null) {
				matrixOp.close();
				matrixOp = null;
			}
		} catch (IOException ioe) {
			Log.log("Error in close", Log.MIN);
			ioe.printStackTrace();
		}
	}
} // end of class DCMatrixDisplayer
```

Java Program 3: DCNeighbours.java

```
import java.util.Vector;

/**
 * This class encapsulates a collection of DC and its
 * neighbours
 */
public class DCNeighbours {
        private String thisDCId;
        private Vector neighbours;

        public void setThisDCId(String aDCId) {
                this.thisDCId = aDCId;
        }
        public String getThisDCId() {
                return thisDCId;
        }
        public void setNeighbours(Vector neighbours) {
                this.neighbours = neighbours;
        }
        public Vector getNeighbours() {
                return neighbours;
        }
        public String toString() {
                StringBuffer rsb = new StringBuffer();
                rsb.append(thisDCId);
                rsb.append(" has neighbours ");
                for (int i = 0; i < neighbours.size(); i++) {
                        String nDC = (String)neighbours.elementAt(i);
                        rsb.append(nDC + ", ");
                }
                return rsb.toString();
        }
} // end of class DC Neighbours
```

Java Program 4: DCPanel.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;
import java.util.Hashtable;
import java.util.Enumeration;

/**
 * This class represents an UI to manage monitor all DCs
 */
public class DCPanel extends JPanel implements ActionListener {
	// static members
	private static Color DC_SELECTED = Color.cyan;
	private static Color DC_UNSELECTED = Color.lightGray;

	// Private member variables
	private Vector dcButtons;
	private JTextArea neighboursText;
	private JTextArea packagesText;
	private JTextArea packageAgentsText;

	// Constructor
	public DCPanel() {
		addDCButtons();
		addNeighboursData();
		addPackagesData();
		addPackageAgentsData();
	}

	private void addDCButtons() {
		// Assuming 36 DCs are present..
		JPanel dcButtonsPanel = new JPanel();
		dcButtonsPanel.setLayout(new GridLayout(3, 12, 5, 5));
		dcButtons = new Vector();

		Vector allDCIds = DCFactory.getInstance().getAllDCIds();
		for (int i = 0; i < allDCIds.size(); i++) {
			JButton dcButton = new JButton((String)allDCIds.elementAt(i));
			dcButton.setBackground(DC_UNSELECTED);
			dcButton.addActionListener(this);
			dcButtons.add(dcButton);
			dcButtonsPanel.add(dcButton);
		}

		dcButtonsPanel.setBorder(BorderFactory.createTitledBorder("All the DCs in the
system"));
		add(dcButtonsPanel);
	}

	private void addNeighboursData() {
		neighboursText = new JTextArea(8, 30);
		neighboursText.setLineWrap(true);
		neighboursText.setWrapStyleWord(true);
		JScrollPane sPane = new JScrollPane(neighboursText);
```

```
			sPane.setBorder(BorderFactory.createTitledBorder("Direct cost to neighbours for 
selected DC"));
			add(sPane);
		}

		private void addPackagesData() {
			packagesText = new JTextArea(10, 70);
			JScrollPane sPane = new JScrollPane(packagesText);
			sPane.setBorder(BorderFactory.createTitledBorder("Packages with selected DC as 
home"));
			add(sPane);
		}

		private void addPackageAgentsData() {
			packageAgentsText = new JTextArea(10, 70);
			JScrollPane sPane = new JScrollPane(packageAgentsText);
			sPane.setBorder(BorderFactory.createTitledBorder("Package Agents present in selected 
DC"));
			add(sPane);
		}

		// Action Listener
		public void actionPerformed(ActionEvent ae) {
			JButton source = (JButton)ae.getSource();

			for (int i = 0; i < dcButtons.size(); i++) {
				((JButton)dcButtons.elementAt(i)).setBackground(DC_UNSELECTED);
			}
			source.setBackground(DC_SELECTED);

			DC aDC = DCFactory.getInstance().getDCForDCId(source.getText());

			if (aDC != null) {
				updateNeighboursData(aDC.getId(), 

		DCFactory.getInstance().getNeighboursForDC(aDC.getId()));
				updatePackagesData(aDC.getHomePackages());
				updatePackageAgentsData(aDC.getPackageAgents());
			}
		}

		private void updateNeighboursData(String selectedDCId, Vector neighbours) {
			StringBuffer txt = new StringBuffer();
			for (int i = 0; i < neighbours.size(); i++) {
				String dcId = (String)neighbours.elementAt(i);
				txt.append("To " + dcId);
				txt.append(", cost is ");
				txt.append(RouteFactory.getInstance().getRouteBetween(selectedDCId, 
dcId).getCost());
				txt.append("\n");
			}
			neighboursText.setText(txt.toString());
		}

		private void updatePackagesData(Vector homePackages) {
			StringBuffer txt = new StringBuffer();
			for (int i = 0; i < homePackages.size(); i++) {
				Package pkg = (Package)homePackages.elementAt(i);
				txt.append("Package id - " + pkg.getPackageId());
				txt.append(": ");
				txt.append("Package Contents - " + pkg.getContentId());
				txt.append("(");
				txt.append(pkg.getContentDesc());
				txt.append("), ");
				txt.append("[Origin - " + pkg.getOriginDCId());
				txt.append(", Dest - " + pkg.getDestinationDCId());
				txt.append("] ");
				txt.append("Sunk Transport Cost = ");
				txt.append(pkg.getSunkTransportCost());
				txt.append("\n");
			}
			packagesText.setText(txt.toString());
		}

		private void updatePackageAgentsData(Vector agents) {
			StringBuffer txt = new StringBuffer();
			for (int i = 0; i < agents.size(); i++) {
				PackageAgent agent = (PackageAgent)agents.elementAt(i);
				txt.append("Agent Id - " + agent.getAgentId());
				txt.append(": ");
				txt.append("Package Contents - " + agent.getThePackage().getContentId());
				txt.append("(");
				txt.append(agent.getThePackage().getContentDesc());
				txt.append("), ");
				txt.append("Current Home of Package - " +
	agent.getThePackage().getCurrentHomeDCId());
				txt.append(". Total Transport cost to this DC = ");
				txt.append(agent.getTotalPackageTransportCostToThisDC());
				txt.append("\n");
			}
			packageAgentsText.setText(txt.toString());
		}


	}
```

Java Program 5: Journey.java

```
import java.util.*;

/**
 * This class encapsulates the journey of a Package.
 * The journey has two components.. the past and the future.
 * The past is the journey so far, upto the current home.
 * The future is the journey, which will be taken from the current
 * home to the current destination, <B>assuming</B> that the
 * routing tables of the DCs do not change.
 * If the routing tables are changed, the future journey also changes.
 */
public class Journey {
	private Vector journeySoFar;
	private Vector theRoadAhead;
	private boolean destinationReached;

	public Journey() {
		this.journeySoFar = new Vector();
		this.theRoadAhead = new Vector();
		this.destinationReached = false;
	}

	public void initialize(String originDCId, String destDCId) {
		Log.log("Creating journey for " + originDCId + " to " + destDCId, Log.FULL);

		DC originDC = DCFactory.getInstance().getDCForDCId(originDCId);

		Log.log("Origin DC - " + originDC.getId(), Log.FULL);

		String nextDCId = originDC.getNextHopToDC(destDCId);

		Log.log("Next DC Id - " + nextDCId, Log.FULL);

		journeySoFar.add(originDCId);
		findTheRoadAhead(nextDCId, destDCId);
	}

	public boolean isDestinationReached() {
		return destinationReached;
	}

	private void findTheRoadAhead(String nextDCId, String destDCId) {
		theRoadAhead.removeAllElements();
		Log.log("\tFrom " + nextDCId + " to " + destDCId, Log.FULL);
		while ((! nextDCId.equalsIgnoreCase(destDCId)) &&
				(! nextDCId.equals(""))) {
			Log.log("\tAdding to the road ahead - " + nextDCId, Log.FULL);
			theRoadAhead.add(nextDCId);
			DC nextDC = DCFactory.getInstance().getDCForDCId(nextDCId);
			nextDCId = nextDC.getNextHopToDC(destDCId);
		}
		theRoadAhead.add(destDCId);
	}
	// Getters
	public List getJourneySoFar() {
		journeySoFar.trimToSize();
		return Collections.unmodifiableList(journeySoFar);
	}

	public List getTheRoadAhead() {
		theRoadAhead.trimToSize();
		return Collections.unmodifiableList(theRoadAhead);
	}

	public void recordHop() {
		String theJustHoppedToDCId = (String)theRoadAhead.elementAt(0);
		String destinationDCId = (String)theRoadAhead.elementAt(theRoadAhead.size() - 1);
		journeySoFar.add(theJustHoppedToDCId);
		if (theRoadAhead.size() == 1) {
			// This means the only DC in the road ahead is the destination DC.
			// Remove it, and get out..
			theRoadAhead.removeElementAt(0);
			destinationReached = true;
			return;
		}
		DC theJustHoppedToDC =
DCFactory.getInstance().getDCForDCId(theJustHoppedToDCId);
		String nextDCId = theJustHoppedToDC.getNextHopToDC(destinationDCId);
		findTheRoadAhead(nextDCId, destinationDCId);
	}

	public void costChanged() {
		// Only the road ahead is affected.. the journey so far is unchangeable.
		String currentHomeDCId = (String)journeySoFar.elementAt(journeySoFar.size() - 1);
		String destDCId = (String)theRoadAhead.elementAt(theRoadAhead.size() - 1);
		DC currentHomeDC = DCFactory.getInstance().getDCForDCId(currentHomeDCId);
		String newNextDCId = currentHomeDC.getNextHopToDC(destDCId);
		findTheRoadAhead(newNextDCId, destDCId);
	}

	public void destinationChanged(String newDestDCId) {
		String currentHomeDCId = (String)journeySoFar.elementAt(journeySoFar.size() - 1);
		DC currentHomeDC = DCFactory.getInstance().getDCForDCId(currentHomeDCId);
		String newNextDCId = currentHomeDC.getNextHopToDC(newDestDCId);
		findTheRoadAhead(newNextDCId, newDestDCId);
	}
}
```

Java Program 6: Log.java

```
import java.util.Observable;
import java.util.Observer;
import java.io.IOException;

/**
 * Logging support for the application.
 */
public class Log extends Observable {
	// Public static members
	public static final int MIN = 1;
	public static final int FULL = 4;
	public static final int DEBUG = 5;

	// Private static members
	private static Log thisInstance;
	private static StringBuffer messageCache = new StringBuffer();
	private static int messageNumber = 0;

	// Private members
	private int logLevel;

	public static void setLogLevel(int logLevel) {
		checkInstance();
		thisInstance.logLevel = logLevel;
	}

	public static int getLogLevel() {
		checkInstance();
		return thisInstance.logLevel;
	}

	public static void log(String message, int messageLevel) {
		if ( (messageLevel != MIN) && (messageLevel != FULL) ) {
			messageLevel = MIN;
		}

		if (getLogLevel() == DEBUG) {
			try {
				System.out.println(message);
				while (System.in.read() == 1);
			} catch (IOException ioe) {
				System.err.println(ioe.getMessage());
				System.err.println("Exiting with error");
				System.exit(-1);
			}
		}

		if ((messageLevel == MIN) || (getLogLevel() == FULL)) {
			checkInstance();
			messageCache.append((++messageNumber) + " : " + message + "\n");

			thisInstance.setChanged();
```

```
				notifyChanges();
			}

		}

		private static void checkInstance() {
			if (thisInstance == null) {
				thisInstance = new Log();
				thisInstance.messageNumber = 0;
				thisInstance.logLevel = MIN;

			}
		}

		public static void notifyChanges() {
			if (thisInstance.countObservers() > 0) {
				thisInstance.notifyObservers(messageCache);
				messageCache.delete(0, messageCache.length());
			}
		}

		public static void registerObserver(Observer observer) {
			checkInstance();
			thisInstance.addObserver(observer);
			notifyChanges();
		}

		public static void unregisterObserver(Observer observer) {
			checkInstance();
			thisInstance.deleteObserver(observer);
		}

	} // end of class Log
```

Java Program 7: LogPanel.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Observer;
import java.util.Observable;

/**
 * All events logger panel
 */
public class LogPanel extends JPanel
                                                implements Observer {
        // Private members
        private JTextArea logArea;

        // Constructor
        public LogPanel() {
                addComponents();
                registerWithObservableLog();
        }

        private void addComponents() {
                logArea = new JTextArea(40, 90);
                JScrollPane sPane = new JScrollPane(logArea);
                sPane.setBorder(BorderFactory.createTitledBorder("Log Messages"));
                add(sPane);
        }

        private void registerWithObservableLog() {
                Log.registerObserver(this);
        }

        // Observer method implementation.
        public void update(Observable o, Object arg) {
                logArea.append(((StringBuffer)arg).toString());
        }


}
```

Java Program 8: MasterUI.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;

/**
 * This is the master control UI of this application
 *
 */
public class MasterUI {
	// Private member variables
	private JFrame masterFrame;
	private JTabbedPane tabs;
	private JPanel dcPanel;
	private JPanel routePanel;
	private JPanel packagePanel;
	private JPanel packageCreatorPanel;
	private JPanel logPanel;

	// Constructor
	/**
	 * Constructor of the class
	 * Creates a main frame, and adds other tabs to this frame. Each
	 * of the tab monitors, or manages each aspect of the application.
	 *
	 * @param plWait an intializing message dialog
	 */
	public MasterUI(JDialog plWait) {
		masterFrame = new JFrame("Karthik S Gandlur : Graduate Thesis : Public Logistics
Network : Master Contoller UI");

		masterFrame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
		masterFrame.addWindowListener(new WindowAdapter() {
			public void windowClosed(WindowEvent we) {
				System.exit(0);
			}});
		masterFrame.setBounds(5, 5, 1100, 800);

		addTabs();

		masterFrame.setResizable(false);
		masterFrame.setVisible(true);

		plWait.dispose();
	}

	/**
	 * Add all the tabs of the application UI.
	 */
	private void addTabs() {
		tabs = new JTabbedPane(JTabbedPane.TOP);

		dcPanel = new DCPanel();
		tabs.addTab("DC Monitor", dcPanel);
		routePanel = new RoutePanel();
		tabs.addTab("Route Manager", routePanel);

		packagePanel = new PackagePanel();
		tabs.addTab("Package Manager", packagePanel);

		packageCreatorPanel = new PackageCreatorPanel();
		tabs.addTab("Package Creator", packageCreatorPanel);

		logPanel = new LogPanel();
		tabs.addTab("Log Monitor", logPanel);

		masterFrame.getContentPane().add(tabs);
	}


	/**
	 * Point of entry to the application.
	 * Usage :
	 * <PRE>
	 * java MasterUI
	 * </PRE>
	 *
	 */
	public static void main(String args[]) {
		if (args.length > 0) {
			if (args[0].equalsIgnoreCase("FULL")) {
				Log.setLogLevel(Log.FULL);
			} else if (args[0].equalsIgnoreCase("DEBUG")) {
				Log.setLogLevel(Log.DEBUG);
			} else {
				Log.setLogLevel(Log.MIN);
			}
		} else {
			Log.setLogLevel(Log.MIN);
		}

		JDialog plWait = pleaseWait(null, "Initializing System. Please wait..");
		plWait.show();

		// Create Instances of all factories, in this order
		RouteFactory.createInstance();
		DCFactory.createInstance();
		PackageFactory.createInstance();

		// Start all routing managers.. one each for a DC
		Vector allDCs = DCFactory.getInstance().getAllTheDCs();
		for (int i = 0; i < allDCs.size(); i++) {
			new Thread(new RoutingManager((DC)allDCs.elementAt(i))).start();
		}

		new MasterUI(plWait);
	}

	private static JDialog pleaseWait(Component parent, String message) {
			JOptionPane wait = new JOptionPane(message,
JOptionPane.INFORMATION_MESSAGE);
			JDialog dialog = wait.createDialog(parent, message);
			dialog.setModal(false);
			return dialog;
		}


}
```

Java Program 9: Package.java

```
import java.util.Properties;
import java.util.Vector;

/**
 * This class represents a package in the system.
 * It is created with an id, and details about its
 * contents. At the time of creation, it is designated
 * with an origin, and a destination.
 *
 */
public class Package {
	// static Strings which are required in a
	// properties object used to construct a package.
	public static String PACKAGE_ID = "package.id";
	public static String PACKAGE_CONTENT_ID = "package.content.id";
	public static String PACKAGE_CONTENT_DESC = "package.content.desc";
	public static String PACKAGE_ORIGIN_DC_ID = "package.origin.dc.id";
	public static String PACKAGE_DESTINATION_DC_ID = "package.destination.dc.id";

	// Private member variables
	private String packageId;
	private String contentId;
	private String contentDesc;
	private String originDCId;
	private String destinationDCId;
	private String currentHomeDCId;
	private double sunkTransportCost;
	private Journey journey;

	// Constructor
	public Package(Properties packageProperties) {
		if (! isPropertiesRightForPackageCreation(packageProperties)) {
			throw new IllegalArgumentException ("Properties supplied are not valid for
Package creation.");
		}
		packageId = packageProperties.getProperty(PACKAGE_ID);
		contentId = packageProperties.getProperty(PACKAGE_CONTENT_ID);
		contentDesc = packageProperties.getProperty(PACKAGE_CONTENT_DESC);
		originDCId = packageProperties.getProperty(PACKAGE_ORIGIN_DC_ID);
		destinationDCId =
packageProperties.getProperty(PACKAGE_DESTINATION_DC_ID);

		// just when created, current home dc id is the same as the origin.
		currentHomeDCId = originDCId;

		sunkTransportCost = 0.0;

		Log.log("Creating package, with id " + packageId + ", [O - " + originDCId +
					", D - " + destinationDCId + "].",
					Log.MIN);

		initializeJourney();

		createAndDistributePackageAgents();
```

```
}

// helper method to determine if a particular Property object has
// all the required properties for the valid creation of a Package.
public static boolean isPropertiesRightForPackageCreation(Properties properties) {
	if (properties.getProperty(PACKAGE_ID).length() == 0) {
		Log.log("PACKAGE_ID is null", Log.MIN);
		return false;
	}
	if (properties.getProperty(PACKAGE_CONTENT_ID).length() == 0) {
		Log.log("PACKAGE_CONTENT_ID is null", Log.MIN);
		return false;
	}
	if (properties.getProperty(PACKAGE_ORIGIN_DC_ID).length() == 0) {
		Log.log("PACKAGE_ORIGIN_DC_ID is null", Log.MIN);
		return false;
	}
	if (properties.getProperty(PACKAGE_DESTINATION_DC_ID).length() == 0) {
		Log.log("PACKAGE_DESTINATION_DC_ID is null", Log.MIN);
		return false;
	}
	return true;
}

private void initializeJourney() {
	journey = new Journey();
	Log.log("Creating journey for Package " + packageId, Log.FULL);
	journey.initialize(originDCId, destinationDCId);
}

private void createAndDistributePackageAgents() {

	DC homeDC = DCFactory.getInstance().getDCForDCId(currentHomeDCId);
	homeDC.becomeHomeForPackage(this);
}

/**
 * One hop towards destination.
 */
public void hop() {
	double costToNextDC;
	DC nextDC;

	if (currentHomeDCId.equalsIgnoreCase(destinationDCId)) {
		// If already at home in the destination DC, dont hop anywhere else.
		return;
	}

	DC homeDC = DCFactory.getInstance().getDCForDCId(currentHomeDCId);
	String nextDCId = homeDC.getNextHopToDC(destinationDCId);

	if (nextDCId == "") {
		// This means that the package is in a DC from which the
		// next hop will lead to the destination DC itself.
		costToNextDC = (homeDC.getCostToDC(destinationDCId)).doubleValue();
		nextDC = DCFactory.getInstance().getDCForDCId(destinationDCId);
```

```
				currentHomeDCId = destinationDCId;
			} else {
				costToNextDC = (homeDC.getCostToDC(nextDCId)).doubleValue();
				nextDC = DCFactory.getInstance().getDCForDCId(nextDCId);
				currentHomeDCId = nextDCId;
			}

			Log.log("Hopping package " + packageId + ", from " + homeDC.getId() +
								" to " + nextDC.getId() + ".", Log.MIN);

			homeDC.unbecomeHomeForPackage(this);
			nextDC.becomeHomeForPackage(this);
			sunkTransportCost += costToNextDC;

			journey.recordHop();

			if (journey.isDestinationReached()) {
				Log.log("Destination Reached for package " + packageId + ".", Log.MIN);
			}

		}

		public double getTotalJourneyCostToDestination() {
			if (currentHomeDCId.equalsIgnoreCase(destinationDCId)) {
				return sunkTransportCost;
			}
			DC homeDC = DCFactory.getInstance().getDCForDCId(currentHomeDCId);
			return sunkTransportCost +
					(homeDC.getCostToDC(destinationDCId)).doubleValue();
		}

		/**
		 * Changes the destination., called usually from the UI
		 */
		public void destinationChanged(String newDestDCId) {
			Log.log("For package " + packageId + ", changing destination from " +
				destinationDCId + " to " + newDestDCId + ".", Log.MIN);
			this.destinationDCId = newDestDCId;
			journey.destinationChanged(newDestDCId);
		}

		// Getters
		public String getPackageId() {
			return packageId;
		}

		public String getContentId() {
			return contentId;
		}

		public String getContentDesc() {
			return contentDesc;
		}

		public String getOriginDCId() {
			return originDCId;
```

```
	}

	public String getDestinationDCId() {
		return destinationDCId;
	}

	public String getCurrentHomeDCId() {
		return currentHomeDCId;
	}

	public double getSunkTransportCost() {
		return sunkTransportCost;
	}
	public Journey getJourney() {
		return journey;
	}

	// Override Object method
	// used to represent this package in the UI
	public String toString() {
		return packageId;
	}

	// Override Object.equals
	public boolean equals(Object o) {
		if (! (o instanceof Package)) {
			return false;
		}
		if (((Package)o).getPackageId().equalsIgnoreCase(packageId)) {
			return true;
		}
		return false;
	} // end of equals
}
```

Java Program 10: PackageAgent.java

```
/**
 * This class represents a PackageAgent.
 * An agent for a Package object.
 */
public class PackageAgent {
	// Private members
	private String agentId;
	private Package thePackage;
	private DC agentHomeDC;

	// Constructor
	public PackageAgent(String agentId, Package thePackage, DC agentHomeDC) {
		this.agentId = agentId;
		this.thePackage = thePackage;
		this.agentHomeDC = agentHomeDC;
	}

	// getters
	public String getAgentId() {
		return agentId;
	}
	public Package getThePackage() {
		return thePackage;
	}
	public DC getAgentHomeDC() {
		return agentHomeDC;
	}

	public double getTotalPackageTransportCostToThisDC() {
		double sunkTransportCost = thePackage.getSunkTransportCost();
		DC packageHomeDC = DCFactory.getInstance().

	getDCForDCId(thePackage.getCurrentHomeDCId());
		double packageHomeToAgentHomeTransportCost = (packageHomeDC.

	getCostToDC(agentHomeDC.getId())).doubleValue();

		return sunkTransportCost + packageHomeToAgentHomeTransportCost;
	}

	// Override Object.equals
	public boolean equals(Object o) {
		if (! (o instanceof PackageAgent)) {
			return false;
		}
		if (((PackageAgent)o).getThePackage().
					getContentId().equalsIgnoreCase(thePackage.getContentId()))
{
			return true;
		}
		return false;
	} // end of equals
} // end of class PackageAgent
```

Java Program 11: PackageCreatorPanel.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Properties;

/**
 * This class presents an UI to create Packages
 */
public class PackageCreatorPanel extends JPanel {
	// Private fields
	private JTextField txtPackageId;
	private JTextField txtContentId;
	private JTextField txtContentDesc;
	private JComboBox cmbOriginDCId;
	private JComboBox cmbDestDCId;

	private JButton cmdCreatePackage;
	private JButton cmdClearFields;

	// Constructor
	public PackageCreatorPanel() {
		addComponents();
	}

	private void addComponents() {
		JPanel compPanel = new JPanel();
		compPanel.setLayout(new GridLayout(6, 2));

		compPanel.add(new JLabel("Package Id : "));
		txtPackageId = new JTextField(10);
		compPanel.add(txtPackageId);

		compPanel.add(new JLabel("Content Id : "));
		txtContentId = new JTextField(10);
		compPanel.add(txtContentId);

		compPanel.add(new JLabel("Content Desc : "));
		txtContentDesc = new JTextField(40);
		compPanel.add(txtContentDesc);

		compPanel.add(new JLabel("Origin DC Id : "));
		cmbOriginDCId = new JComboBox(DCFactory.getInstance().getAllDCIds());
		compPanel.add(cmbOriginDCId);

		compPanel.add(new JLabel("Destination DC Id : "));
		cmbDestDCId = new JComboBox(DCFactory.getInstance().getAllDCIds());
		compPanel.add(cmbDestDCId);

		cmdCreatePackage = new JButton("Create Package");
		cmdCreatePackage.addActionListener(new ActionListener(){
				public void actionPerformed(ActionEvent ae) {
					if (checkFieldsValidity()) {
						createPackage();
```

```
                                        }
                                }
                        });
                compPanel.add(cmdCreatePackage);

                cmdClearFields = new JButton("Clear all fields");
                cmdClearFields.addActionListener(new ActionListener(){
                                public void actionPerformed(ActionEvent ae) {
                                        clearAllFields();
                                }
                        });
                compPanel.add(cmdClearFields);

                compPanel.setBorder(BorderFactory.createTitledBorder("Creation of Packages"));
                add(compPanel);
        }

        private boolean checkFieldsValidity() {
                if (txtPackageId.getText().length() == 0) {
                        showErrorMessage("Package Id cannot be empty.");
                        return false;
                }

                if (txtContentId.getText().length() == 0) {
                        showErrorMessage("Content Id cannot be empty.");
                        return false;
                }

                if (((String)cmbOriginDCId.getSelectedItem()).equalsIgnoreCase(
                                (String)cmbDestDCId.getSelectedItem()) ) {
                        showErrorMessage("Origin DC Id, and the Destination DC Id cannot be the
same.");
                        return false;
                }

                return true;
        }

        private void createPackage() {
                Properties props = new Properties();
                props.setProperty(Package.PACKAGE_ID, txtPackageId.getText());
                props.setProperty(Package.PACKAGE_CONTENT_ID, txtContentId.getText());
                props.setProperty(Package.PACKAGE_CONTENT_DESC, txtContentDesc.getText());
                props.setProperty(Package.PACKAGE_ORIGIN_DC_ID,
(String)cmbOriginDCId.getSelectedItem());
                props.setProperty(Package.PACKAGE_DESTINATION_DC_ID,
(String)cmbDestDCId.getSelectedItem());

                try {
                        PackageFactory.getInstance().createPackage(props);
                        JOptionPane.showMessageDialog(this,

        "Package created successfully.",

        "Package Creator : Success Message",
        JOptionPane.INFORMATION_MESSAGE);
                } catch (Exception e) {
                        showErrorMessage("Error in creating Package : " + e.getMessage());
                }
        }

        private void clearAllFields() {
                txtPackageId.setText("");
                txtContentId.setText("");
                txtContentDesc.setText("");
                cmbOriginDCId.setSelectedIndex(0);
                cmbDestDCId.setSelectedIndex(0);
        }
        private void showErrorMessage(String errorMessage) {
                JOptionPane.showMessageDialog(this, errorMessage,
                                                                        "Package Creator : Error
message",
        JOptionPane.ERROR_MESSAGE);
        }

}
```

Java Program 12: PackageFactory.java

```
import java.util.Properties;
import java.util.Vector;

/**
 * This class is a package creating factory.
 * This will probably be used by an GUI to create
 * packages.
 */
public class PackageFactory {
	// Facotry instance
	private static PackageFactory theInstance;

	// Member variables
	private Vector allPackages;


	// Constructor
	private PackageFactory() {
		theInstance = this;
		allPackages = new Vector();
	}

	// Factory methods
	public static void createInstance() {
		if (theInstance == null) {
			Log.log("PackageFactory: theInstance is null.. so creating it anew.",
Log.FULL);
			new PackageFactory();
		}
	}

	// Get the Factory Instance.
	public static PackageFactory getInstance() {
		createInstance();
		return theInstance;
	} // end getInstance

	// Override Object.finalize
	protected void finalize() throws Throwable {
		Log.log("PackageFactory::finalize. Setting theInstance to null", Log.FULL);
		theInstance = null;
	}

	public Package createPackage(Properties properties) {
			Package aPackage = new Package(properties);
			allPackages.add(aPackage);
			return aPackage;
	}

	public Vector getAllPackages() {
		return allPackages;
	}

	public Package getPackageForPackageId(String packageId) {
		for (int i = 0; i < allPackages.size(); i++) {
			if ( ((Package)allPackages.elementAt(i)).getPackageId().equals(packageId)) {
				return (Package)allPackages.elementAt(i);
			}
		}
		return null;
	}

	public void routeCostChanged() {
		for (int i = 0; i < allPackages.size(); i++) {
			Package aPackage = (Package)allPackages.elementAt(i);
			aPackage.getJourney().costChanged();
		}
	}
}
```

Java Program 13: PackagePanel.java

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

/**
 * This is an UI to monitor packages in the system.
 */
public class PackagePanel extends JPanel {
	// Private members
	private JList packagesList;
	private JTextField txtPackageId;
	private JTextField txtContentId;
	private JTextField txtContentDesc;
	private JTextField txtOriginDCId;
	private JTextField txtCurrentHomeDCId;
	private JTextField txtSunkTransportCost;
	private JTextField txtCostFromHomeToDest;
	private JTextField txtTotalJourneyCost;
	private JComboBox cmbDestDCId;
	private JButton cmdRefreshPackagesList;
	private JButton cmdChangeDestination;
	private JButton cmdHop;

	private JTextArea test;

	// Constructor
	public PackagePanel() {
		addComponents();
	}

	private void addComponents() {
		JPanel listPanel = new JPanel(new BorderLayout(10, 10));

		packagesList = new JList();
		packagesList.setPrototypeCellValue("XX-XX-XX-XX-XX-XXXXXX");
		packagesList.setVisibleRowCount(15);
		packagesList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
		packagesList.addMouseListener(new MouseAdapter() {
			public void mouseClicked(MouseEvent me) {
				packageSelected();
			}});
		JScrollPane sPane = new JScrollPane(packagesList);
		listPanel.add(sPane, BorderLayout.CENTER);

		cmdRefreshPackagesList = new JButton("Refresh Packages List");
		cmdRefreshPackagesList.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent ae) {
				refreshPackagesList();
			}});
		listPanel.add(cmdRefreshPackagesList, BorderLayout.SOUTH);

		listPanel.setBorder(BorderFactory.createTitledBorder("All Packages"));
```

```
add(listPanel);


JPanel detailsPanel = new JPanel(new GridLayout(10, 2, 5, 5));

detailsPanel.add(new JLabel("Package Id : "));
txtPackageId = new JTextField(10);
txtPackageId.setEditable(false);
detailsPanel.add(txtPackageId);

detailsPanel.add(new JLabel("Content Id : "));
txtContentId = new JTextField(10);
txtContentId.setEditable(false);
detailsPanel.add(txtContentId);


detailsPanel.add(new JLabel("Content Desc : "));
txtContentDesc = new JTextField(40);
txtContentDesc.setEditable(false);
detailsPanel.add(txtContentDesc);


detailsPanel.add(new JLabel("Origin DC : "));
txtOriginDCId = new JTextField(4);
txtOriginDCId.setEditable(false);
detailsPanel.add(txtOriginDCId);


detailsPanel.add(new JLabel("Current Home DC : "));
txtCurrentHomeDCId = new JTextField(4);
txtCurrentHomeDCId.setEditable(false);
detailsPanel.add(txtCurrentHomeDCId);


detailsPanel.add(new JLabel("Sunk Cost : "));
txtSunkTransportCost = new JTextField(8);
txtSunkTransportCost.setEditable(false);
detailsPanel.add(txtSunkTransportCost);


detailsPanel.add(new JLabel("Cost From Home To Dest : "));
txtCostFromHomeToDest = new JTextField(8);
txtCostFromHomeToDest.setEditable(false);
detailsPanel.add(txtCostFromHomeToDest);


detailsPanel.add(new JLabel("Total Journey Cost : "));
txtTotalJourneyCost = new JTextField(8);
txtTotalJourneyCost.setEditable(false);
detailsPanel.add(txtTotalJourneyCost);


detailsPanel.add(new JLabel("Destination DC : "));
cmbDestDCId = new JComboBox();
initializeDestDCCombo(null, null);
detailsPanel.add(cmbDestDCId);
```

```
		cmdChangeDestination = new JButton("Change Destination");
		cmdChangeDestination.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent ae) {
				destinationChanged();
			}});
		detailsPanel.add(cmdChangeDestination);

		cmdHop = new JButton("Hop");
		cmdHop.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent ae) {
				hop();
			}});
		detailsPanel.add(cmdHop);

		detailsPanel.setBorder(BorderFactory.createTitledBorder("Details of selected Package"));
		add(detailsPanel);

		test = new JTextArea(10, 70);
		JScrollPane sPane1 = new JScrollPane(test);
		sPane1.setBorder(BorderFactory.createTitledBorder("Depiction of selected package's
journey."));
		add(sPane1);
	}

	private void initializeDestDCCombo(String homeDCId, String destDCId) {
		cmbDestDCId.removeAllItems();
		Vector allDCIds = DCFactory.getInstance().getAllDCIds();
		if (homeDCId != null) {
			if (! homeDCId.equalsIgnoreCase(destDCId)) {
				allDCIds.remove(homeDCId);
			}
		}
		for (int i = 0; i < allDCIds.size(); i++) {
			cmbDestDCId.addItem(allDCIds.elementAt(i));
		}
		if (destDCId != null) {
			cmbDestDCId.setSelectedItem(destDCId);
		}
	}

	private void packageSelected() {
		if (packagesList.getSelectedIndex() < 0) {
			clearAllFields();
			return;
		}
		Package aPackage = (Package)packagesList.getSelectedValue();
		txtPackageId.setText(aPackage.getPackageId());
		txtContentId.setText(aPackage.getContentId());
		txtContentDesc.setText(aPackage.getContentDesc());
		txtOriginDCId.setText(aPackage.getOriginDCId());
		txtCurrentHomeDCId.setText(aPackage.getCurrentHomeDCId());
		txtSunkTransportCost.setText(Double.toString(aPackage.getSunkTransportCost()));
		double homeToDestCost = aPackage.getTotalJourneyCostToDestination() -

	aPackage.getSunkTransportCost();
```

```
			txtCostFromHomeToDest.setText(Double.toString(homeToDestCost));

		txtTotalJourneyCost.setText(Double.toString(aPackage.getTotalJourneyCostToDestination()));
			initializeDestDCCombo(aPackage.getCurrentHomeDCId(),
aPackage.getDestinationDCId()) ;
			depictJourney();
		}

		private void depictJourney() {
			test.setText("");
			if (packagesList.getSelectedIndex() < 0) {
				return;
			}
			Package aPackage = (Package)packagesList.getSelectedValue();
			Journey journey = aPackage.getJourney();
			ListIterator soFarIt = journey.getJourneySoFar().listIterator();;
			ListIterator roadAheadIt = journey.getTheRoadAhead().listIterator();

			test.append("So Far = ");

			while(soFarIt.hasNext()) {
				String dc = (String)soFarIt.next();
				test.append(dc + " -> ");
			}
			test.replaceRange("", test.getText().length() - 3, test.getText().length());

			test.append("\nRoad Ahead = ");
			while(roadAheadIt.hasNext()) {
				String dc = (String)roadAheadIt.next();
				test.append(dc + " -> ");
			}
			test.replaceRange("", test.getText().length() - 3, test.getText().length());
		}

		private void destinationChanged() {
			Package aPackage = (Package)packagesList.getSelectedValue();
			aPackage.destinationChanged((String)cmbDestDCId.getSelectedItem());
			packageSelected();
		}

		private void hop() {
			if (packagesList.getSelectedIndex() < 0) {
				JOptionPane.showMessageDialog(this,
										"Please select a package
before hopping it.",
										"Please Select",

		JOptionPane.INFORMATION_MESSAGE);
				return;
			}
			if
(txtCurrentHomeDCId.getText().equalsIgnoreCase((String)cmbDestDCId.getSelectedItem())) {
				JOptionPane.showMessageDialog(this,
										"This package has already
reached the Desination.",
                                                                                "Destination Already
Reached",

        JOptionPane.INFORMATION_MESSAGE);
                }
                ((Package)packagesList.getSelectedValue()).hop();
                packageSelected();
        }

        private void clearAllFields() {
                txtPackageId.setText("");
                txtContentId.setText("");
                txtContentDesc.setText("");
                txtOriginDCId.setText("");
                txtCurrentHomeDCId.setText("");
                txtSunkTransportCost.setText("");
                txtCostFromHomeToDest.setText("");
                txtTotalJourneyCost.setText("");
                initializeDestDCCombo(null, null);
                test.setText("");
        }

        private void refreshPackagesList() {
                Vector allPackages = PackageFactory.getInstance().getAllPackages();
                packagesList.setListData(allPackages);
                packagesList.setSelectedIndex(-1);
                clearAllFields();
                packageSelected();
        }
}
```

Java Program 14: PathVector.java

```
import java.util.*;

public class PathVector {
	private String ownerDCId;
	private String destDCId;
	private Double cost;
	private Vector route;

	// Ctor
	public PathVector(String ownerDCId,
					String destDCId,
					Double cost,
					Vector route) {
		this.ownerDCId = ownerDCId;
		this.destDCId = destDCId;
		this.cost = cost;
		this.route = route;
	}
	public String getOwnerDCId() {
		return ownerDCId;
	}
	public String getDestDCId() {
		return destDCId;
	}
	public Double getCost() {
		return cost;
	}
	public void setCost(Double cost) {
		this.cost = cost;
	}
	public Vector getRoute() {
		route.trimToSize();
		return route;
	}
	public void setRoute(Vector route) {
		this.route = route;
	}
	public String toString() {
		return ("[O - " + ownerDCId +
				", D - " + destDCId +
				", C - " + cost +
				". R - " + route.toString());
	}
}
```

Java Program 15: Route.java

```
/**
 * This class encapsulates a valid Route of the system
 */
public class Route implements Cloneable {
        private String aDCId;
        private String anotherDCId;
        private double cost;

        public void setADCId(String aDCId) {
                this.aDCId = aDCId;
        }
        public String getADCId() {
                return aDCId;
        }
        public void setAnotherDCId(String anotherDCId) {
                this.anotherDCId = anotherDCId;
        }
        public String getAnotherDCId() {
                return anotherDCId;
        }
        public void setCost(double cost) {
                this.cost = cost;
        }
        public double getCost() {
                return cost;
        }
        /**
         * Returns the DC id of another end of the route.
         * given one end of the route
         */
        public String getTheOtherDCId(String dcId) {
                if (dcId.equalsIgnoreCase(aDCId)) {
                        return anotherDCId;
                }
                if (dcId.equalsIgnoreCase(anotherDCId)) {
                        return aDCId;
                }
                return null;
        }
        // Override Object method toStirng
        public String toString() {
                return aDCId + "->" + anotherDCId + "(" + cost + ")";
        }
} // end of class Route
```

Java Program 16: RouteFactory.java

```
import java.io.*;
import java.util.*;

/**
 * This class is a Route Factory, used to create
 * and initialize Routes. Also acts as a registry of
 * Routes
 */
public class RouteFactory {
   public static String ROUTE_FILE_NAME="E:\\Thesis\\Stable\\AllRoutes.txt";

            // private static members
            private static RouteFactory thisInstance;

            // private members
            private Vector allRoutes;

            // Constructor
            private RouteFactory() {
                        allRoutes = new Vector();
                        initialize();
                        thisInstance = this;
            }

            private void initialize() {
                        initializeAllRoutes();
            }

            public static void createInstance() {
                        if (thisInstance == null) {
                                    Log.log("RouteFactory's factory instance is null. So creating new instance.",
Log.FULL);
                                    new RouteFactory();
                        }
            }

            // Override Object.finalize
            protected void finalize() throws Throwable {
                        Log.log("RouteFactory::finalize. Setting theInstance to null", Log.FULL);
                        thisInstance = null;
            }

            // Get the Factory Instance.
            public static RouteFactory getInstance() {
                        createInstance();
                        return thisInstance;
            } // end getInstance

            public int getNumberOfRoutes() {
                        return allRoutes.size();
            }

            private boolean initializeAllRoutes() {
                        try {
```

```
				File inpFile = new File(ROUTE_FILE_NAME);
				LineNumberReader lir = new LineNumberReader(new FileReader(inpFile));
				while(lir.ready()) {
					String line = lir.readLine();
					StringTokenizer st =  new StringTokenizer(line, "|");
					Route r = new Route();
					r.setADCId(st.nextToken());
					r.setAnotherDCId(st.nextToken());
					r.setCost(Float.parseFloat(st.nextToken()));
					if (! doesRouteAlreadyExist(r)) {
						Log.log(lir.getLineNumber() + " : Adding route from " +
r.getADCId() +
								" to " + r.getAnotherDCId() + " with cost " +
r.getCost() + ".", Log.FULL);
						allRoutes.add(r);
					}
				}
				lir.close();
			} catch (IOException ioe) {
				Log.log("IO Error", Log.MIN);
				ioe.printStackTrace();
				System.exit(-1);
			}
			Log.log("Initialized " + allRoutes.size() + " routes.", Log.MIN);
			allRoutes.trimToSize();
			return true;
		}

		private boolean doesRouteAlreadyExist(Route r) {
			Route existingRoute = getRouteBetween(r.getADCId(), r.getAnotherDCId());
			if (existingRoute != null) {
				return true;
			} else {
				return false;
			}
		}

		public Route getRouteBetween(String dc1Id, String dc2Id) {
			for (int i = 0; i < allRoutes.size(); i++) {
				Route r = (Route)allRoutes.elementAt(i);
				if (r.getADCId().equalsIgnoreCase(dc1Id) &&
					r.getAnotherDCId().equalsIgnoreCase(dc2Id)) {
						return r;
				}
				if (r.getAnotherDCId().equalsIgnoreCase(dc1Id) &&
					r.getADCId().equalsIgnoreCase(dc2Id)) {
						return r;
				}
			}
			return null;
		}

		public void costChanged(Route aRoute, double newCost) {
			Log.log("Changing route cost from " + aRoute.getADCId() +
					" to " + aRoute.getAnotherDCId() + " from " + aRoute.getCost() + " to
" + newCost + ".", Log.MIN);
```

```
			boolean costIncreased = false;
			if (aRoute.getCost() < newCost) {
				costIncreased = true;
			}
			aRoute.setCost(newCost);

			DCFactory.getInstance().routeCostChanged();

			// Wait for some time before asking the Packages to
			// recalculate their journey.. to allow for the
			// propogation to take place..
			try {
				Thread.sleep(10 * 1000);
			} catch (Exception e) {}

			PackageFactory.getInstance().routeCostChanged();
		}

	}
```

Java Program 17: RoutePanel.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

/**
 * This class presents an UI to monitor / manipulate Routes,
 * and their costs.
 */
public class RoutePanel extends JPanel {
	// Private members
	private JComboBox cmbOriginDC;
	private JList lstDestDC;
	private JTextField txtCost;
	private JButton cmdChangeCost;
	private Route selectedRoute;

	private JTextField rDep;

	public RoutePanel() {
		selectedRoute = null;
		addComponents();
		originDCSelected();
	}

	private void addComponents() {
		Vector allDCIds = DCFactory.getInstance().getAllDCIds();
		cmbOriginDC = new JComboBox(allDCIds);
		cmbOriginDC.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent ae) {
				originDCSelected();
			}});
		cmbOriginDC.setBorder(BorderFactory.createTitledBorder("Start"));
		add(cmbOriginDC);

		lstDestDC = new JList();
		lstDestDC.setPrototypeCellValue("DC23  ");
		lstDestDC.setVisibleRowCount(10);
		lstDestDC.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
		lstDestDC.addMouseListener(new MouseAdapter() {
			public void mouseClicked(MouseEvent me) {
				destDCSelected();
			}});
		JScrollPane sPane = new JScrollPane(lstDestDC);
		sPane.setBorder(BorderFactory.createTitledBorder("End"));
		add(sPane);

		rDep = new JTextField(20);
		rDep.setEditable(false);
		add(rDep);

		add(new JLabel("Cost between selected start and end DCs : "));
		txtCost = new JTextField(8);
		add(txtCost);
```

```
			cmdChangeCost = new JButton("Change Cost");
			cmdChangeCost.addActionListener(new ActionListener() {
				public void actionPerformed(ActionEvent ae) {
					costChanged();
				}});
			add(cmdChangeCost);
		}

		private void originDCSelected() {
			Vector neighbours = DCFactory.getInstance().getNeighboursForDC(
									(String)cmbOriginDC.getSelectedItem());
			lstDestDC.setListData(neighbours);
		}

		private void destDCSelected() {
			rDep.setText("");
			txtCost.setText("");
			selectedRoute = null;
			selectedRoute =
RouteFactory.getInstance().getRouteBetween((String)cmbOriginDC.getSelectedItem(),
						(String)lstDestDC.getSelectedValue());
			if (selectedRoute != null) {
				txtCost.setText(String.valueOf(selectedRoute.getCost()));
				rDep.setText(selectedRoute.toString());
			}
		}

		private void costChanged() {
			double cost;
			try {
				cost = Double.parseDouble(txtCost.getText());
			} catch (NumberFormatException nfe) {
				JOptionPane.showMessageDialog(this,
														"Cost is
not a valid number.",
														"Data
error.",

		JOptionPane.ERROR_MESSAGE);
				return;
			}
			JDialog plWait = pleaseWait(this, "Updating Cost. Please wait..");
			plWait.show();
			RouteFactory.getInstance().costChanged(selectedRoute, cost);
			destDCSelected();
			plWait.dispose();
			JOptionPane.showMessageDialog(this,
												"Cost changed.",
												"Cost Change.",

		JOptionPane.INFORMATION_MESSAGE);
		}

		private JDialog pleaseWait(Component parent, String message) {
			JOptionPane wait = new JOptionPane(message,
JOptionPane.INFORMATION_MESSAGE);
			JDialog dialog = wait.createDialog(parent, message);
			dialog.setModal(false);
			return dialog;
		}
}
```

Java Program 18: RoutingManager.java

```
package edu.ncsu.ie.gt.ksg;

import java.util.Vector;
/**
 * Runs in a separate thread and propogates the
 * routing table of all DCs.
 *
 * Can be made to speed up its schedule, if desired.
 * This feature is used when a cost is changed.
 *
 */
public class RoutingManager implements Runnable {
	// private static members
	/**
	 * normal interval in seconds between successive
	 * propogations of routing tables.
	 * suggested value = 30
	 */
	private static int NORMAL_INTERVAL = 10;
	/**
	 * speed interval in seconds between successive
	 * propogations of routing tables.
	 * suggested value = 0
	 */
	private static int SPEED_INTERVAL = 0;
	/**
	 * to count propogations.. used for debugging
	 */
	private static int PROP_COUNT = 0;
	/**
	 * the number of propogations needed initially to
	 * stabilize the routing tables.
	 * It is 2 * E,  where E is the number of edges (routes)
	 *
	 */
	private static int INIT_PROP = 2 * RouteFactory.getInstance().getNumberOfRoutes();
	/**
	 * if true, normal interval is used between successive
	 * propogations, else speed interval is used.
	 */
	private boolean normalRun = true;
	/**
	 * whether to force cost increases or not.
	 */
	private boolean force;

	//private members
	private DC dc;


	// Ctor
	public RoutingManager(DC dc) {
		this.dc = dc;
		this.force = false;
		// start with a speed run, till initial propogation is done,
		// and then continue with  normal propogation
		setNormalRun(false);
	}

	public void costChanged(boolean force) {
		setNormalRun(false);
		setForce(force);
	}

	private synchronized void setNormalRun(boolean normal) {
		normalRun = normal;
	}

	private synchronized boolean isNormalRun() {
		return normalRun;
	}

	private synchronized void setForce(boolean force) {
		this.force = force;
	}

	private synchronized boolean isForce() {
		return force;
	}

	private synchronized void incrementPropCount() {
		PROP_COUNT++;
	}

	private synchronized int getPropCount() {
		return PROP_COUNT;
	}

	private int getInitialPropogationCount() {
		return INIT_PROP;
	}

	public void run() {
		while(true) {
			Log.log("Propogating Path Vector for " + dc.getId() + ".", Log.FULL);
			try {
				dc.propogatePV(isForce());
				incrementPropCount();
				if (isNormalRun()) {
					Thread.sleep(NORMAL_INTERVAL * 1000);
				} else {
					Thread.sleep(SPEED_INTERVAL * 1000);
					if (getPropCount() >= getInitialPropogationCount()) {
						// If initial propogation is done, then stop speed run
						// and do normal run
						setNormalRun(true);
					}
				}
			} catch (Exception e) {
```

```
                    System.err.println("Exception e : " + e.getMessage());
                    System.err.println("Exiting with error");
                    System.exit(-1);
                }
            }
        }
    }
```

Java Program 19: RoutingMonitor.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class RoutingMonitor extends JPanel implements Runnable {
		public void run() {
		}
}
```

Java Program 20: RoutingTable.java

```
import java.util.*;

public class RoutingTable {
	// static members
	public static final Double INFINITY = new Double(2000);
	public static final Double ZERO = new Double(0);

	// private members
	private String ownerDCId;
	private Hashtable pathVectorsTable;

	// Ctor
	public RoutingTable(String ownerDCId, Hashtable costToNeighbours) {
		Log.log("Routing table ctor : " + ownerDCId + " - " + costToNeighbours.toString(),
Log.FULL);
		this.ownerDCId = ownerDCId;
		this.pathVectorsTable = new Hashtable();

		initializePathVectorsTable(costToNeighbours);

	}

	private void initializePathVectorsTable(Hashtable costToNeighbours) {
		Vector allDCIds = DCFactory.getInstance().getAllDCIds();
		for (int i=0; i < allDCIds.size(); i++) {
			String aDCId = (String)allDCIds.elementAt(i);
			Double cost = INFINITY;
			Vector route = new Vector();

			if (costToNeighbours.containsKey(aDCId)) {
				cost = (Double)costToNeighbours.get(aDCId);
				route.add(aDCId);
			}
			if (aDCId.equalsIgnoreCase(ownerDCId)) {
				cost = ZERO;
			}
			PathVector pv = new PathVector(ownerDCId, aDCId, cost, route);
			Log.log("INIT :For owner " + ownerDCId + ", putting pvt for dest " + aDCId +
" as", Log.FULL);
			Log.log(pv.toString(), Log.FULL);
			pathVectorsTable.put(aDCId, pv);

		}
	}

	private void reinitializePathVectorsTable(Hashtable newCostToNeighbours) {
		initializePathVectorsTable(newCostToNeighbours);
	}

	public Hashtable getPathVectorsTable() {
		return pathVectorsTable;
	}

	public PathVector getPathVectorForDestination(String destDCId) {
			return (PathVector)pathVectorsTable.get(destDCId);
		}

		public void updatePathVector(String fromDCId, Hashtable newPVT, boolean force) {
			Double costToNeighbour = getCostToNeighbouringDC(fromDCId);
			Enumeration destDCIds = newPVT.keys();
			while (destDCIds.hasMoreElements()) {
				String destDCId = (String)destDCIds.nextElement();
				if (destDCId.equalsIgnoreCase(ownerDCId)) {
					// for this the cost would have been already set either by
					// initialiazation
					continue;
				}
				PathVector neighbourPV = (PathVector)newPVT.get(destDCId);
				PathVector thisPV = getPathVectorForDestination(destDCId);
				double newCost = neighbourPV.getCost().doubleValue() +
costToNeighbour.doubleValue();
				Log.log("PROP :For owner " + ownerDCId + ", from neighbour " + fromDCId
+
							", putting pvt for dest " + destDCId + " as",
Log.FULL);
				if (! isThisDCInNeighboursRouteToDestination(ownerDCId, neighbourPV)) {
					Log.log("OC - " + thisPV.getCost().doubleValue() + ", NC - " +
							newCost + ", force - " + force, Log.FULL);
					if (newCost < thisPV.getCost().doubleValue()) {

						Log.log("\tFor dest DC " + destDCId, Log.FULL);
						Log.log("\t..changing cost from " +
									thisPV.getCost().doubleValue() + "
to " + newCost, Log.FULL);

						thisPV.setCost(new Double(newCost));
						Vector routeToNeighbour =
((PathVector)pathVectorsTable.get(fromDCId)).getRoute();
						Vector neighboursRoute = neighbourPV.getRoute();
						Vector newRoute = new Vector();
						newRoute.addAll(routeToNeighbour);
						newRoute.addAll(neighboursRoute);

						Log.log("\t..changing route from " +
thisPV.getRoute().toString() +
								" to " + newRoute.toString(), Log.FULL);

						thisPV.setRoute(newRoute);
					}
				}
				Log.log(thisPV.toString(), Log.FULL);
				pathVectorsTable.put(destDCId, thisPV);
			}
		}

		public Double getCostToDC(String dcId) {
			PathVector pv = (PathVector)pathVectorsTable.get(dcId);
			return pv.getCost();
		}
```

```
	public Hashtable getCostToNeighbours() {
		Vector neighbours = DCFactory.getInstance().getNeighboursForDC(ownerDCId);
		Hashtable costToNeighbours = new Hashtable();
		for (int i = 0; i < neighbours.size(); i++) {
			String neighboursDCId = (String)neighbours.elementAt(i);
			costToNeighbours.put(neighboursDCId,
getCostToNeighbouringDC(neighboursDCId));
		}
		return costToNeighbours;
	}


	public Double getCostToNeighbouringDC(String neighbourDCId) {
		// Later .. throw an exception if neighbourDCId is not really a neighbour
		return getCostToDC(neighbourDCId);
	}

	//public void setCostToNeighbouringDC(String neighbourDCId, Double newCost) {
	//		// Later .. throw an exception if neighbourDCId is not really a neighbour
	//		PathVector pv = (PathVector)pathVectorsTable.get(neighbourDCId);
	//		pv.setCost(newCost);
	//		pathVectorsTable.put(neighbourDCId, pv);
	//}

	public void reinitialize(Hashtable newCostToNeighbours) {
		reinitializePathVectorsTable(newCostToNeighbours);
	}

	private boolean isThisDCInNeighboursRouteToDestination(String ownerDCId,

			PathVector neighboursPV) {
		Vector tRoute = neighboursPV.getRoute();
		Log.log("In neighbour " + neighboursPV.getOwnerDCId() + "'s route to dest, owner DC
"
					+ ownerDCId + " is " +
					((tRoute.contains(ownerDCId))?" present.":" not present"),
					Log.FULL);
		return tRoute.contains(ownerDCId);
	}

	/**
	 * Used mainly for reporting purposes in this Version
	 */
	public Hashtable getCostToAllDCs() {
		Hashtable rTable = new Hashtable();
		Enumeration dcs = pathVectorsTable.keys();
		while(dcs.hasMoreElements()) {
			String dc = (String)dcs.nextElement();
			rTable.put(dc, getCostToDC(dc));
		}
		return rTable;
	}

	/**
	 * For journey determination..
	 */
	public String getNextHopToDC(String destDCId) {
		PathVector pv = (PathVector)pathVectorsTable.get(destDCId);
		Vector route = pv.getRoute();
		if (route.size() > 0) {
			// The first element is the next hop of the route.
			Log.log("At DC " + ownerDCId + ", getting next hop to DC " +
					destDCId + " as .. " + (String)route.elementAt(0), Log.FULL);
			return (String)route.elementAt(0);
		} else {
			Log.log("At DC " + ownerDCId + ", next hop to DC " + destDCId + " is not yet
defined.", Log.FULL);
			return "";
		}
	}


	// Backward compatibility - Warning ! dont use this method
	public Hashtable getNextHopToAllDCs() {
		return new Hashtable();
	}
}
```

APPENDIX B

INPUT DATA FILE 1: AllRoutes.txt

```
DC1|DC2|1
DC1|DC6|2
DC1|DC10|3
DC2|DC1|1
DC2|DC9|4
DC2|DC11|5
DC2|DC29|6
DC3|DC4|7
DC4|DC3|7
DC4|DC5|8
DC4|DC7|9
DC5|DC4|8
DC5|DC6|10
DC5|DC8|11
DC6|DC1|2
DC6|DC5|10
DC7|DC4|9
DC7|DC8|12
DC7|DC24|13
DC7|DC25|14
DC8|DC5|11
DC8|DC7|12
DC8|DC9|15
DC8|DC10|16
DC9|DC2|4
DC9|DC8|15
DC9|DC10|17
DC9|DC11|18
DC9|DC25|19
DC9|DC26|20
DC10|DC1|3
DC10|DC8|16
DC10|DC9|17
DC11|DC2|5
DC11|DC9|18
DC11|DC28|21
DC11|DC29|22
DC12|DC13|23
DC12|DC14|24
DC12|DC28|25
DC12|DC36|26
DC13|DC12|23
```

```
DC13|DC14|27
DC14|DC12|24
DC14|DC13|27
DC14|DC29|28
DC15|DC16|29
DC15|DC31|30
DC16|DC15|29
DC16|DC31|31
DC17|DC22|32
DC17|DC23|33
DC17|DC30|34
DC18|DC19|35
DC18|DC20|36
DC18|DC22|37
DC18|DC35|38
DC19|DC18|35
DC19|DC20|39
DC19|DC21|40
DC19|DC35|41
DC20|DC18|36
DC20|DC19|39
DC20|DC25|42
DC20|DC26|43
DC21|DC19|40
DC21|DC26|44
DC21|DC27|45
DC22|DC17|32
DC22|DC18|37
DC22|DC30|46
DC23|DC17|33
DC23|DC24|47
DC23|DC25|48
DC24|DC7|13
DC24|DC23|47
DC24|DC25|49
DC25|DC7|47
DC25|DC9|19
DC25|DC20|42
DC25|DC23|48
DC25|DC24|49
DC25|DC26|50
DC26|DC9|20
DC26|DC20|43
DC26|DC21|44
DC26|DC25|50
DC27|DC21|45
```

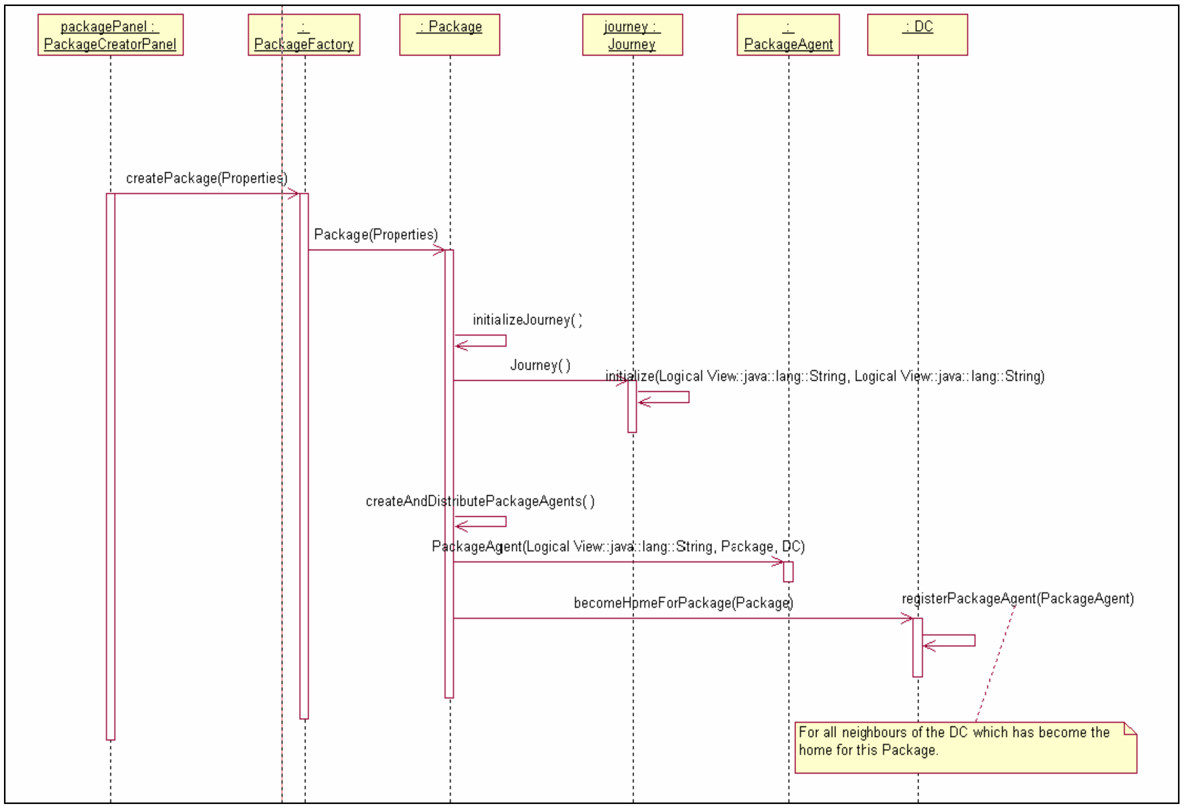
```
DC27|DC28|51
DC27|DC35|52
DC28|DC11|21
DC28|DC12|25
DC28|DC27|51
DC28|DC29|53
DC29|DC2|6
DC29|DC11|22
DC29|DC14|28
DC29|DC28|53
DC30|DC17|34
DC30|DC22|46
DC30|DC31|54
DC30|DC33|55
DC30|DC34|56
DC31|DC15|30
DC31|DC16|31
DC31|DC30|54
DC31|DC32|57
DC32|DC31|57
DC32|DC33|58
DC33|DC30|55
DC33|DC32|58
DC33|DC35|59
DC33|DC36|60
DC34|DC30|56
DC35|DC18|38
DC35|DC19|41
DC35|DC27|52
DC35|DC33|59
DC35|DC36|61
DC36|DC12|26
DC36|DC33|60
DC36|DC35|61
```

INPUT DATA FILE 2: AllNeighbours.txt

```
DC1|DC2|DC6|DC10|
DC2|DC1|DC9|DC11|DC29|
DC3|DC4|
DC4|DC3|DC5|DC7|
DC5|DC4|DC6|DC8|
DC6|DC1|DC5|
DC7|DC4|DC8|DC24|DC25|
DC8|DC5|DC7|DC9|DC10|
DC9|DC2|DC8|DC10|DC11|DC25|DC26|
DC10|DC1|DC8|DC9|
DC11|DC2|DC9|DC28|DC29|
DC12|DC13|DC14|DC28|DC36|
DC13|DC12|DC14|
DC14|DC12|DC13|DC29|
DC15|DC16|DC31|
DC16|DC15|DC31|
DC17|DC22|DC23|DC30|
DC18|DC19|DC20|DC22|DC35|
DC19|DC18|DC20|DC21|DC35|
DC20|DC18|DC19|DC25|DC26|
DC21|DC19|DC26|DC27|
DC22|DC17|DC18|DC30|
DC23|DC17|DC24|DC25|
DC24|DC7|DC23|DC25|
DC25|DC7|DC9|DC20|DC23|DC24|DC26|
DC26|DC9|DC20|DC21|DC25|
DC27|DC21|DC28|DC35|
DC28|DC11|DC12|DC27|DC29|
DC29|DC2|DC11|DC14|DC28|
DC30|DC17|DC22|DC31|DC33|DC34|
DC31|DC15|DC16|DC30|DC32|
DC32|DC31|DC33|
DC33|DC30|DC32|DC35|DC36|
DC34|DC30|
DC35|DC18|DC19|DC27|DC33|DC36|
DC36|DC12|DC33|DC35|
```

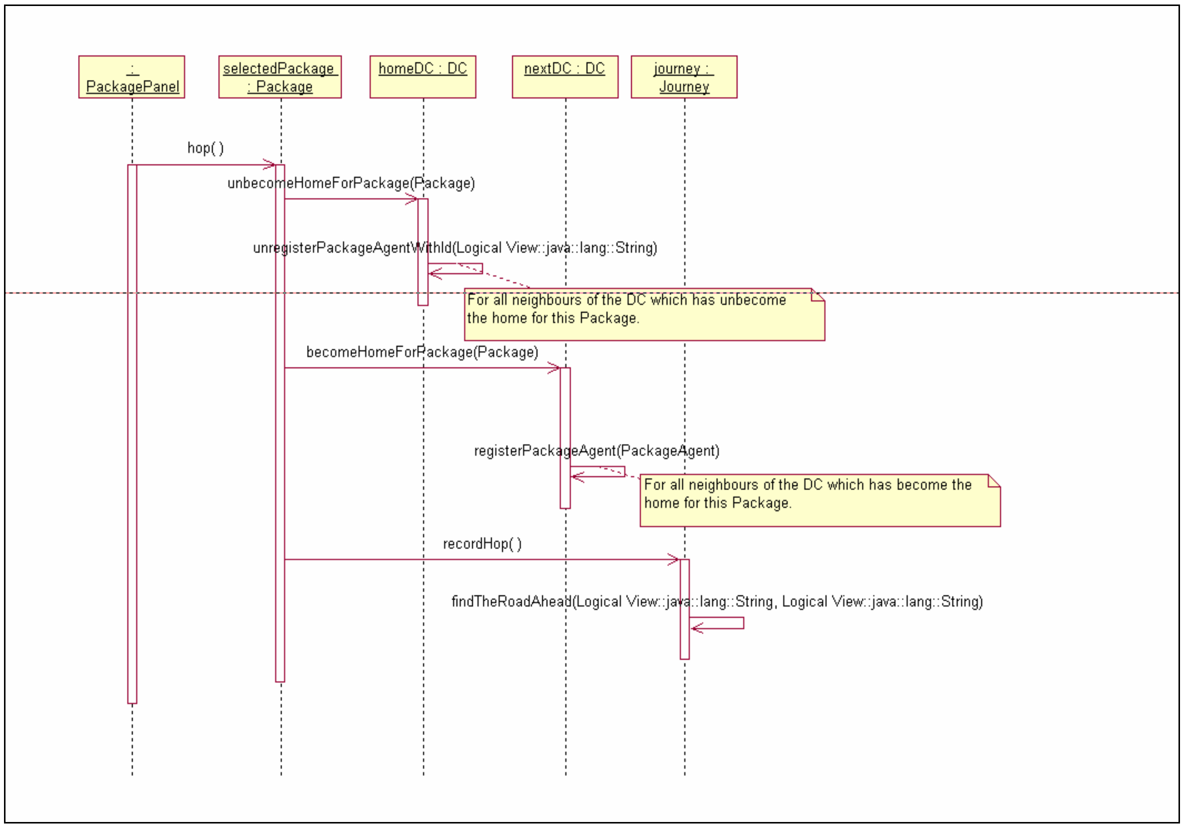
APPENDIX C:

C.1 Package Creation – UML Sequence Diagram

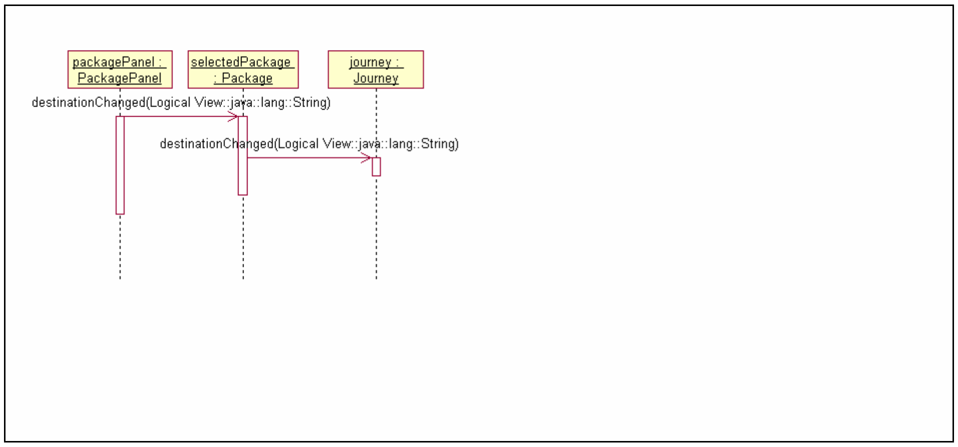


packagePanel : PackageCreatorPanel
: PackageFactory
: Package
journey : Journey
: PackageAgent
: DC
createPackage(Properties)
Package(Properties)
initializeJourney( )
Journey( )
initialize(Logical View::java::lang::String, Logical View::java::lang::String)
createAndDistributePackageAgents( )
PackageAgent(Logical View::java::lang::String, Package, DC)
becomeHomeForPackage(Package)
registerPackageAgent(PackageAgent)
For all neighbours of the DC which has become the home for this Package.

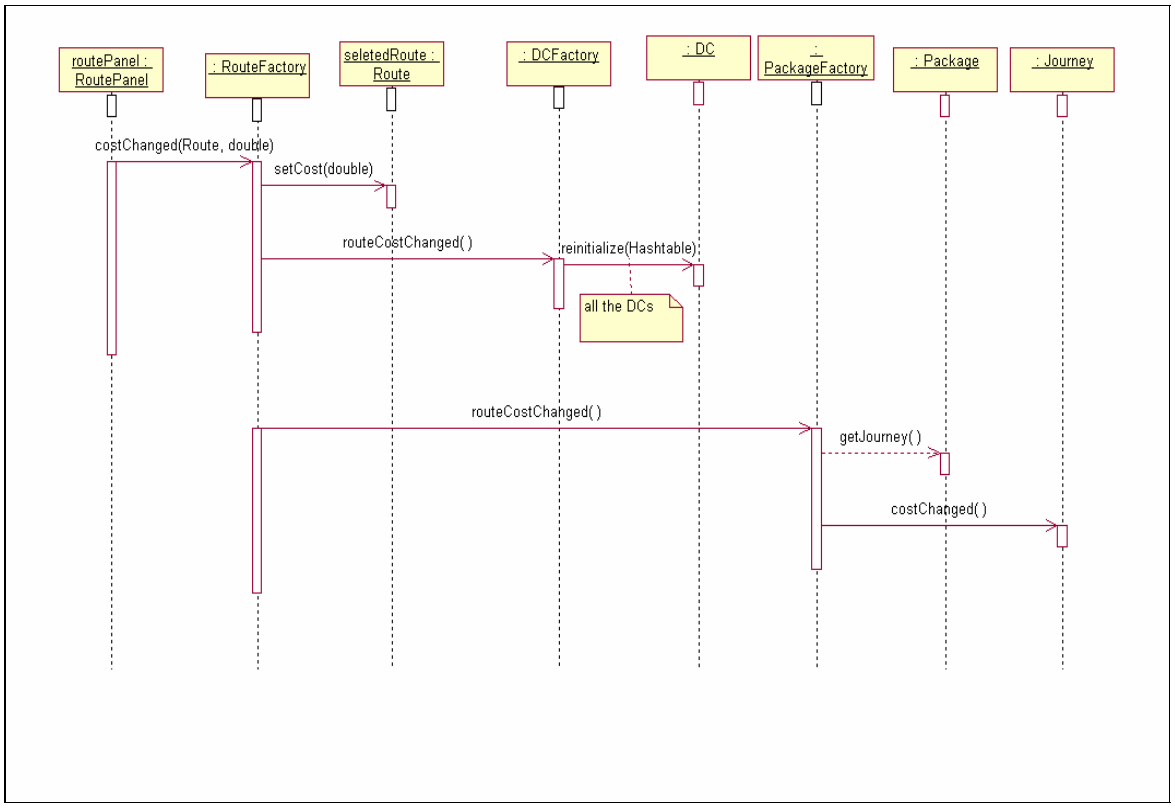
Appendix C.2: Package Hop – UML Sequence Diagram


: PackagePanel
selectedPackage : Package
homeDC : DC
nextDC : DC
journey : Journey
hop( )
unbecomeHomeForPackage(Package)
unregisterPackageAgentWithId(Logical View::java::lang::String)
For all neighbours of the DC which has unbecome the home for this Package.
becomeHomeForPackage(Package)
registerPackageAgent(PackageAgent)
For all neighbours of the DC which has become the home for this Package.
recordHop( )
findTheRoadAhead(Logical View::java::lang::String, Logical View::java::lang::String)

Appendix C.3: Destination Change – UML Sequence Diagram


packagePanel : PackagePanel
selectedPackage : Package
journey : Journey
destinationChanged(Logical View::java::lang::String)
destinationChanged(Logical View::java::lang::String)

Appendix C.4: Route Cost Change – UML Sequence Diagram



REFERENCES:

[1] Dawe, R.L., "The evolving role of the total service provider in the e-business era". (http://www.clm1.org/research/2001/EBusinessCS.pdf)

[2] Steward S., Callaghan J., and Rea T.: 'The eCommerce revolution', BT Technology, 17, No 3, pp 124 - 132 (July 1999).

[3] Owen, M.J., and Nunez-Suarez, J.: 'Agent based solutions for eCommerce', British Telecommunications Eng J, 17, No 4, pp 237—244 (1999)

[4] Kay, M.G., Parlikad, A.N., "Material Flow Analysis of Public Logistics Networks," Proc. International Material Handling Research Colloquium, Portland, ME, June 1-5, 2002.

[5] "The Impact of technology on outsourced logistics" White paper, Yantra Corporation.

[6] Hammant, J., "Information Technology Trends in Logistics," *Logistics Information Management*, Vol 8, No 6, 1996, pp 32–37.

[7] DTD Tutorial (http://www.w3schools.com/dtd/)

[8] Parlikad, A.N., "Performance Analysis of Intelligent Supply Chain Networks," Master's Thesis, Dept. of Industrial Engineering, North Carolina State University, Raleigh, NC, 2002.

[9] Kay, Michael G., "Requirements for package routing in public logistics networks," Tech. Report, Department of Industrial Engineering, North Carolina State University, Raleigh, NC, 2002.

[10] Keshav, S., An Engineering Approach to Computer Networking, Addison Wesley Professional, 1997 (ISBN 0-201-63442-2), Chapter 11: Routing

[11] Kay, M.G., Wilson, J.R., and Seifert, R.W., "Evaluation of AGV Routing Strategies Using Hierarchical Simulation," (ncsu.edu/pub/eos/pub/jwils IJPRV25PS)

[12] Bradshaw, J., "Introduction to Software Agents," *Software Agents*, AAAI Press/The MIT Press, 1997.

[13] Grossner C., Preece, A., Radhakrishan, T., and Newborn, M., (1995) “Sharing Data in multi-agent systems”. (http://www.cs.concordia.ca/~staffcs/cliff/ dai/dai-list.html.)

[14] Glossary of Shipping Terms (http://www.marad.dot.gov/publications/glossary/T.html).

[15] Swarm Development Group, www.swarm.org

[16] BT Intelligent Agent Research (http://www.labs.bt.com/projects/agents/zeus/).

[17] Java Technology & Web Services (http://java.sun.com/webservices/)

[18] Jim Youll, Joan Morris, Raffi Krikorian, and Pattie Maes, “MIT Media Lab” Impulse: Location-based Agent Assistance (agents.www.media.mit.edu/groups/ agents/projects/impulse/)

[19] XML Tutorial (http://www.w3schools.com/xml/)

[20] Simple Object Access Protocol (SOAP) 1.1 (http://www.w3.org/TR/SOAP/#_Toc478383486)

[21] Web Services and UDDI (http://www-3.ibm.com/services/uddi/)

[22] UML Resource Center (http://www.rational.com/uml/index.jsp)

[23] Torsten Heverhagen, Rudolf Tracht, “Negotiation Scenarios between Autonomous Robot Cells in Manufacturing Automation: A Case Study” (http://www.rational.com/media/products/rose/ssd.pdf)