Real Time Inventory Tracking in Distribution Networks

by

Paritosh Somani

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and

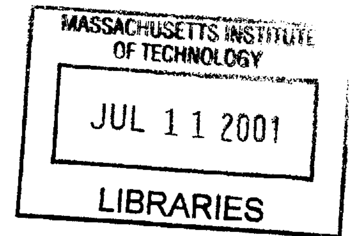
Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2001

BARKER



© MMI Massachusetts Institute of Technology. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 23, 2001

Certified by ....
Kai-Yeung 'Sunny' Siu
Associate Professor of Mechanical Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Real Time Inventory Tracking in Distribution Networks

by
Paritosh Somani

Submitted to the
Department of Electrical Engineering and Computer Science

May 23, 2001

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents a design for the MIT Auto-ID Center tracking server application. This technology is used to track and monitor objects through the supply chain, thereby increasing information visibility and optimizing supply chain management. We propose a distributed network architecture for the tracking servers. To validate results, we implement a simulation using a design analogous to actual servers. Simulation results are provided to justify our claims for using a distributed as opposed to a centralized architecture. Specifically, we examine message complexity and the number of shipments with different distribution network topologies.

Thesis Supervisor: Kai-Yeung Siu
Title: Associate Professor

Acknowledgements

Firstly, I would like to thank Professor Sunny Siu for giving me the chance to work with him. He introduced me to the Auto-ID center and gave me the opportunity to work on a project which was interesting, applicable, and challenging. I would especially like to thank Ph.D. candidate Ching Law for all his time and effort in guiding me during this past year. Without the hundreds of email exchanges and late night meetings, I fear I would have been terribly lost.

I am grateful to all the people at the Auto-ID Center. It was a rewarding experience to be able to work with such a great lab, and I enjoyed the weekly meetings and group retreat. I would especially like to thank Yun Kang for his contributions in developing our simulation model. I am very thankful to Dan Engels, Sanjay Sarma, Sunny Siu, and Kevin Ashton for letting me be a part of the center, and also providing me with funding over the last year.

Above all, I would like to thank my family and friends. Amar, Sneha, Richa, Dharmesh, and Amit have made my years at MIT memorable ones. Last and most of all, I would like to thank Mom, Dad, Ash, Anju, and Neesha for their love and support.

Table of Contents

Chapter 1: Introduction **6**

1.1 MIT Auto-ID Center 7
1.2 Motivation 8
1.2.1 Invoice Adjustments and Shipment Receival 9
1.2.2 Counting of Inventory 9
1.2.3 Inventory Reduction 10
1.3 Contributions of this Thesis 11
1.4 Outline of Subsequent Chapters 12

Chapter 2: Tracking Server **13**

2.1 Architecture 13
2.2 Tracking Algorithms 17
2.2.1 Locate 17
2.2.2 Track 17
2.3 Implementation 19
2.3.1 Storing Data 20
2.3.2 Message Passing 21
2.3.3 Querying Algorithms 22

Chapter 3: Simulation Model **23**

3.1 simjava 24
3.1.1 Simulation Tools 24
3.1.2 simjava Design 25
3.1.3 Testing simjava 26
3.2 Supply Chain Design 28
3.2.1 Nodes 28
3.2.2 Topology 30
3.2.3 Object Flow 30
3.3 Implementation 31
3.3.1 Storage 33
3.3.2 Message Passing 36
3.3.3 Multiple Query Handling 38
3.3.4 Entities 42
3.3.5 Running the Simulation 45

Chapter 4: Simulation Results and Discussion 48

4.1 Model Parameters 48
4.2 Results and Analysis 51
4.2.1 Simulation Variables 52
4.2.2 Comparison with Centralized Model 53
4.2.3 Shipments versus Topology 55
4.2.4 Effects of Depth 58

Chapter 5: Conclusion and Future Work 62

5.1 Increasing Tracking Server Functionality 62
5.1.1 Implementing New Algorithms 63
5.1.2 Caching 64
5.2 Further uses of Tracking Technology 64
5.2.1 Restocking of Store Shelves 65
5.2.2 Finding Items in the Backroom 66
5.2.3 Automated Checkout 66
5.2.4 Theft Prevention 67
5.3 Simulation Extensions 68
5.3.1 Simulation Model 68
5.3.2 Animations 69
5.4 Conclusion 70

References 72

Chapter 1

Introduction

Intelligent tracking technologies have great potential for improvements in the management of supply chain and manufacturing processes. These technologies include global positioning systems (GPS), wireless, and radio frequency identification (RFID), among others. They are discussed in detail in [3] but in this section we only mention ones relevant to this thesis.

The premise behind intelligent tracking technologies is that data can be collected, monitored, and analyzed in an automated fashion. This could lead to many supply chain applications. One application discussed in [3] is postponement transportation whereby a shipment can leave a distribution center without an order. Then, when an order occurs, the shipment can be alter its route to head in that direction – this can save time in shipping items. There are several other applications, many related to tracking object movement and inventory management which are areas the MIT Auto-ID center is interested in.

The Auto-ID center is using RFID technology to make tags for objects and readers that can wirelessly track tagged items. RF technology is already in use in the supply chain by firms such as the United States Postal Service, BMW, and others [3]. The value in RFID is not the technology itself, but the applications that can take advantage of it. Using this technology, the MIT Auto-ID center has built an infrastructure that fosters applications such as tracking servers that will be able to track and monitor objects in the supply chain, increasing chain efficiency through information visibility.

1.1 MIT Auto-ID Center

The MIT Auto-ID center is working to develop the technology to RFID-tag everyday objects and provide an infrastructure where information pertaining to these objects can be tracked. The lab has dissected the infrastructure into three parts: the electronic product code (ePC), the product markup language (PML), and the object naming service (ONS).

The ePC is a numbering scheme used to assign unique identification numbers to each object. The lab has currently proposed a 96-bit numbering scheme with four partitions [4]. The first part is the header, compromised of eight bits, which allows for version control and overhead bits. The next 48 bits are split evenly between a manufacturer identification and a product (SKU) category. The last 40 bits are for *unique* identification of each item. Together, there is room to identify one trillion objects from sixteen million manufacturers and SKUs [12].

PML is a language based off of XML and describes physical objects. It provides a framework for building layers of specific data about objects [5]. It is still being developed at the lab and the specification is being finalized. It will have both static data

and dynamic data capabilities to give general information about the object as well the location, temperature and other dynamic variables.

The ONS is analogous and based on the Domain Name Services (DNS) of the internet. It can take an ePC code as an input and return a URL of where information about the object is stored. A prototype has been built.

Together, the ePC, PML, and ONS complete the infrastructure the MIT Auto-ID Center is striving to build. One of the most pertinent applications to the sponsors of the lab, and the focus of this thesis, is using the Auto-ID infrastructure to build a mechanism to track objects through the supply chain. This increased information visibility can have a profound effect on supply chain efficiency.

1.2 Motivation

Supply chains are complex distribution networks that are inefficient due to the delay in getting information from one end of the chain to the other. As a result, customers in a store can find that the product they are looking for is out of stock, as is the case 8.2% of the time [3]. Furthermore, there can be an abundance of inventory for items that are not selling quickly. Companies continually face the challenge of optimizing their supply chain operations, and spend 12-15% of their revenue in doing so [8].

There are many uses for auto-identification and object tracking technology within the supply chain infrastructure. This thesis specifically focuses on object tracking. The ability to track objects in real time will provide many applications for cost savings and information visibility. In this chapter, we discuss these applications briefly as described in an interview [10].

1.2.1 Invoice Adjustments and Shipment Receival

One potentially costly area for manufacturers is in discrepancies between a manufacturer and its customers during the shipment process. Disagreements on what was shipped can account for a loss of 5% or even more of a manufacturer's revenue. Once items have been shipped from a manufacturer to a distributor, the paperwork and handling to process the incoming order is lengthy and inefficient. Often times, the arriving objects have to be manually checked against an order list to verify arrival. Furthermore, sometimes each case of objects has to be located and labeled. This is an error-prone process because items can be missed or even mislabeled, thus leading to disagreements between what one party says was shipped and what the other says arrived.

With tracking technology, we believe we can reduce or eliminate this discrepancy. With readers positioned on the shipping vehicles and tracking servers at each point in the node, it would be possible to track a shipment from the manufacturer to the customer in real time. If any objects were lost, it would be possible to pinpoint when and where the loss occurred in the chain. Real time tracking would provide the ability to continuously cross-check the shipment against the shipment record at the manufacturer. Furthermore, the tracking server at the manufacturer can send a message to the distributor to provide an electronic list of the items that should be arriving. Then once the physical shipment arrives, readers can quickly and accurately get a count of the objects and the tracking server can do an automated verification.

1.2.2 Counting of Inventory

Both retailers and distribution centers perform regular counting of SKUs. This is a very time and labor intensive process because it is done manually. At the retailer level each

item needs to be counted; each case needs to be counted at the distributor level. Due to the effort and coordination required to perform such tasks, the counting activities are limited to times when the store or distribution center is closed or not very busy.

Using object tracking and tracking server technology, real time information about inventory level will be available. Each tracking server will be able to monitor its inventory level of items, as well as those at locations in other points of the distribution network. By keeping track of shipments made and received, each server will be able to get a count of the number of items that should be present at a location. Furthermore, by using readers within stores and warehouses, real-time counting of items will be possible so long as all tags are placed close to at least one reader.

1.2.3 Inventory Reduction

Predicting consumer demand is very difficult because of the dynamics of purchasing patterns. As a result of this, stores and distribution centers will often maintain extra inventory in order to be prepared for increases in demand. Keeping this extra inventory means greater space requirements, difficulty in locating items due to larger amounts, and potential losses in perishable items that are kept too long. We believe such problems can be eliminated or reduced in severity by increased visibility of item movement through the use of tracking servers.

Object tracking technology and tracking servers can provide real-time information about object flow and stock levels in the chain to facilitate inventory management. Tracking servers will also be able to give counts of how many items should be at a particular location or where stock levels are low. This will help gauge when new shipments are needed. This real-time information will help reduce manual adjustment of

inventory, and allow both stores and distribution centers to operate at lower levels of inventory while still maintaining high service levels to customers.

1.3 Contributions of this Thesis

Once the auto-identification technology is in place, one of the first applications will be tracking servers. Just as servers with databases, the machines will be placed at each node (such as the manufacturer, distributor, or retailer) in the supply chain. They will use the unique tags on items to monitor objects that flow through the node. In addition, they will also maintain information about the objects and have the capability to communicate with other tracking servers. So when a manufacturer is queried for an item, it can directly contact the distributor's tracking server where this object might have been shipped.

In this paper, we provide a design for the tracking servers. We base our design on [11] and thus use a distributed model for storage of information at each node. This allows for load-balanced algorithms. We provide implementations for the Track and Locate algorithms presented in [11].

Given the complicated supply chain dynamics, we present a simulation to provide analysis of the tracking algorithms. The simulation was built using a Java-based simulation tool. It provides a model of the supply chain equipped with tracking servers to test our technology. The simulation provides a mechanism to monitor the message complexity and time efficiency of queries. We include simulation results from running some basic simulations that give us insight into the advantages and disadvantages of our design.

1.4 Outline of Subsequent Chapters

This thesis presents the architecture for building servers to track objects using the infrastructure built by the MIT Auto-ID center. It also presents a simulation of the supply chain that implements the tracking server technology on which object tracking can be tested. Chapter 2 provides an analysis of the design of the tracking server. An overview of the algorithms to be used is also discussed in this chapter. The simulation is presented in Chapter 3. We discuss how the simulation was designed and implemented. Simulation results, as well as interpretation of those results, are presented in Chapter 4. Chapter 5 gives insight into future work that can be done using the auto-id technology in conjunction with the tracking servers and the simulation. We summarize our contributions in this chapter and end with our conclusions.

Chapter 2

Tracking Server

Tracking servers at each node in the supply chain will allow for real time tracking of items from the manufacturer down to retailers. This will offer the ability to locate and monitor objects as they progress through the chain. To accomplish this, we propose a distributed tag-tracking protocol in order to avoid bottlenecks and extensive search times. This is described in detail in [11]. In this chapter, we discuss the architecture of the tracking server and nodes (by node we mean a point in the supply chain, so a manufacturer, distributor, or retailer), and the algorithms used.

2.1 Architecture

As described in [11], the goal of the tracking server design is to have a tag-tracking protocol that displays good load balancing between nodes, and good time, storage and communication complexities. To achieve this, we propose the storage of two sets of data

at each node: tags which have arrived at the node (the Incoming Set) and tags which have left (the Outgoing Set). The format in which information about the tags (or items) a node has received resembles a row in a database, or record. The algorithms used for object tracking are structured around these fundamental design decisions.

A record may specify many attributes about an item. Such properties include the tag number, the expiration date, the next node in the chain if the record is in the incoming set, the node the tag came from if it is in the outgoing set, and the inventory in all subsequent nodes in the chain. An optimization we made was to store similar items in the same record. This is possible under the assumption that items are numbered sequentially so that each record can hold a range of tags by having a start and end field for each tag number. Then, the rest of the attributes described above would be the same. This scheme allows for great savings in space requirements when large groups of SKUs are numbered sequentially. However, even if they are not, the system still operates, but requires more storage for tag information. For example, if a cereal company was shipping 1000 boxes of the same cereal with the same expiration date and stock number (1000), and the cereal boxes were tagged with sequential numbering, then this could go as one line (see Figure 2-1)[1] in the database.

First Tag	Last Tag	Next	Expiration Date	Stock
1001	2000	"Node1"	2/02/2002	1000

Figure 2-1: Sample record where all tags are sequential.

[1] Each tag number is actually 96 bits, but for simplicity in this figure and all subsequent figures we use smaller numbers.

However, if the tag numbers did not happen to be sequential, and there were no two sequential numbers in the thousand, this would mean we would need 1000 records to store this data. This is shown in Figure 2-2.

First Tag	Last Tag	Next	Expiration Date	Stock
1001	1001	"Node1"	2/02/2002	1000
3021	3021	"Node1"	2/02/2002	1000
3300	3300	"Node1"	2/02/2002	1000
⋮	(996) ⋮	⋮	⋮	⋮
9356	9356	"Node1"	2/02/2002	1000

Figure 2-2: A worst case scenario where no two tags are sequential in number and thus 1000 rows are required for 1000 tags.

Rather than simply keeping which tags are present at a node, we propose to maintain the two sets (incoming and outgoing) to provide a "history" of a tag through the chain. This history is what allows for quick object tracking through recording details of items that have reached and left the node. Because most records that are received at a node will be a range of tags (assuming sequential numbering), it is likely that not all of the tags will be going to the same destination and thus require different records in the outgoing set. An example of this is shown in Figure 2-3. In general, two tags must have identical field values in order to share the same row.

First Tag	Last Tag	Next	Expiration Date	Stock
1001	1300	"Node1"	2/02/2002	1000
1301	1690	"Node2"	2/02/2002	1000
1691	2000	"Node3"	2/02/2002	1000

Figure 2-3: A case where the same SKUs have different destinations and thus require different records for each destination.

Under the storage architecture described, searching for a tag requires looking first for the tag in the incoming set. If it is not there, it means that this tag was never sent to

this node. If the tag is found in the incoming set, then the outgoing set is searched for the tag. If it is not found in the outgoing set, this means the tag has not left this node and the location has been discovered. It is interesting to consider the case when the tag is found in the incoming set, and also in the outgoing set. This implies that either the tag has left the node and gone to the next node in the chain, or it is currently in transit. Figure 2-4 depicts the three most plausible scenarios of a tag's location during a search.

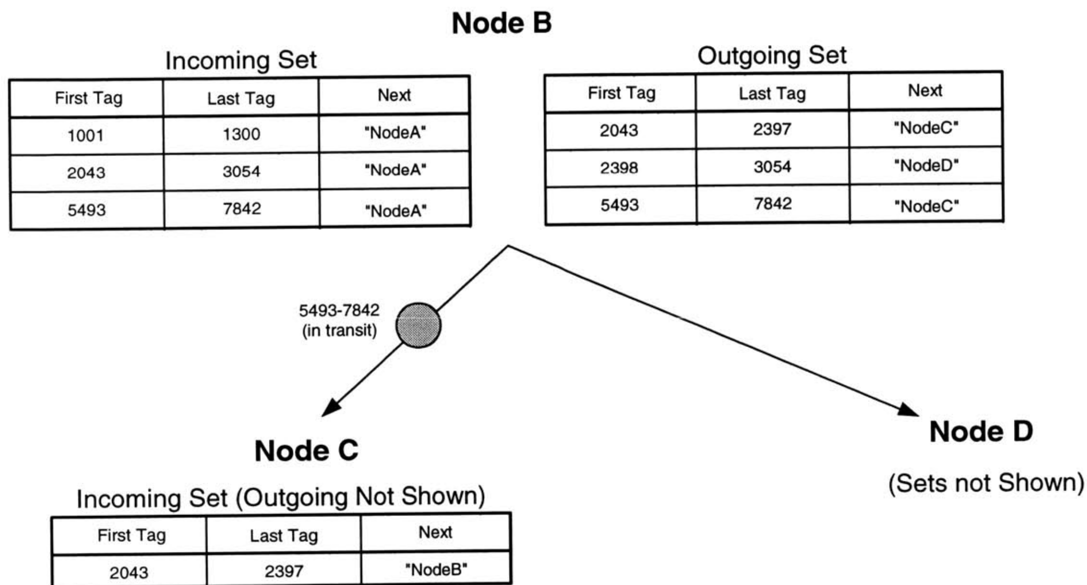


Figure 2-4: An illustration of the three cases for an object's location during a query.

1. Tags 1001-1300 are still at NodeB.
2. Tags 2043-2397 have left NodeB and are at NodeC.
3. Tags 5493-7842 are in transit between NodeB and NodeC.

During transit there is always the possibility that the tag could get lost as well. For simplicity, we do not use a "transit" state and assume that if a tag has left a node, but has not yet been received at the next node, then its location is still the sending node. This can be extended to include the cases where the tag may be lost in transit or other such complex scenarios, in future work. One possible solution may be to use a confirmation of receipt from the receiving node.

2.2 Tracking Algorithms

As discussed in [11], once the tracking server infrastructure is in place, there are many possible searches or tracking functions that companies may want to be able to do. In this thesis, we focus mainly on the track and locate algorithms, although as described in [11], there are several others that will be implemented in tracking servers.

2.2.1 Locate

The locate algorithm is used to find the location of an object. This more formally describes the searching process we discussed earlier. A query is made using thc tag number and the following steps are carried out:

1) Check if the tag is in the incoming set of the current node. If it isn't, return "tag not found," else:
2) Check if the tag is in the outgoing set of the current node. If it is, then find out from the record of the tag where it went next, and call the locate method with the same tag at the next node. If it isn't in the outgoing set:
3) Return self.

Thus, the locate algorithm uses the history maintained by each node in its two sets to locate tags.

2.2.2 Track

The track algorithm is very similar to the locate algorithm. A query is also made using the tag number with the following steps:

1) Check if the tag is in the incoming set of the current node. If it is not, return "tag not found," else:

2) Check if the tag is in the outgoing set of the current node. If it is, then find out from the record of the tag where it went next. Then return "self" *along with* the value returned by calling track at the next node with the same tag. If it is not in the outgoing set:

3) Return self.

Using these steps, this method is able to return a history of the path the tag took to get to its current location. So along with the current location of the tag, this method provides a history of how it got there.

The track and locate algorithms are the two queries that this thesis focuses on. They allow for object tracking through the supply chain that let firms track individual objects. This can also mean keeping track of which objects have been sold or need to be re-shipped. This is where other algorithms that deal with inventory maintenance and expiration are pertinent.

Inventory counting and stock-level monitoring are the focus of other algorithms presented in [11]. Such methods include "total" and "total snapshot" which allow for a measure of the stock level of a product in the chain. To keep information accurate throughout the chain, the usage of update methods to keep stock level information is proposed. In order to insure that storage requirements do not grow indefinitely, an "expire" function is also included, which eliminates records with tags that have left the chain (through the retailer) after a set period of time. This period should be long enough to avoid returns as well as warranty claims, but will still help eliminate old or unneeded records from the nodes' data sets.

The next step will be to look at fault tolerance. Problems such as node failures, corrupted data, and lost items are all possibilities given the complexity of supply chain

processes. Addressing these issues will require additional message passing to do checks and verifications. Features such as time-outs in the networks will be needed to address node and transportation failures.

2.3 Implementation

As mentioned earlier, we built a simulation to model supply chain dynamics using tracking server technology. In order to keep the simulation as realistic as possible, both the simulation and the actual tracking servers share the same basic design, and even code. In this section, we discuss the interfaces and fundamental classes that are shared by the server and the simulation. The implementation of the simulation is also discussed in this thesis, whereas the server implementation is described in [20].

The design of the actual tracking server and the simulation are analogous. We considered doing the simulation and the tracking server independently rather than sharing the same core design. This would have made the simulation easier to implement because we would not need to model a database or a message passing system. But by abstracting these functionalities, we mimicked the tracking server design which offers a more accurate simulation. Furthermore, changes in the tracking servers later on will be more easily adaptable by the simulation.

The entire implementation is done in Java because of its portability and compatibility with other development tools. Both the simulation and tracking server implementations are based around two basic classes and three interfaces. Both use the `Tracking` class, but provide their own implementation for the `DataStorage`, `NetworkServices`, and `Location` interfaces. The fifth structure is the `Record` class which is used for holding information about a sequence of tags. The simulation

uses a subclass of this class because it needs additional functionality that the actual tracking servers do not. The five classes and interfaces reside together in the `track` package. The simulation is composed of this package as well as the remaining simulation implementation which is in a package called `tracksim`. Analagously, the server code is compromised of the `track` package as well as its own, `trackserver` package. The primary difference between the simulation and the actual server is that a large portion of the simulation is simulating the supply chain.

2.3.1 Storing Data

In order to keep the design flexible and easily sharable between the simulation and actual server code, the class responsible for the data management of records was made an interface, called `DataStorage`. It relies on the `Record` class, which is a data structure to hold the elements of a record such as the range of tags and expiration date. The `DataStorage` interface has methods to search the data such as checking if a tag is in a set, and it can also add records to either set when items arrive. The abstraction for a data management class was made to leave flexibility for future changes.

Both the simulation and server rely on the `Record` data structure to hold information about a set of tags. The simulation needs an additional method in the `Record` class for supply chain purposes, so a subclass that serves as the fundamental unit in its data sets is used, emulating a row in the database. Although the tracking server does not need `Records` for the same purpose, it needs them to temporarily hold data when it is pulled out of the database. So whereas the tracking servers use a database [20], the simulation uses the Java `TreeSet` for each set of data (incoming and outgoing).

This is done for quick searching using binary search and we discuss this in detail in Section 3.3.1.

Because the `Record` class is used by both the simulation and the actual tracking servers, the representation of an ePC tag was critical. As described in the specification, the ePC is 96-bit number which is too large for Java primitives. Hence, we use the `BigInteger` class from the `java.math` package (which has no limit on size) to hold tag numbers. Although harder to use with other Java classes, it allows us to hold the starting and ending tags in a sequence of a record, each as one object rather than having to split them into two parts.

2.3.2 Message Passing

Similar to what was done with data storage, message passing or inter-node communication was abstracted to its own interface, `NetworkServices`. By using an interface, the underlying techniques that the nodes in the supply chain will use to electronically communicate with one another can be treated as a black box. The interface has methods to get a node's current location and to send a message to another node. Each of these messages is flagged to signal its type (a track query, a locate query, etc.). Sending messages for the actual servers is done using the Simple Object Access Protocol (SOAP) [20]. For the simulation, sending messages is incorporated into the simulation tool which we will discuss in section 3.3.2.

Since the idea of a location is very different for the simulation and for the physical world, the `Location` interface provides an abstraction mechanism. For the server, a location may be something like a URL [20], whereas for the simulation it is just a code for the simulator to understand (also see Section 3.3.2). Our goal in designing the

tracking server was to use abstraction to enable flexibility in order to allow for changes and extensions in the future. To minimize future changes in both the simulation and the tracking server, we were focused on delivering a design that could both share to foster realism and simplify modifications.

2.3.3 Querying Algorithms

In order to keep the simulation as similar to the actual tracking server operation as possible, both versions share the *same* algorithm implementation, which is in the `Tracking` class. The only other class that they both share (as opposed to interfaces) is the `Record` class. In this fashion, the simulation and tracking server are running off the same algorithms at all times.

In the `Tracking` class, all references are made only to other elements of the `track` package such as the 3 interfaces or the `Record` class. This keeps the actual implementation of the interfaces open and flexible to change. As more algorithms for tracking related functions are implemented, they will simply be added to `Tracking` class on top of the existing design infrastructure.

Chapter 3

Simulation Model

Supply chain management is a complex process given the random and dynamic nature of how items flow through the chain. The Auto-ID center is building a particularly intricate system since the goal is to track not just SKUs but unique items as well. Although there is some level of order to the movements in the chain, supply chain analysis is an unpredictable science. As a result, we believe a simulation can be useful in mimicking the logistics of an actual supply chain to provide insight into how tracking servers will perform.

In order to build a useful simulation, there were two parts we considered. The first was modeling the supply chain to provide a sense of an actual distribution network of products. The second part was to place on top of this framework the tracking server technology in order to test the functionality provided by tracking servers. Both of these were done using a simulation tool that we chose based on applicability to modeling our

problem. In this chapter, we discuss details of the simulation from design to implementation.

3.1 simjava

Before building a model of the supply chain, we were interested in locating a simulation tool that would be useful given our constraints. We considered building our own simulation model, but were able to find several free simulation tools that had the basic modeling functionality we required. Out interest was particularly in Java-based tools – because the actual tracking server is implemented in Java, we wanted the flexibility of sharing code between the server and simulation.

3.1.1 Simulation Tools

Four simulation tools were seriously considered: Ptolemy, JSIM, JavaSim, and simjava. Ptolemy is a versatile package that offers "heterogeneous, concurrent modeling, and design" [15]. It is based on hierarchical graphs, which are sets of entities and relations between the entities. Ptolemy comes with numerous support packages to do advanced math, statistics, modeling, and graphics. However, our conclusion was that it offered more functionality than we needed; because these extra capabilities came with added complexity, we opted to use a simpler tool.

JavaSim is a Java implementation of the C++SIM tool, a popular C++ simulator. We downloaded this and found it easy to learn and use, and fairly extensible. Furthermore, there was a detailed users manual with good documentation [7]. However, JavaSim is based on JDK 1.0.2 and lacks good support. In order to avoid the possibility

of bugs in the tool with no support, we did not use this tool and looked for a more recently developed one.

A positive feature of the simulation tool JSIM is that it offers an "animation environment supporting Web-Based Simulation" [9]. We found this aspect promising and carried out some further investigation. JSIM is built on JDK 1.2 and offers the ability to execute the simulations as either applets or Java Beans. There is also strong technical support for this tool, so JSIM was one of our strongest candidates. Ultimately, however, we found simjava to be easier to use and less complicated while still offering the functionality we needed.

Like the other simulation tools, we tested simjava to learn more about its capabilities and limitations. It is a "process based discrete event simulation package for Java" and is similar to Jade's Sim++ [17]. It also has animation facilities, one of our desired features. simjava seemed like a logical fit for our supply chain simulation because the basic structure of the simulation is entities which communicated with one another through message passing (or in our case, item shipments). Moreover, upon emailing the support list in regards to whether it was a good fit, we received a prompt that was favorable so we decided to use simjava as the tool to build the simulation.

3.1.2 simjava Design

There are three packages that make up the simjava simulation tool: `simjava`, `simanim`, and `simdiag`. `simjava` is the core package that can be used to make stand-alone text-only java simulations. We only use this package and discuss its functionality later. `simdiag` uses JavaBeans to display results of the simulation in a graphical format. The `simanim` works on top of the `simjava` package to produce an

applet displaying an animated simulation. In the future, `simanim` and `simdiag` may be useful for visual simulations. However, because our current use of the simulation is for research and algorithm development only, we solely use the text based package.

simjava is based around having a core set of "simulation foundation" classes that can be used to make discrete event simulation models on top of them [18]. The `simjava` package is targeted at simulations of static networks where nodes are active entities. Communication is executed by sending passive event objects via ports. Although we only provide a brief overview here, the simjava guide provides a more detailed explanation [19].

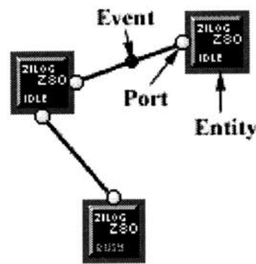


Figure 3-1: The basic simulation architecture [19].

A simulation consists of entities, each running on its own thread. An entity is connected to another one via ports. Using these ports, they can send event objects with specified delays. A static class (`Sim_system`) is used to "control all the threads, advance the simulation time, and maintain the event queues" [19]. During the simulation, a tracefile is created that records the trace messages created by the entities. This file provides a summary of the simulation for analysis.

3.1.3 Testing simjava

We wanted further validation that simjava would be a good tool before starting to implement our supply chain model. We felt this was important because the simulation

would be repeatedly used and extended as development on the tracking server progressed. To do this, we implemented a simulation of the Name-Dropper algorithm presented in [6]. A resource discovery algorithm, it is used for nodes in a network to locate all neighboring machines through randomized message passing.

The purpose of building this simulation was to provide a more rigorous analysis of simjava. After finishing and testing the Name-Dropper, we found simjava easy to use with an architecture that would be a good fit for a supply chain simulation. The tracefiles were particularly useful to provide a summary of the simulation results. Development was also straightforward because of the package based design of the tool. Hence, we started work on the tracking server simulation using simjava. Furthermore, as a result of the positive feedback, the tool was also the basis for another simulation and M.Eng. Thesis in the Auto-ID center [13].

One problem we had with simjava was discovered once we started to run complex and large simulations. Since simjava gives a new process to each thread and each entity runs on its own thread, there is a system resource shortage with simulations of 1000 nodes or more. Our system would crash in such scenarios. We downloaded another version of Java that used "green threads" [2]. Green threads are user level threads so they are all managed by a single process. They do not work with a multiprocessor since a single process will appear as a single thread to the operating system. However, for our purposes they sped up the simulation because of the large number of threads we had in large simulations.

3.2 Supply Chain Design

In order to produce a useful simulation to analyze the algorithms used for the tracking server, it was crucial to make a realistic model of the supply chain. Because every supply chain is different, our goal was to construct a model which was general and flexible enough to simulate real life distribution networks. This meant that designing and implementing the supply chain, rather than the tracker server portion, required the majority of the effort.

The simulation we built is based on a distribution network for a single product in a basic chain. It is extendable to simulate a network with the distribution of more than one item, and also have a more complex chain – this can be the focus of future work. In order to replicate the dynamics of the supply chain, we relied on random-number generation to help introduce variability in the chain. We also made most parts of the chain variable based on user input to keep the simulation easily changeable.

3.2.1 Nodes

In our simulation, we use a node based network topology to mimic the structure of real-life supply chains. We use five basic types of nodes, or entities as we refer to them in order to avoid confusion:

- manufacturer is at the top of the chain, with only one per simulation
- distributors are the next level of the supply chain below manufacturers
- retailers are the end point of the supply chain, where eventually items will leave the distribution network
- the supplier, as we call it, controls the rate of item inflow into the chain

- the client is used to query for objects to test how well tracking servers can respond

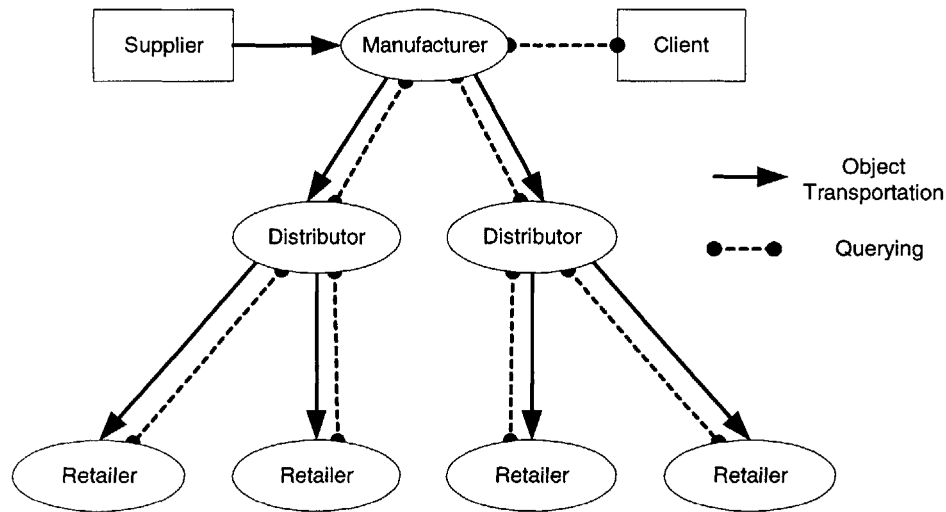


Figure 3-2: An overview of the nodes in the simulation.

Figure 3-2 shows us a basic supply chain and the relations of the five entities we use for the simulation. The supplier is responsible for controlling the rate of entry of items into the chain. The supplier name is misleading because "supplier" has other meanings in a supply chain context. For our purposes, one can think of the supplier as creating the parts the manufacturer needs. The client is responsible for generating queries to track objects. The queries are sent to the manufacturer and it is by analyzing the time to process these queries that we are able to assess the object tracking protocols and system design. The manufacturer, distributor, and retailers are involved in product shipment. Although they basically have the same function, the simulation has different entities for these three in case we want to differentiate their functionality in the future. The retailer is currently different because once received, items do not leave the retailer. However, this will change once product sales are incorporated into the simulation.

3.2.2 Topology

Although we were aiming for a realistic model in the simulation, we did not want to compromise flexibility and simplicity. The purpose of this goal was to have extensibility later on once the simulation needs to be evolved. To build a basic model, we use a chain with only a single item in distribution (hence only one manufacturer) such as a detergent or type of drink. Another assumption used is that no two distributors ship to the same retailer. As a result, each topology generated for the simulation is a tree structure. Both assumptions are extendable in the future if a model with more than one manufacturer/product or cross shipment between distributors to the same retailer is needed.

In order to achieve variability and provide more data points to study, the network topology of the supply chain is generated with two degrees of randomness: both the number of distributors the manufacturer feeds to and the retailers each distributor feeds to are gaussian random variables. We also provide another dimension of variability in the network graph by making the depth of the tree variable. However, for all of our purposes, we use a standard depth of three.

3.2.3 Object Flow

The interesting part of the supply chain is how objects flow. When designing this, we modeled a very straightforward flowing pattern while still trying to have dynamics that one might see in real life. There are two parts to our flowing model: 1) how items enter the chain, and 2) what each node does with the item it receives.

As discussed earlier, the supplier is responsible for controlling the rate of entry of new items into the chain. It does this by determining:

- how often to send a shipment to the manufacturer
- the size/numbering of the shipment

By varying these characteristics, one can create different flow behavior. For simplicity, we keep all tag numbering sequential when shipping items. So in summary, when a supplier sends a shipment, it must decide how many tags to put in the shipment as well as the spacing in numbering between this shipment and the next.

Once a shipment arrives at a node, it is broken up (depending on how many nodes are below it) into pieces and each piece is shipped independently. We assume that every object at the manufacturer and distributor gets shipped so that they all eventually reach the retailers. By varying how long items spend at each node as well as how they are split up before being shipped to the next level in the chain, it helps foster further flow dynamics. The algorithm we use for processing shipments at node arrival is as follows:

1. A shipment of items numbered A to D arrive at Distributor1, which can ship to Retailer1, Retailer2, and Retailer3.
2. Distributor1 chooses B and C randomly (between A and D) so we now have split the package into three parts: A – B, B – C, and C – D.
3. Each group of items is randomly assigned to one of the retailers.
4. Distributor1 ships the three sub-packages after different amounts of time so that the time spent at NodeA varies by each sub-package.

Using this methodology, each package is split up so that eventually all items move down to the retailers. At the retailer level, as mentioned before, no further shipment occurs.

3.3 Implementation

All implementation was done in Java using the simjava simulation tool. As discussed in section 2.3, the `tracksim` package implements both the tracking server interfaces built

in the track package, and the supply chain model needed to complete the simulation. The design of the tracking server is incorporated into the track package whereas the simulation version of the implementation is in tracksim. All of the design and the implementation is incorporated into the tracksim package for the supply chain model. Hence, running the simulation entails using both the track and tracksim packages.

Simulating the tracking server required implementing the interfaces in the track package. So most of the pieces of the track package had a simulation complement in the tracksim package, which is depicted in Figure 3-3.

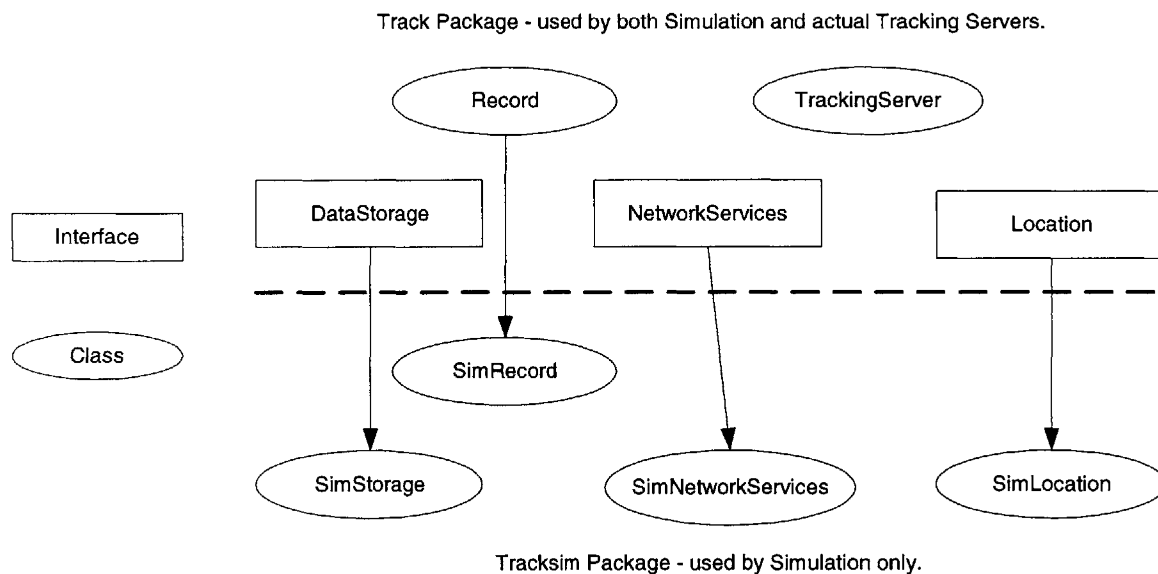


Figure 3-3: An overview of the tracking server components. The tracksim counterparts of the track classes and interfaces are used to implement the simulation.

The simulation counterparts[1] of the track package interfaces provide us with the fundamental components of a tracking server. We are able to integrate this with the supply chain design where each node in the topology has a tracking server to simulate a real-life distribution network with tracking server technology.

[1] It is interesting to note that as discussed in section 2.3, the simulation uses the tracking server algorithms from the track package which are the same ones the actual tracking server uses.

3.3.1 Storage

As show in Figure 3-3, the simulation uses the `SimStorage` class to store data at each node. This data is stored in the `SimRecord` format as opposed to using the `track.Record` class. Each `SimStorage` object, which simulates a database, is comprised of `SimRecords`, which simulate database rows.

To simulate the two sets of data (incoming and outgoing) for each node, we use the Java `TreeSet` object. This was done to provide faster search times by keeping data sorted. But in order to sort, we needed to implement the Java `Comparable` interface with the `Record` class. We did this by sorting on the first tag in each `Record` object. And since `BigInteger`, which is how we represent tags, already implements `Comparable`, it was a matter of extracting this portion from each record being compared.

The `SimStorage` object implements all the basic database functions such as adding new records to sets or searching for tags in a set. It is able to take advantage of the sorting done by the `TreeSet` to do a binary search for tags and minimize search times for tags. This will be useful for large-scale simulations when the amount of data is large.

In the simulation, we mentioned that part of object flow is randomly distributing received packages. In our design, rather than each node implementing the algorithm (discussed in the previous section) to do this, it is taken care of by each record. This allows different types of items to be distributed differently using an interface. But because only the simulation uses this functionality, we created the `SimRecord` class for records in the simulation. `SimRecord` implements the interface for distributing objects.

The `Distributable` interface declares the method used to distribute records. This interface is implemented by `SimRecord`. In order to execute the algorithm for distributing the received shipment, knowledge of the possible recipients of the broken up shipment is required. Using Java's random number function, `SimRecord` can split up the range of tags randomly between the possible recipients.

The last step of distributing a range of tags from a `SimRecord` involves assigning random delays to each new group of tags. To do this, we had two choices:

1. Make a table and keep track of time and delays to know when to send each new shipment out.
2. Use simjava's scheduling and event queue.

Method (2) was much easier to implement and achieved the same effect as (1). So once the objects in a `SimRecord` have been split up into new, sub-packages and been assigned destinations with delays, they are each shipped (using simjava) to the *original* node with the information and various delays. They are flagged as such so that when the node receives these messages, it knows that the delay period has passed and it is time to send the shipment forward. Figure 3-4 goes through the described protocol. The new packages are also stored as `SimRecords` themselves.

One aspect of the protocol not mentioned is dealing with object mutability. Because `SimRecords` are added to the incoming set of a node upon arrival, and to the outgoing set upon leaving, and moreover those records leaving are headed to other nodes' incoming sets, we must be careful to insure that if one field of a record is changed at some point in the process, it does not inadvertently change the same field in another record. For example, consider the case where a `SimRecord` with a sequence of tags arrives at arrives at NodeB from NodeA. In NodeA's outgoing set, the `SimRecord`'s

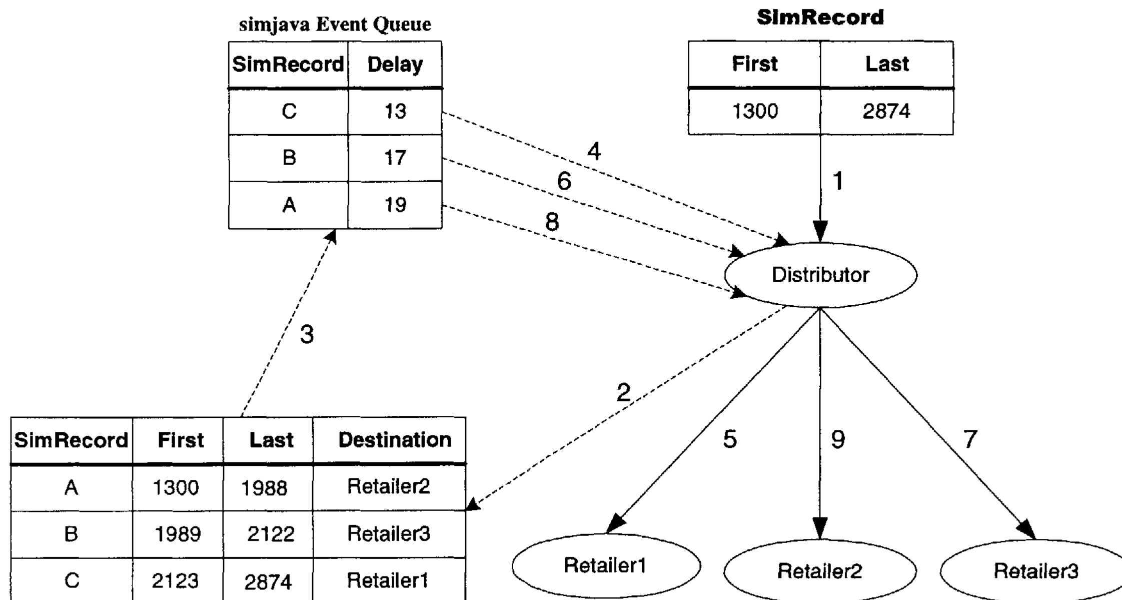


Figure 3-4: An example of the protocol implemented in each `SimRecord` to divide and distribute a received record:

1. A shipment with tags numbered 1300 – 2874 arrives at the Distributor.
2. The Distributor uses the distribute method of `SimRecord` to randomly pick two numbers (1988 and 2122) to split up the shipment into three parts. It randomly assigns the recipients to these parts.
3. Next, random delays are assigned to each new shipment using a gaussian with mean 17 and deviation 3. The new `SimRecords` can now be placed on the event queue using simjava.
4. After 13 simulation units, the first event is taken off the queue and shipped to the Distributor. It is market as a distribution message so the Distributor knows it is not a new shipment.
5. The Distributor immediately ships the package to the destination.
6. After 4 more simulation units, step 4 is repeated with `SimRecord` B.
7. Step 5 is repeated with `SimRecord` B.
8. After 2 more simulation units, step 4 is repeated with `SimRecord` A.
9. Step 5 is repeated with `SimRecord` A.

"next" field will say "NodeB" to signify that the tags were shipped to NodeB. However, NodeB will want to change the same field in the received `SimRecord` to read "NodeA" in its incoming set to indicate the tags came from NodeA. The danger is that by making this change, it will also change NodeA's outgoing set. To address this, we have built a cloning method in the `Record` class (which is called anytime records are shipped) to

insure that the described error is avoided and only copies are sent. The cloning method does a deep clone so that each element of the `Record` is copied rather having elements of the copied `Record` still pointing to the original objects in the original record.

3.3.2 Message Passing

As discussed in section 2.3.2, message passing and inter-node communication is done using simjava. The simulation tool has built in functionality that allows entities to communicate to each other by placing events on the Event Queue. These events can hold objects and can be flagged to indicate the type of message being sent (query vs. shipment, etc.).

In order to place events on the queue, a destination location must be specified. simjava gives each entity an identification number which is simply an integer specifying the order in which the entity was created in the simulation. We use this number to give each node in the simulation a `SimLocation`, which is the simulation's implementation of the `Location` interface discussed earlier. This entity-unique number allows for addressing for inter-node communication using the simjava Event Queue. Entities can place messages for other entities on the queue using their `SimLocation` and add delays to when the messages should be sent (as shown when dividing and distributing a received shipment).

In the simulation there are two instances where an entity passes messages to another entity: 1) one entity is shipping items to another entity, or 2) there is a query from one entity to another. We have already discussed the sequence of steps involved in shipping objects from a node to the next nodes in the supply chain. In this case, each node will directly (using simjava scheduling) send a `SimRecord` to the receiving node.

This is a straightforward process with no steps in between because there was no "real-life" scenario we were trying to model. Once we learn more about how shipping will occur with respect to the tracking server infrastructure, we can model this more accurately in the simulation. However, in case (2), in order to emulate what will happen with the actual tracking servers, we implement the `NetworkServices` interface with a `SimNetworkServices` class that is responsible for inter-node communication.

The `SimNetworkServices` class is responsible for all network-related functions such as maintaining the node's location as well as sending messages. Like the tracking server design, we abstract this functionality into its own class for the sake of flexibility in case the method of communication between nodes changes in the future. This abstraction requires us to use `SimNetworkServices` whenever a node wishes to talk to another node in regards to a query, even though the *actual* message sent will be from the node and not from the `SimNetworkServices` object. It is done this way because only entities in simjava can send messages to other entities via the Event Queue. And since SimNetworkServices is not an entity, it must have access to its parent node's simjava message sending function.

The process for a node to send a query to another node (if the tag being searched for cannot be found at the initial node) becomes a multi-stage process through the use of `SimNetworkServices`. As Figure 3-5 shows, sending an inter-node message requires more steps than sending a shipment. A shipment is sent directly without the use of `SimNetworkServices` because there is no real-life counter part we were trying to simulate in this transaction. However, because with querying, actual tracking servers will be using their `NetworkServices` object for message passing, we use a similar model for the simulation. If we did not, step 5 could have been skipped in the protocol.

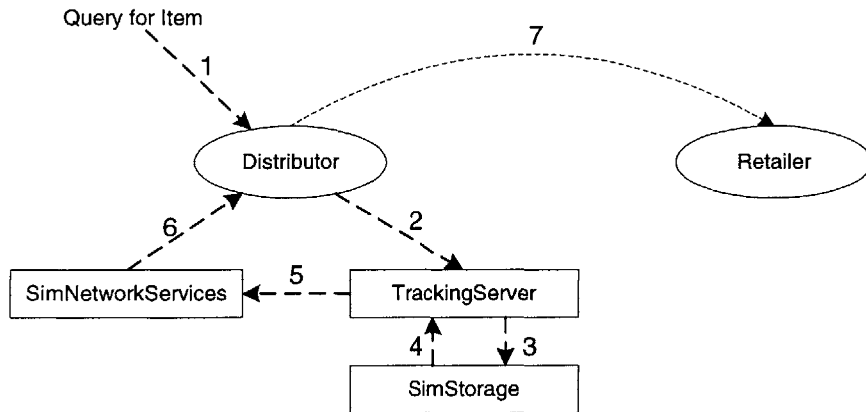


Figure 3-5: An overview of the protocol for redirecting a query.

1. A query for an item that was sent to the `Distributor` arrives.
2. The Distributor send the query to its `Tracking` to process.
3. The `Tracking` checks with node's data sets to see where the item is.
4. The search reveals that the item was shipped to a retailer.
5. `SimNetworkServices` is told the `SimLocation` of where the item was shipped.
6. `SimNetworkServices` accesses the simjava sending function (via Event Queue).
7. A query to the Retailer is sent.

3.3.3 Multiple Query Handling

Thus far, we have presented a simplified model of how querying actually works in the simulation. However, we soon realized we would need a better design to handle multiple queries at a node. The problem is that if an entity does not have the item being queried for, then it must send a query out down the chain. However, while it is waiting for the query result, other events such as shipments or query results for other items can arrive. Deciding on the course of action under such a scenario was a decision that required us to consider several options.

One possible solution was to hold off on processing any of the newly arrived events until the original query returned. However, this might lead to inaccurate

information because new items might arrive at the node, and since it will still be waiting for the query to return and not process the new events, there would be a delay before the node would realize it had received new items. Furthermore, if the query takes a long time or gets lost, it may leave the node hanging, at least until a time-out. Another possible solution we contemplated was that once the query left the parent node, the node would not wait for the response and start processing new events upon arrival. But such a protocol would mean that the query sent would have to be marked and kept track of so that when the response arrived, the node would know how to process it. Keeping track of all sent queries is a disadvantage because of the added memory requirement. Furthermore, to uniquely identify a query would require the item being queried for, the destination, and the *time* it was sent (because of duplicate queries) or at least some unique identifier. Although a possibility, we sought a better solution.

Because queries actually e take minutes at most to execute whereas object transportation takes hours or even days, one approximation we considered for the simulation was to remove any delay in running queries. We could achieve this by making the `Tracking` objects of each node public and provide every node with a list of all other nodes. Then, if a node needed to query another node, it could use its list and just call the query method directly without having to use message passing through simjava. This would mean queries take zero time units to run. However, after further analysis, we came to the conclusion that there were some interesting cases to consider with non-zero querying delay. Situations such as queries getting lost or taking long periods of time, which are real possibilities, would be missed. And because our overall goal in the simulation was to model the real life tracking servers as accurately as possible, we came up with a new design that we believe achieves this goal.

In order to deal with multiple queries at the same time, we simulate multi-threading by allowing nodes to "spawn" off entities that execute queries. We implement this entity with the NodeChild class. A node can spawn as many NodeChilds as it needs, which depends on how many simultaneous queries the node has to run. Hence, each NodeChild represents a thread for the node.

Since simjava does not have a way to eliminate entities once they are created, rather than spawn new ones for each new set of queries, we propose a way to recycle created NodeChilds. Each node maintains a list of the NodeChilds that it has created, and each NodeChild keeps a flag indicating whether it is busy processing a query. So when a new query arrives at a node, the node goes through its list of NodeChilds and if one is free, then assigns the query to it. Otherwise, a new NodeChild is created. This process is shown in Figure 3-6.

Creating an entity during the simulation requires a delay period for the simulation to put the "create" event on the event queue and process it. So in the simulation, we put a delay whenever a new query arrived (whether a new NodeChild has to be created or not). One can think of this delay as the initial overhead in processing a query, which is why we include it even when an existing NodeChild can be used for a query as opposed to requiring a new one. The part of the node that handles the initial query and receiving and sending shipments is implemented with the NodeParent class, and this class also handles the delay to create NodeChild objects. All query processing is done by NodeChilds.

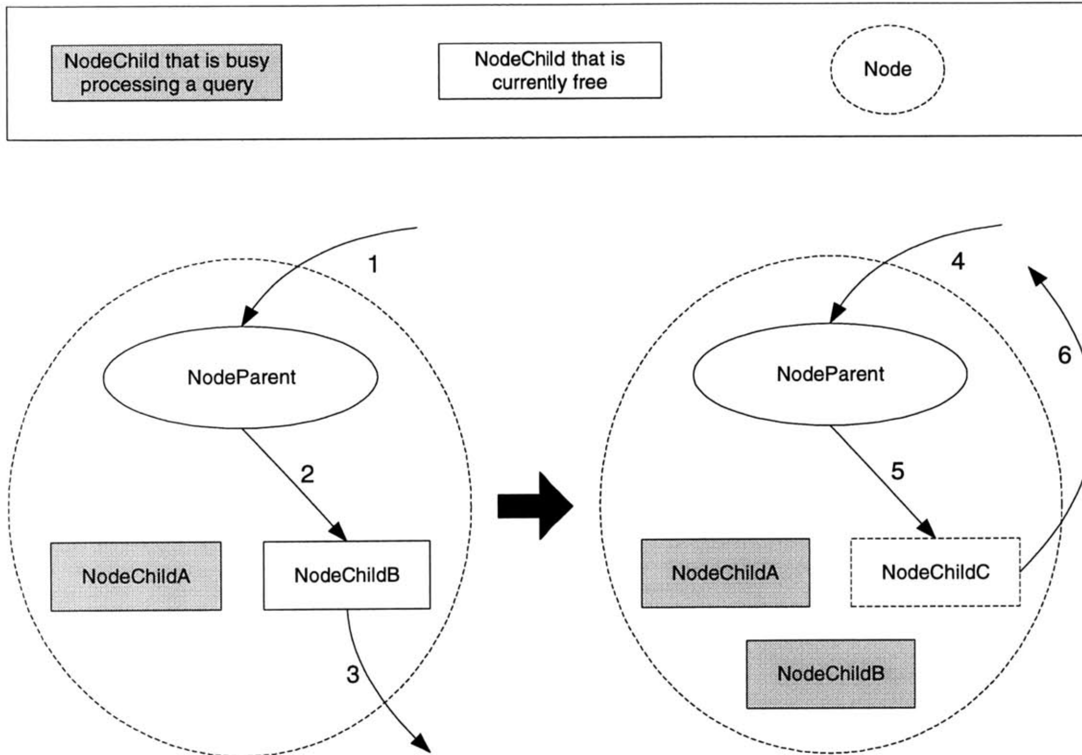


Figure 3-6: An illustration of the use of `NodeChilds` to handle simultaneous queries.

1. A query is sent to the node and the `NodeParent` receives it.
2. The `NodeParent` checks its list of `NodeChilds` and sees the currently `NodeChildA` is busy but `NodeChildB` is free.
3. `NodeChildB` processes the query and sees that the item being searched for has been shipped, so it passes the query to the next location.
4. Another query arrives at the node.
5. Because all of the `NodeChilds` are busy, the `NodeParent` creates a new `NodeChild` and passes the query to it.
6. `NodeChildC` processes the query and is able to find the tag at the current node. Rather than having to go back via the `NodeParent`, it is able to directly send the query response to the original sender.

In order for a `NodeChild` to be able to process a query, it needs to have access to the `NodeParent`'s data. Since all of the `NodeChilds` are using this same data, in addition to the `NodeParent`, there was potential for multiple parties trying to access/modify the data through the `SimStorage` object simultaneously. To address this, we use the Java `synchronize` keyword to make methods in `SimStorage`

synchronized so that only one can be used a time. This way, if the `NodeParent` is trying to add new items to its sets while a `NodeChild` is trying to search for some items in the sets, there will be no discrepancy.

3.3.4 Entities

Until now, we have only focused on entities such as the manufacturer, distributor, and retailer. However, there are other nodes in the simulation that also play key roles. We have been using node and entity interchangeably. However from now on, node will only refer to the `Node` object while entity will refer to any object in the simulation. In this section, we discuss all the entities used for the simulation as well as the architecture used for implementation. Since many of the functions are the same for different entities, we use an inheritance design that allows for sharing of functionality but also flexibility. The overall structure is shown in Figure 3-7.

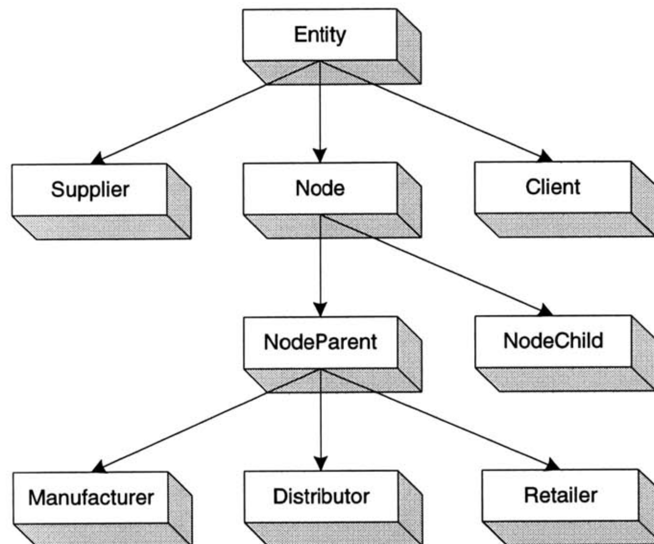


Figure 3-7: A view of the entity architecture.

The fundamental class of the supply chain is the `Node` class. `Node` objects are equipped with the ability to send messages to other entities and have an interface for

processing received events. This basic entity also has a variable to keep track of how many messages it has sent, as well as the SimStorage and Tracking objects for basic operation. It initializes both of these and all other fundamental objects relevant to the simulation. All Manufacturer, Distributor, and Retailer objects are Node objects. However, the Node class is an abstract class so it is implemented differently depending on the different subclasses.

The NodeParent is a subclass of the Node class and holds the basic functionality for the Manufacturer, Distributor, and Retailer objects. As discussed previously, all querying capabilities are abstracted away into the NodeChild class. The NodeParent handles sending shipments, as well as assigning queries to NodeChilds while maintaining a list of them. Furthermore, once the simulation is finished, the NodeParent is responsible for getting a count of all messages sent by the node for analysis purposes. Like the Node class, NodeParent is implemented as an abstract class. It also uses the Java Vector to keep track of NodeChild's because of the dynamic ability to add to a Vector. Each NodeParent goes through this Vector when a query arrives, looking for the first free NodeChild, and only creating a new one if none is found.

The other subclass of the Node class is the NodeChild. As discussed, it is responsible for processing queries. To do this, each NodeChild has access to the data storage object for a particular Node. All NodeChild's of a NodeParent share the same SimStorage object. A flag is maintained specifying whether the NodeChild is busy so the NodeParent can quickly scan its list of NodeChild's to see which ones are free. To process a query, each NodeChild constructs a Tracking object using the track package (the same algorithms for the actual servers). Then, the NodeChild can

process queries and pass the returned value from the query back to the initiator of the query (not the NodeParent, but the actual sender).

Under the current implementation, the Distributor and the Manufacturer are basically the same. In the future, this can be extended to add more specific functionality. Currently, both of them just wait for events to arrive and then use their superclass (NodeParent) processing capabilities. When they receive a shipment of items, they first add the new items to their incoming set of data and then use the distribute method we talked about in Section 3.3.1. Originally, the Manufacturer was also responsible for the inflow of items into the chain. But this was extracted to the a Supplier entity for modularity purposes.

The Retailer class is very similar to the Distributor and Manufacturer classes. It too just waits for events and processes them accordingly. The main difference is in the handling of received shipments. Because we do not implement the selling of items at the retailer level, we assume all items received remain there. In the future, this will be extended to incorporate sales, and match the rate of inflow of items. Returned items will also be taken into account to see how the tracking technology can handle it.

The Supplier class is used to control the rate of inflow of items into the supply chain. Like the Manufacturer, there is only one Supplier in each simulation. So rather than being a member of the distribution network, the Supplier is there to create new shipments that go to the Manufacturer. In some sense, one can think of the Supplier as the one "supplying parts" that the Manufacturer needs to manufacture products. Originally, the Supplier functionality was incorporated into

the Manufacturer but we extracted this and in essence split the Manufacturer into two parts.

The Supplier decides how large a shipment of tags is, which is randomized around inputs for the mean and deviation of the size. It also decides how the numbering works and what the gap in the numbering should be between shipments. This too is randomized in the same fashion using a gaussian random variable. The shipment pattern is easily changeable, but currently we use a model were there is an initial burst of many shipments, which is followed by a constant flow of items at regular intervals.

The Client class is used to provide random queries to simulate querying to actual tracking servers. By calculating the time and number of messages required for queries, we can get a measure of the efficiency of the tracking algorithms, and tracking server architecture. The protocol for querying is easily changeable depending on what one wants to examine. We currently use a model where the Client takes an input for the querying rate. It randomly picks a tag that has been sent to the Manufacturer and queries for this tag. To keep track of what has been sent to the Manufacturer, the Supplier and Client share a list of all items that have been shipped. Every time a new shipment is made, it is added to this list so that the Client always pick from an updated list of shipped items. The Client can be configured to use certain types of queries (right now we only have track and locate). This will be extended in the future.

3.3.5 Running the Simulation

In this section we will focus on how the simulation is started and how it is finished. Starting the simulation requires the initialization of several variables which dictate how

the simulation runs. The results are collected at the end of the simulation. In the next chapter, we discuss in detail which variables are tracked and are included in the results pool.

As we have discussed, the first step of the simulation is the generation of the topology. There are two ways to do this, depending on the depth of the chain. If the depth is greater than two, the node branching factor is computed using the inputs of depth and retailers. It is calculated using the relation:

$$(\text{branching factor})^{\text{depth}} = \text{retailers}$$

Then the same branching factor is applied to each node throughout the chain. For typical supply chains, the depth standard is two. With these chains, we use inputs for the mean and standard deviation of the number of retailers and distributors to make the chain.

Aside from generating distribution network topologies, there are many other inputs. All seventeen, as shown in Figure 3-8, are used to control how the simulation functions:

Variable Category	List of Variables
Generating Topology	- depth of the supply chain - fixed number of retailers (as opposed to using mean and deviation numbers) - mean number of distributors - standard deviation of distributors - mean number of retailers per distributor - standard deviation of retailers
Running the Simulation	- length of the simulation - time for steady state - rate at which queries are made
Shipping Items	- rate at which new items enter the chain - mean number of items in a new shipment - standard deviation of the number of items in a new shipment
Object Flow	- time to ship items between nodes - time for a query between nodes - time to spawn a NodeChild - mean time items spend at the manufacturer or a distributor - standard deviation of the time spend at the manufacturer or distributor

Figure 3-8: A summary of all the inputs for the simulation.

Each variable has a default value (discussed in Chapter 4) but the value can be changed on the command line when starting the simulation. With these inputs, the dynamics of the flow of items can be altered and customized. We rely heavily on using the gaussian random variable to give our simulation a more dynamic nature.

To end the simulation, we use another entity. The `SimManager` takes as an input the length of the simulation, and does nothing for this period. This period is actually the inputted simulation time *plus* the initial delay. This is because the interesting part of the simulation is once it has reached some "steady-state" and is beyond the initialization phase. Once the time has elapsed, the `SimManager` is responsible for ending the simulation. It does this by sending a specially tagged message to all of the entities (including the `Client`) signaling them to stop all activities. The `SimManager` asks all entities for a response with a count of all the analysis variables the entities have been tracking. These variables include the number of messages sent and the number of shipments made. The `Client` keeps track of results of the queries sent during the simulation. Once all of the responses have been received by the `SimManager`, it halts the simulation and tracefiles or output files can be viewed for simulation results.

Chapter 4

Simulation Results and Discussion

The simulation provides a tool to test our infrastructure for bottlenecks and low performance areas. We currently implemented the track and locate algorithms in the simulation. Our results include simulation where these two queries were executed in various fashions with a myriad of topologies. Our goal was to use the results to justify our choices of a distributed model and show where this model is an advantage and where a centralized model may have been better.

In this chapter we discuss in detail the several parameters of the simulation and how they might affect results. We also provide results from running the simulation and our interpretation. Extensions of the simulation are discussed in Section 5.3.

4.1 Model Parameters

There are several parameters in our simulation, both for the supply chain characteristics as well as querying. We have implemented the simulation with default values for all of

these variables, but any of them can be changed at the beginning of our simulation. The default values were chosen after an interview with a supply chain specialist in the MIT Auto-ID center to construct a realistic model. To make changing default values easier, we assigned a two-letter code to each variable to use on the command line as flags to change the value. For example, if the code for time was 'tm', then to change time one would add to the command line, "`-tm newvalue`." In this section, we explain our choices of the default values and the significance of each parameter.

Since the simulation runs by increasing simulation units, we modeled our values to treat each simulation unit as one second. This would also make analysis easier and more intuitive. The following is our calculation of values, with the two digit letter code next to each input in parentheses:

- Depth (dp): The depth refers to the number of levels in the supply chain. For most simulations and supply chains, the structure is a depth of two with the manufacturer at the top, the distributor at the next level, and the retailer at level three. However, we allow for more levels in between the manufacturer and the distributor.

- Retailers (rt): This input is used when the depth desired for the supply chain is greater than two. It is used to specify the number of retailers in the simulation, and along with the depth, the node branching factor can be calculated. The default value was set to 1000 retailers to go along with the average number of retailers we had been using in previous supply chains.

- Distributor Mean (dm): The mean number of distributors is used to center the gaussian for randomly choosing the number of distributors in a simulation. Since all supply chains vary in size and logistics, using our guidance from [10], we made the default value ten distributors.

- Distributor Deviation (dd): The deviation number is used for the standard deviation of the gaussian for randomly choosing the number of distributors. The default value was set to zero because in many simulations it would be useful to know the topology beforehand.

- Retailer Mean (rm): The mean number used to center the gaussian for randomly choosing the number of retailers per distributor in a simulation. Again like the

distributor mean, this was set at a general value of 100 retailers that was appropriate for typical supply chains.

• Retailer Deviation (rd): The deviation is once again analogous to the distributor case. Here again we used a default value of zero for the standard deviation on the number of retailers.

• Transport Delay (td): The transport delay is the amount of time it takes to ship items from one point in the supply chain to another. Based on our research from [10], we wanted to model the transport delay to be between one and two days. Hence the default value we used is 130000 units. This way, since one unit is one second in our model, the value represents 1.5 days.

• Query Delay (qd): The query delay is the amount of time it takes to send a message between two points the supply chain that are one level apart (i.e. Manufacturer to Distributor, or Distributor to Retailer). To keep our model simple and easy to understand, we assumed that each message would take 1 second or 1 unit to send.

• Slave Delay (sd): The slave delay is the amount of time it takes a `NodeParent` to spawn a `NodeChild`. In simjava, in order to add new entities to the simulation dynamically, there has to be a delay before the simulation can process the creation. We use a default value of 0.1 units (0.1 second). However, in order to keep our simulation running on 1.0 unit intervals, anytime a `NodeChild` is created, the next message sent by that node has a delay of 0.9 units instead of 1.0 which restores the cycle back.

• Process Mean (pm): This input is the mean for the gaussian used to center the amount of time it takes to process a shipment, i.e. the amount of time items spend at the Manufacturer and Distributor before moving on. Based on our research, this value can vary greatly but is roughly around a week. Hence the default value used was 600,000 units (6.94 days).

• Process Deviation (pd): As a result of the great variance of the amount of time items spend at points in the supply chain, we wanted to reflect this with the standard deviation used to randomly select this value. We use a default value of 300,000 units (3.47 days).

• Rate Delay (ry): The rate of input of new items into the supply chain is a critical factor of the dynamics of distributions networks. Again, this value is variable and different from chain to chain. We estimated one shipment from the manufacturer to each distributor each day. To do this, the rate delay was set as 86,400 units (1 day).

• Range Mean (rn): This value does not have much significance in the simulation results currently. It is used to center the number of items shipped each day by the

Manufacturer. For now, we have set a default of a million items but this will become more relevant once stock and inventory functionality is implemented in the tracking servers.

• Range Deviation (rv): Like the mean, the standard deviation does not impact any of the current simulation results. We use a default value of 100,000 for the deviation on the items shipped per day.

• Initial Delay (id): To make sure the simulation results we tabulate are not outliers from the "initialization" phase, we use the initial delay as a period to wait before starting to keep track of results. This way, the system can reach a level of "steady-state" before monitoring results. Since we already set the value for time at a node for about a week, we wanted sufficient time for products to reach retailers before looking at our results. The default value we use is 1,500,000 units (about 17 days).

• Simulation Time (st): For now, we run our simulations for about four weeks. This correlates to a default value of 2,500,000 units (about 29 days). Note that the simulation time is *in addition* to the initial delay. So by simulation time, we actually mean the amount of time where data is collected.

• Query Rate (qr): The query rate determines how often queries are sent from the `Client` to the `Manufactuer`. This can be adjusted to stress test servers and see how the system responds to high query rates. The default value has been set to one query every 10,000 units (about 8.5 per day). Note that the querying does not commence until the initial delay has passed.

4.2 Results and Analysis

When deciding what simulations to run, we were interested in several relationships. One of our main goals was to show where our distributed model is more efficient than a centralized architecture. In particular, we focused on the number of messages required by each system. We were also interested in the relation of the number of shipments to the topology structure. Lastly, we examined how querying and shipping were related to the depth of the supply chain.

4.2.1 Simulation Variables

To study simulations, there were five areas of the simulation that were monitored. They were chosen because they provide an insight in to message complexity for querying as well as an indication of the level of object flow. More variables can be added and monitored if needed. Below is a list of the areas along with the number of variables within each area in parentheses:

- Number of Queries (2): The number of queries sent, broken down into track and locate. This is monitored by tracking how many queries the `Client` initiates during a simulation.

- Messages (2): The number of messages sent when processing queries during a simulation. This is broken down by query type (track or locate). We count initial queries from the `Client` to the `Manufacturer` in the messages count, and any further inter-node communication. Note that since we use a default of 1.0 simulation time units per message sent, the number of messages used for a query is the same as the time for a query.

- Shipments (1): The number of shipments during a simulation. This includes each unique shipment from the `Manufacturer` a `Distributor`, or a `Distributor` to a `Retailer`.

- Size (2): The size of the messages shipped during queries for items. Since locate message are all of size one, the total *size* of all locate messages is equivalent to the *number* of locate messages. This is not true with track messages because if an item queried for is at the retailer-level in a supply chain of depth 2, the size of the message returned to the `Distributor` will be 1, to the `Manufacturer` 2, and finally to the `Client` 3.

- Time (2): The simulation time taken to execute queries. This is maintained for both track and locate queries.

These nine variables are outputted at the end of the simulation in the output file.

The results under various querying rates with all other variables at their default value are shown in Figure 4-1. As explained earlier, the number of messages for a query is the same as the time taken to execute a query. Also, the locate messages have a size equivalent to this as well.

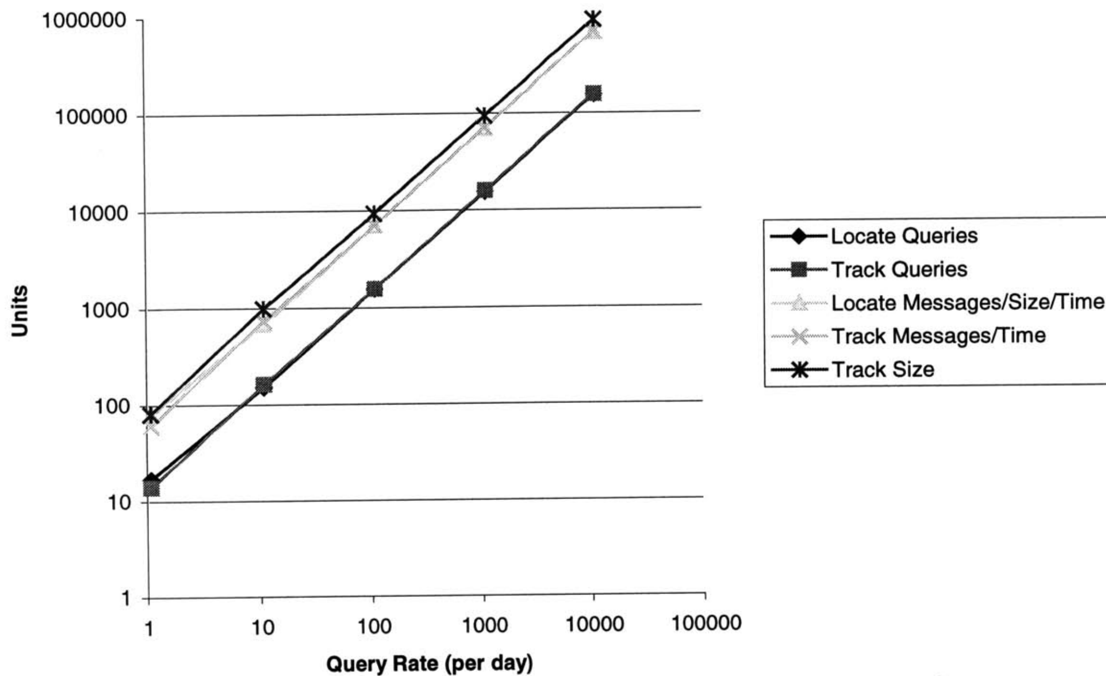


Figure 4-1: A graph of the simulation variables under different querying rates.

As the figure shows, all the variables in question scaled linearly with the query rate. Since the track and locate queries are chosen randomly with equal probability, there is overlap between their plots. This is the case for both the number of track and locate queries as well as the track and locate messages. The main difference is that while the locate queries are all of size 1.0, the track queries vary in size between three (for items at the retailer) and one (for items at the manufacturer).

4.2.2 Comparison with Centralized Model

To justify our distributed architecture, we wanted to compare results with a centralized one. Rather than implementing a simulation with such an architecture, we estimated how many messages would be sent in a centralized model when responding to queries, using our current model. Our estimation was:

$$\text{Messages}_{\text{centralized}} = \text{Shipments} + 2 * (\text{Total Queries})$$

This is because in a centralized model, whenever a shipment is sent, the central database has to be updated. This requires one message to be sent for every shipment. Furthermore, each query would require a message to the central server and one back to the sender. Note that shipments are not included in our count with the distributed model because we assume that when a shipment arrives, part of the receiving process is to update the node's database, and does not require extra messages. Shipments *are* included with the centralized model because the update has to be done remotely and not locally. Figure 4-2 shows a comparison of the two models. The query rate is varied around the range where the messages in two models intersect.

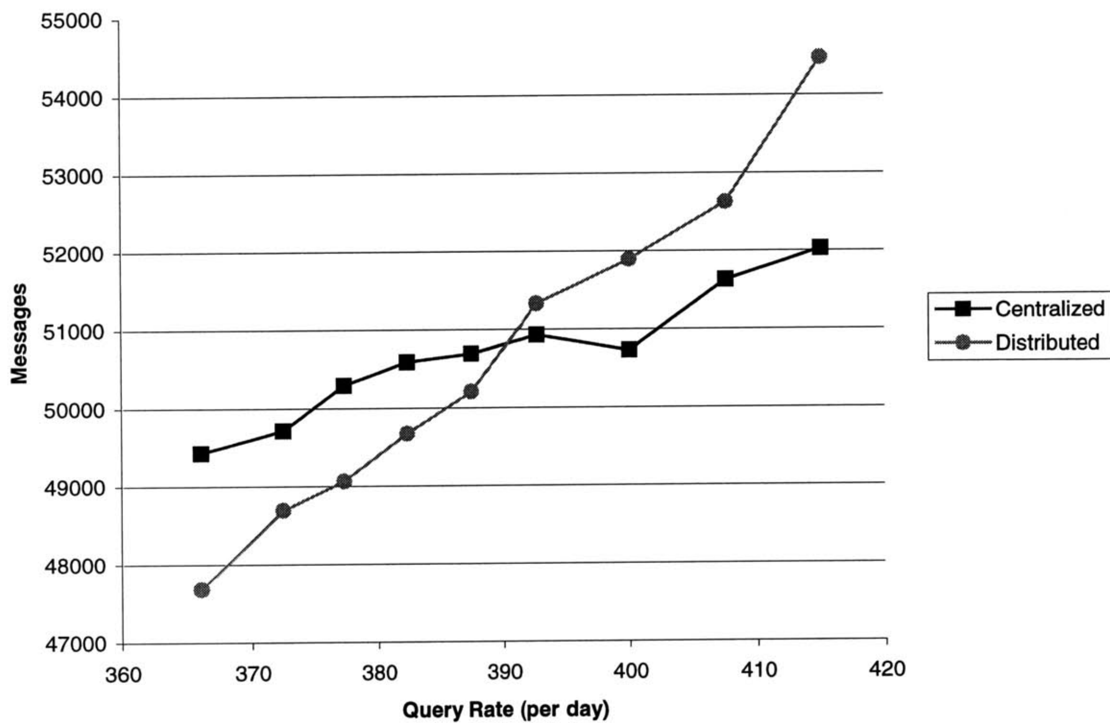


Figure 4-2: A comparison of the number of messages in a centralized architecture versus our distributed model.

As the figure shows, at around the query rate of 390 queries per day, the message complexities in the centralized and distributed models are the same. At rates higher than

this, we find that the centralized model sends fewer messages, while the distributed model does better with query rates below 390. This is because when the query rate gets very high, the overhead of having to update with each shipment becomes less significant relative to the number of queries. The jaggedness in the graphs comes from the randomness of some of the variables. In particular, since the time spent at each node is chosen from a gaussian, there is variability in what portion of the items will be further down the chain as opposed to higher up. And since items at the retailer level require more messages per query than ones at the manufacturer level, this introduces the variability we see in the graph.

Although these results approximate 390 queries/day as the breakpoint for distributed versus centralized model, there are many other factors to consider. For example, our results are based on a shipping model of one shipment per day from the manufacturer to each distributor and a topology of 10 distributors and 100 retailers. However, if the shipping rate is higher or the branching factor in the supply chain is higher, both of these will improve the performance of the distributed model. Furthermore, a distributed model is more robust because there is no dependence on a single server. One must consider these factors as well in evaluating the system architecture.

4.2.3 Shipments versus Topology

One of the relationships we sought to discover through the simulation was that of the number of shipments to the size and topology of the supply chain. To test this, we ran the simulation a number of times while keeping all simulation variables at the input default value, and only varying the size and structure of the topology. The results can be seen in

Figure 4-3. Namely, for supply chains with 5, 10, 15, and 20 distributors, we ran the simulation with 50, 100, 150, and 200 retailers for each distributor configuration. This resulted in topologies of 250 `Retailers` at the low end (5 distributors and 50 retailers per distributor) and 4000 `Retailers` at the high end (20 distributors and 200 retailers per distributor).

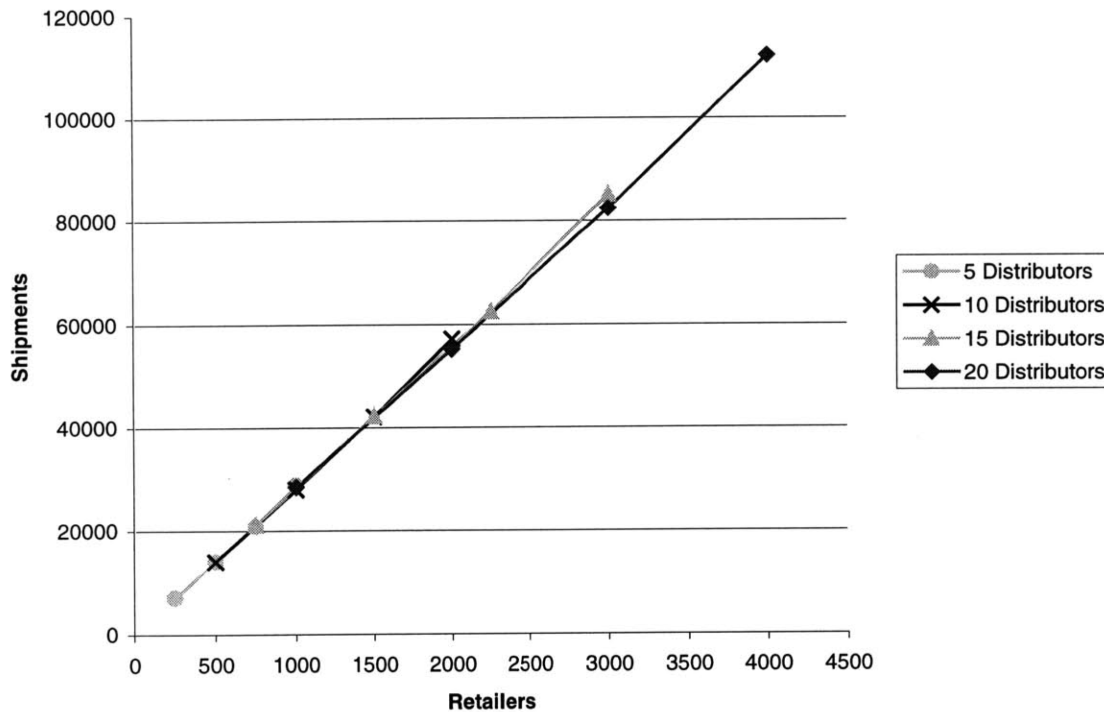


Figure 4-3: A examination of the number of shipments while varying the underlying supply chain topology.

It was surprising to see such a linear relationship between the different topologies. Our conclusion was that in our model, there is a direct correlation between the number of retailers and the number of shipments (so long as the branching factor from each distributor to each retailer is fixed). To confirm this, we fixed the number of retailers for a couple configurations and looked at the number of shipments with varying number of distributors. This is shown in Figure 4-4.

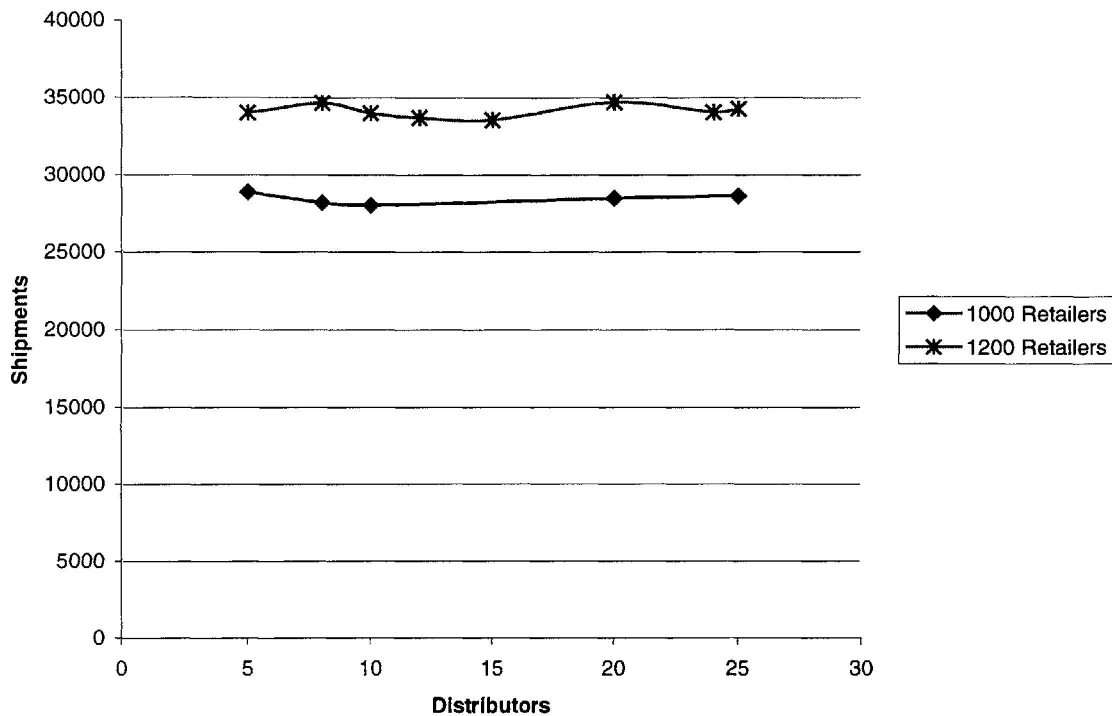


Figure 4-4: An illustration of the number of shipments with fixed retailers.

The figure illustrates that there is a direct relationship between the number of total retailers and shipments. Again, we see a little variability in the graph which is brought out because of the random amount of time items spend at each node. Interestingly enough, the number of shipments in our model is independent of the ratio of distributors to retailers. This implies that supply chains that have high branching factors from the distributors to the retailers have similar shipping complexity to those chains that have lower branching factors but more distributors. However, all our conclusions were derived from results with a fixed depth of two. To investigate this further, we looked at varying the depth of the chain and the behavior of the shipments and query variables.

4.2.4 Effects of Depth

Although we used a model with a manufacturer, distributor and retailer, there are also cases where the number of levels in the supply chain is more than three. This is especially for large retailers and who may have multiple steps in their distribution process. To look at the effect of the depth of the supply chain, we analyzed the number of messages used in querying at two different query rates, again showing this both for our distributed model as well as our estimated centralized model.

To run the simulations, we used default values for all parameters except the query rate and the initial delay. The query rate was altered to provide a comparison between different rates. The other parameter was the initial delay. Since there the default initial delay is about seventeen days, items were not making it to lower levels of deep distribution networks since the average time at each node has a default of about a week. As a result, we made the initial delay a function of the depth of the supply chain so that it would be:

Initial Delay = 900,000 units (or about 10 days) * (supply chain depth)

This gave time for items to fully flood the chain during the initialization period so that once we started monitoring variables for results, the simulation was in a "steady-state." Note that the simulation time (i.e. the time where we actually kept track of result variables) was the same (default value) for all simulations.

Figure 4-5 shows that at a query rate of 108[1] queries per day, the number of messages is always higher under a centralized model, regardless of depth. There is a

---

[1] We used a query rate of 108 per day for ease of calculations. Since there are 86400 seconds in day, we typically used wait times between queries of 80000, 8000, 800, 80 etc. A 800 second wait time between queries corresponds to a query rate of 108 per day, while 80 corresponds to 1080 queries per day.

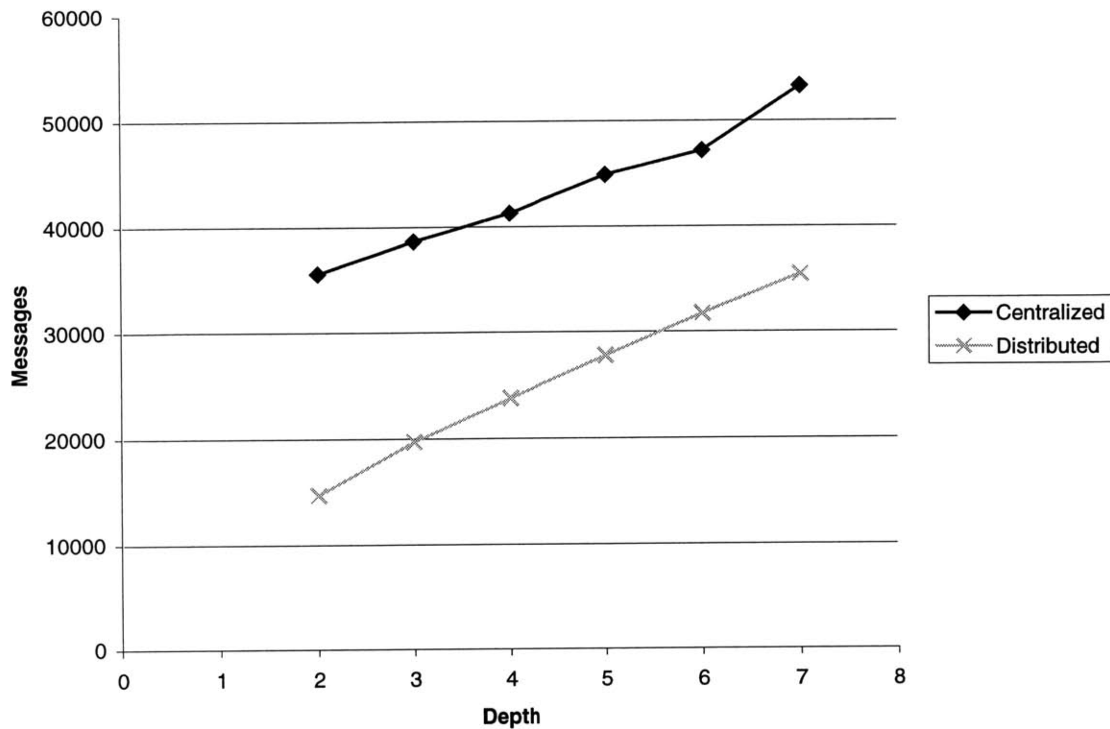


Figure 4-5: A comparison of the number of system messages at a query rate of 108 queries per day.

slight non-linearity in the graph because of the random variables we use in the simulation such as the time objects spend at each node. Furthermore, since the branching factor is calculated from the depth and the number of retailers, and is usually not an integer, there is some added variability in the final size and retailer count in the topology. For example, if the inputted depth is 6 and `Retailers` is the default 1000, the branching factor is computed to be ~3.16. Thus, at each node, we assign the branch as three with probability of 84% and four with probability of 16%. This means the actual number of retailers in the graph may or may not be 1000. As a result, we see some jaggedness in the plots.

The difference in the number of messages between the two models in Figure 4-5 is about constant since the factors that go into the message count all increase linearly with depth. This finding is consistent with our earlier finding that up until around 400 queries

per day, the distributed model performs better in terms of message complexity. However, we have now extended this claim to different supply chain depths as well.

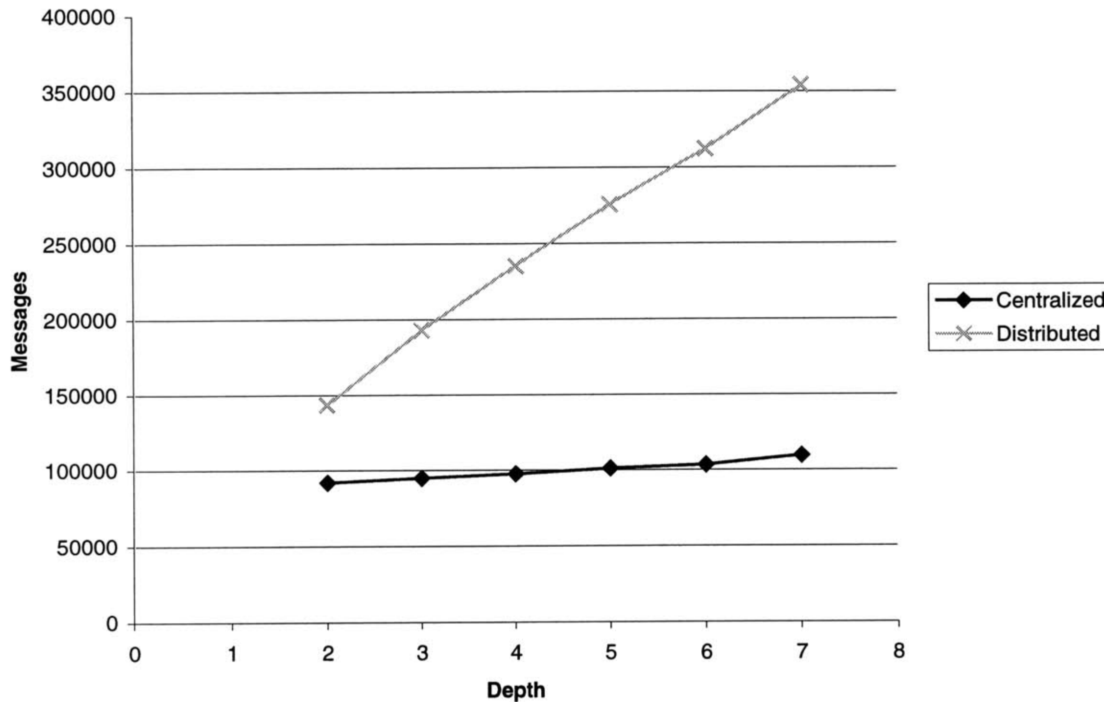


Figure 4-6: A comparison of the number of system messages at a query rate of 1080 queries per day.

Figure 4-6 is similar to the previous figure except that it shows the results of a query rate of 1080 queries per day. As expected, we see that at such a high query rate, the centralized model requires fewer messages. Moreover, as the depth of the supply chain increases, the relative performance of the distributed model erodes further. This is because the effect of having to propagate queries node-to-node in the distributed architecture is much stronger with deep topologies leading to higher message counts.

Finally, we examined the relation between the number of shipments and the depth of the supply chain with our production model. Figure 4-7 shows this to be a roughly linear relationship. The dip in the graph is a result of the variability discussed earlier in this section. Although we ran simulations up to depth seven, in actuality, most supply

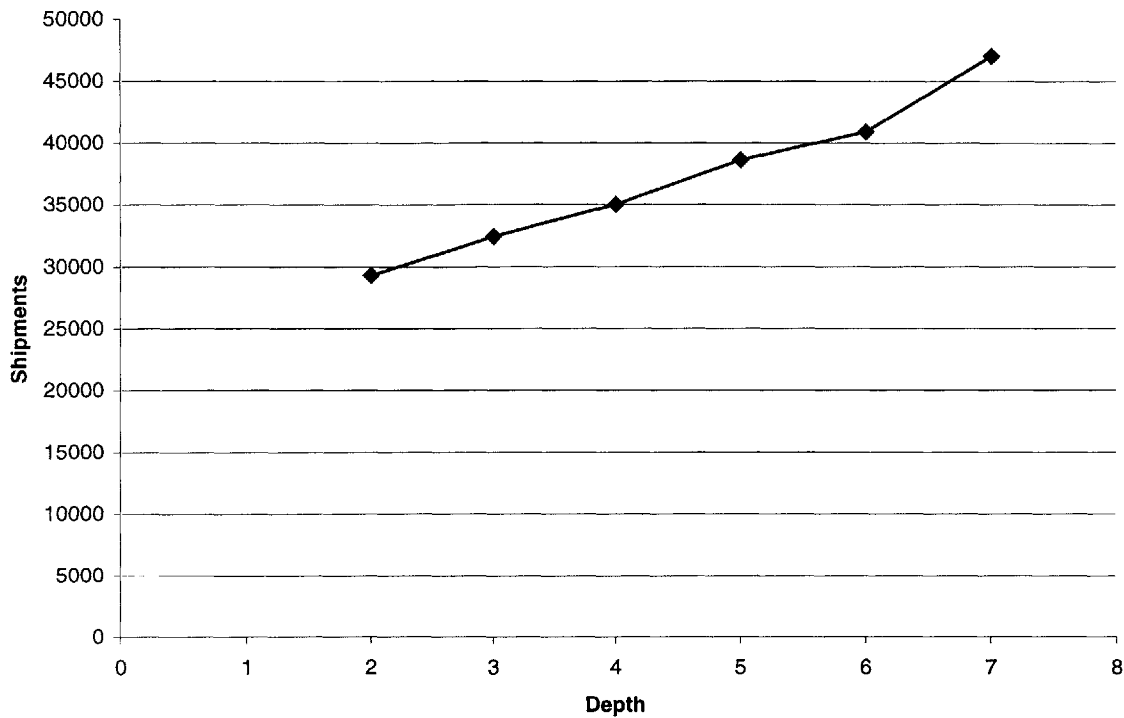


Figure 4-7: An illustration of the effects of varying supply chain depth on the number of shipments.

do not have a depth greater than two or three. The most frequent arrangements according to [10] are:

- Manufacturer Plant, Manufacturer Distribution Center, Retailer Distribution Center, and Retailer Store
- Manufacturer Plant, Retailer Distribution Center, and Retailer Store
- Manufacturer Plant , Third Party Warehouse, Retailer Store

Nonetheless, we were interested in studying these relations at higher depths for added insight into how our model works.

Chapter 5

Conclusion and Future Work

The current implementation of the tracking server is a very basic one that has potential for added functionality to be more useful for vendors in the supply chain. The same is true of the simulation, which is a good template but has room for much to be added. Overall, we believe there are three main areas of extension to our current work:

- Adding functionality to the tracking server
- Increasing the areas of the supply chain where tracking servers are used
- Extending the capabilities of the simulation

Through future research and implementation in these areas, we believe supply chain management can be optimized for maximum efficiency.

5.1 Increasing Tracking Server Functionality

The current design of the tracking server only implements the track and locate algorithms for object tracking. However, with the tracking infrastructure, there are many more useful procedures relevant to supply chain management. As discussed in detail in [11], these are currently being developed. We believe another area of future work will be to

increase the performance of tracking servers through the use of caching. Both of these extensions are discussed in this section.

5.1.1 Implementing New Algorithms

Although we have mentioned the use of tracking servers through the track and locate procedures, much of the value-added from this technology will come from inventory maintenance and monitoring. As such, one of the key capabilities will be the ability to get counts of an SKU. In [11], there are two versions of this method discussed. One provides an estimate of the items in the chain by using each node's stock number which will be updated once a certain number of items have been sold. The other algorithm is more intensive and propagates through the chain getting real time counts of an item at each node.

In order for the inventory counting to work, nodes need to be able to update their "stock" field for each record of items. Furthermore, in order to insure that the storage requirement does not grow indefinitely, we also need a way to remove tags from the system once they have been sold for a predetermined period of time, such as the warranty period. Until then, there is a chance of returns or reentry of the items into the chain. To deal with information accuracy, the retailer initiates updates. To do this, the retailer will update the local count of an SKU which is then propagated up the supply chain so other nodes can have an accurate count. The retailer can also invoke the Expire protocol [11] to remove tags that were sold and whose warranty period is up.

Currently, every time a new query comes to a node in the supply chain, the node does a new search for the tag. This can be a time-intensive process, especially if the item being searched for is at the "bottom" of the supply chain. Caching could be useful if

there were some correlation between queries such that the likelihood of a query for a tag were higher if that same tag had been queried for recently. Then by caching the results of queries, responses for further queries regarding the same item could be returned immediately. However, the cache would have to be cleared frequently because of the possibility that the item may have moved in the chain. Caching is a feature we have started to think about and we believe it may be worth investigating in the future.

5.1.2 Caching

Currently, every time a new query comes to a node in the supply chain, the node does a new search for the tag. This can be a time-intensive process, especially if the item being searched for is at the "bottom" of the supply chain. Caching could be useful if there were some correlation between queries such that the likelihood of a query for a certain tag were higher if that same tag had been queried for recently. Then by caching the results of queries, responses for further queries regarding the same item could be returned immediately. However, the cache would have to be cleared frequently because of the possibility that the item may have moved in the chain. Caching is a feature we have started to think about and believe it should be investigated in the future.

5.2 Further uses of Tracking Technology

In Section 1.2, we mentioned some uses of the object tracking technology for supply chain management. Those uses were based on a design with tracking servers at each node using the object tracking and inventory maintenance procedures we have discussed in this paper. However, there are other ways to improve supply chain operations by doing "in-house" tracking, i.e. by monitoring objects within a node. Some of these

applications are not offshoots of the tracking server technology, but more of the tracking technology in general and the ability to locate objects.

5.2.1 Restocking of Store Shelves

Currently, the process for restocking items on retailer shelves is an inefficient one. According a recent case study [1], on average, 8.2% of the products at supermarkets are out of stock on shelves and 5% of them are not in-store (so ~3% of the time, the items are in the back room inventory). If an item is not re-shelved, there are three possible scenarios: 1) the customer purchases a competitor product, 2) the customer goes to another retailer to find the product he was looking for, or 3) the customer decides not to buy the product he was looking for. In case 1, the manufacturer of the product loses revenue. In case 2, the retailer loses revenue. In case 3, both the manufacturer and the retailer lose revenue.

To detect missing items, a store manager usually uses manual detection. This is a labor-intensive process that may require walking through the isles and using a visual scan to see what needs to be re-shelved. However, this is very error-prone because items may be overlooked and not be detected as low or missing.

The above process can be automated with our technology. With readers on shelves, it would be easy to detect when item stock levels were low. Furthermore, with tracking servers, the manufacturers themselves would be able to use inventory analysis methods to see where inventory was low and what stores might need replenishing of items. This new system would eliminate manual errors and also provide real time information so that the chance of losing revenue, either for the manufacturer or the retailer, would be reduced substantially.

5.2.2 Finding Items in the Backroom

Simply locating items from backroom storage is a problem for retailers. It can be a very time consuming process given the sheer number of items a retailer may sell. Manual sorting of these items in the backroom is error prone, especially due to the constrained space. Even by splitting up the backroom into different types of items, each category is still left with a myriad of items. So finding a specific one from an area remains a challenge for store managers. Furthermore, if an item is accidentally placed into the wrong section, it can mean a very low chance of finding it when needed.

With readers positioned strategically in the backroom, we believe we can reduce the severity of this problem. If items are placed around sectioned readers, each section can have its own "sub-inventory" of items in that area. Then when a store manager is looking for a particular item, a query can be made and each section reader can respond whether it has such an item, and where it is within the section. Such a process would even make it unnecessary for store workers to partition backroom items into different categories. This would lead to time savings and flexibility in searching for SKUs when needed.

5.2.3 Automated Checkout

Currently, retail stores invest a lot of their manpower to staff cash registers. Furthermore, during peak sales periods, lines get very long leading to frustrated customers. In some instances, customers limit their purchasing during these peak periods to be able to use the express lines. This is another area that tag and reader technology will be able to revolutionize in the future.

If readers are placed at each cash register, customers will only need to roll their cart through the designated areas and the readers will scan every item in the cart. The MIT Auto-ID center currently has a demonstration of this technology. This will eliminate the need for individual scanning of items using the barcode system. Furthermore, using a self-checkout debit account, there would be no need for a cashier altogether.

5.2.4 Theft Prevention

Under the barcode system, there is no infrastructure for theft prevention. Additional security devices are being used, and because these devices are too expensive to place onto every single object, firms must randomize placement. Furthermore, there is labor and cost involved in inserting the security tags onto products which can be avoided using the RF tags.

With RFID tags, each item has a unique identity so distinctions can be made between items that have been paid for and those that have not. For example, there can be a security indicator active on every item in the store that gets turned off once the item has been processed at the checkout counter. Then readers placed at store exits can be set to scan for tags with an active indicator, which would spark an alarm.

Another application of theft prevention is related to shelf-based readers. If stores are able to place readers on shelves, in addition to being able to tell when an item needs to be restocked (as mentioned in Section 5.2.1), item movement can be monitored for suspicious activities. For example, if a reader picks up that thirty shavers have been removed from the shelf within seconds, the security department can be notified of this activity. This would allow for store personnel to stop the theft before the culprit leaves.

5.3 Simulation Extensions

The current simulation provides a basic model of the supply chain with very little consideration for system glitches and complicated dynamics. One area of development for the future will be to improve and build on the current simulation model. Another area is to improve the simulation output. Because we are currently using the simulation for research purposes only, we have not been interested in a way to show the simulation in action. But simjava comes with an animation package that can be exploited to produce web-embedded simulations that can be user interactive.

5.3.1 Simulation Model

The simulation model we created is a simple one. To make the model more useful, it can be improved by adding more features that increase the realism of the supply chain movements. Such features include the sale of items, item returns, damaged items, more complicated supply chain, and greater number of products.

By allowing items to leave the retailer, we can simulate sales of goods. This will allow us to test such possibilities as returns where an item leaves but has to re-enter the supply chain. Furthermore, if the item is returned because it is damaged or has a manufacturer defect, that will mean possible up-flow in the supply chain which will create further dynamics. By allowing such features, we will be able to test the use of warranties as a measure of when an item can be removed from the system so that storage does not keep growing indefinitely. The response of new shipments when stock levels drop too low can also be examined. These additions will require the new tracking algorithms in [11] to be implemented.

The current generation of the supply chain topology uses a basic tree architecture which may oversimplify actual operations. To increase the realism of the simulation and thus make the results more meaningful, the topology generation can provide a more complicated graph with overlapping of retailers amongst distributors. Another addition to be developed will be running the simulation for more than one product line. This will mean having more than one manufacturer and will vastly increase the amount of items flowing in the chain, and the storage requirements for each node. We suspect that as this scales, real databases may be required to house node storage information. With the added features discussed in this section, the current simulation can be extended to one that is closer to actual distribution networks.

5.3.2 Animations

In addition to making the simulation model more realistic and complicated, an area of future work will be to exploit the full abilities of the simjava package. One of the main features that may be useful will be to use the simanim package which is a companion package to simjava to convert a simulation into an applet [18]. The animation will allow each entity and port to have their own icons, and connections will be represented by wires. Events traveling between two events will be animated [16]. The animation can also show certain parameters (such as inventory) for each entity during the simulation. The animation will be useful for demonstration purposes for sponsors and to provide a visual understanding of what auto-id technology will mean for supply chain management.

5.4 Conclusion

In this thesis, we have presented a design for tracking servers that will enable firms to track and locate items in the supply chain. The servers sit on top of the infrastructure laid out by the technology developed at the MIT Auto-ID Center. With this application, firms will be able to increase their information visibility in the supply chain to optimize supply chain management.

We have proposed a distributed network architecture for the tracking servers whereby each node in the supply chain keeps track of the items it has seen. We chose this design over a centralized model to decrease message complexity, provide a more robust structure, and to avoid a bottleneck at a single server. Two main algorithms, track and locate, were presented for use with tracking servers. However, there are many more, mostly related to inventory tracking which will automate many of the internal processes that firms experience when handling shipments.

To validate results we build a simulation tool that uses a design analogous to actual tracking servers. This provides the simulation a good replication of how physical tracking servers handle queries and shipping. We researched real-life supply chains to create a realistic logistics model. Using this model, we found that the distributed model performs better in terms of message complexity unless the query rate is very high. Other relationships examined included how the number of shipments changed with the size of the topology, and the ratio of distributors to retailers. We found that these relationships are linear, and attribute this to the supply chain model we used. Changing the object flow of the chain could alter these general characteristics.

We believe we have built a good starting point for research into supply chain dynamics and how tracking servers can improve operations. The simulation is extendable and can be used to study other applications as well. Since supply chains are very dynamic and complex, refining the simulation model to more accurately reflect real-life chains will increase the value of the simulation results. This is the direction we believe further research will be most useful.

References

[1] Andersen Consulting, "Where to Look for Incremental Sales Gains: The Retail Problem of Out-of-Stock Merchandise," Case Study Conducted for the Coca Cola Retailing Research Council.

[2] Blackdown Java Linux, [www.blackdown.org].

[3] Brewer A., Sloan N., Landers T., "Intelligent tracking in manufacturing," *Journal of Intelligent Manufacturing*, pp 245 – 250, 1999.

[4] Brock D., "The Electronic Product Code," MIT Auto-ID Center White Paper, January 2001.

[5] Brock D., "The Physical Markup Language," MIT Auto-ID Center White Paper, February 2001.

[6] Harchol-Balter M., Leighton T., Lewin D., "Resource Discovery in Distributed Networks," MIT Laboratory for Computer Science.

[7] JavaSim Simulation Tool, [http://javasim.ncl.ac.uk/manual/javasim.pdf].

[8] Joshi Y., "Information Visibility and Its Effects on Supply Chain Dynamics," S.M. Thesis, MIT, June 2000.

[9] JSIM Simulation Tool, [http://orion.cs.uga.edu:5080/~jam/jsim/].

[10] Kang Y., Personal Interview, MIT Auto-ID Center Ph.D. Candidate, April 4, 2001.

[11] Law C., "Tagged Object Tracking in Distribution Networks," Working Copy of Research Paper.

[12] Kundapur N., "On Integrating Physical Object with the Internet," S.M. Thesis, MIT, 2000.

[13] Mehta A., "Ad Hoc Network Formation Using Bluetooth Scatternets," M. Eng. Thesis, MIT, 2001.

[14] MIT Auto-ID Center, [http://auto-id.mit.edu].

[15] Ptolemy Simulation Tool. [http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm].

[16] simanim package paper, "simjava: A Guide to Writing simanim Animations," [http://www.dcs.ed.ac.uk/home/hase/simjava/simjava-1.2/doc/simanim_guide/index.html].

[17] simjava Simulation Tool, [http://www.dcs.ed.ac.uk/home/hase/simjava/simjava-1.2/].

[18] simjava Paper, [www.dcs.ed.ac.uk/home/fwh/emin/docs/websim], "simjava: a discrete event simulation package for Java."

[19] simjava Guide, [http://www.dcs.ed.ac.uk/home/hase/simjava/simjava-1.2/doc/simjava_guide].

[20] Somani A., "An XML Server for Networked Physical Objects," M. Eng. Thesis, MIT, 2001.