The Dynamic Connection Framework:

Intelligently Creating and Maintaining Connections

in a Volatile Network Environment

by

Jaime A. Meritt

B. S., Computer Science

Massachusetts Institute of Technology, 1998

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Computer Science

at the Massachusetts Institute of Technology

May 1999

Author____

Department of Electrical Engineering and Computer Science

May 21, 1999

Certified by____

Hari Balakrishnan

Thesis Supervisor

Accepted by____

Arthur C. Smith

Chairman, Department Committee on Graduate Theses

The Dynamic Connection Framework:

Intelligently Creating and Maintaining Connections

in a Volatile Network Environment

by

Jaime A. Meritt

Submitted to the Department of Electrical Engineering and Computer Science

On May 21, 1999 in Partial Fulfillment of the Requirements

for the Degree of Master of Engineering in

Computer Science

Abstract

The Dynamic Connection Framework (DCF) is an enhancement to the existing enterprise architecture at American International Underwriters. The DCF's purpose is to seamlessly integrate into the existing CORBA environment and compensate for the weaknesses inherent in the current architecture. The DCF utilizes agent technology to intelligently create, monitor, and maintain connections to components located on servers located on the network. Benefits of the DCF include transparent failure recovery, load balancing, and centralized server information coupled with a light footprint and easy integration into legacy code.

Thesis Supervisor: Hari Balakrishnan

Title: Professor of Computer Science

Acknowledgements

The work presented in this thesis could never have been completed without the input and assistance of many individuals. First and foremost I would like to thank the staff of the Component Development Services group at American International Underwriters. The guidance of Mike Bracuti, William Pan, and Scott Chalmers is especially appreciated. The DCF was built within the infrastructure that they established at AIU.

Secondly I would like to thank my friends and family for the constant support and advice they gave me throughout the project. My parents have always been the example that I compare myself to. Hopefully I have lived up to their expectations. Risa Matsumoto motivated me to complete my work more than any other influence in my life. Her constant attempts to organize my schedule and life by taking much of the burden on herself will forever be appreciated. I would like to recognize the assistance I received from my fellow MIT alumnus and graduate student, Kirk Seward, who helped me in the organizational aspects of thesis preparation. Lastly, I would like to thank the staff at X-Collaboration Software for understanding the burdens of my school schedule and providing me with substantial time to complete my graduate studies.

Table of Contents

Chapter 1: Introduction and Background

1.1: Motivation

In recent years, American International Underwriters (AIU) has shifted their system development strategy toward the goal of creating reusable component-based applications. These applications were the first attempt by AIU to delve into the domain of enterprise computing. Now that the paradigm has been successfully proven, the second generation of enterprise applications are being designed and implemented. These second-generation applications, being more robust than anything previously developed, are beginning to point out weaknesses in the current infrastructure. The goal of the Dynamic Connection Framework is to compensate for these deficiencies while adding new functionality to the current application architecture.

CORBA, the middleware standard at AIU, is extremely reliable and scalable making it a valid choice for a robust architecture. However, it lacks certain features that would make it a better solution. CORBA has no inherent failure recovery and load balancing mechanisms. Currently, a CORBA component that fails for some reason can cause an entire application to grind to a halt. Exception handling allows the developer to prepare for many of these failures, but truly fatal errors are not handled. The load-balancing feature is desirable in enterprise applications because currently, components that receive too many requests may fail causing loss of data and application unavailability.

The creation of client programs that interact with components can be an arduous task. A client developer must use CORBA-specific methods to create a connection with a component. Most developers have little distributed object experience and waste time and energy trying to accomplish even the simplest of tasks in this environment. Figure 1.1 is a code excerpt that illustrates this point. In this example, only one line of code calls an operation on the desired component. Connection creation and exception handling incantations form the bulk of the body of code. Clearly a better solution is possible.

```
      ORB.init();  //initialize the ORB
DAL dal=null; //variable declaration
String ior="IOR:020938473829103203eb000007478392192103088439 98...";
try
{
          //connection incantation
    dal=DALHelper.narrow(ORB.string_to_object(ior));
      }
catch(SystemException se)
{
    //can't connect
          System.out.println("Cannot connect to DAL component");
          System.exit(1); //exit with error status
      }
dal.createConnectionToDatabase("livtransl"); //the DAL can now be used
```

Figure 1.1: Connection creation code

Connections between a client and a component are implemented as streams that IIOP requests are passed through. Simplicity has many benefits, but an enterprise application must often add complexity to achieve a robust solution. A connection that could provide statistics concerning the service that the client is connected would allow programmers to have more direct control over the client/component interaction.

A final motivational force driving the creation of the DCF is experimentation with new technology. Intelligent Agents (IA) have been proven as an effective solution to common problems in enterprise computing. Use of agents in a mission critical application is the first step towards acceptance of a new technology within the corporation.

1.2: Enterprise Applications

1.2.1: Overview

Technological superiority is a competitive advantage that companies in every industry are battling to acquire. Falling behind in information technology can lead to a rapid loss of market share and can prevent a company from expanding to new markets. Due to these conditions, information technology has become the driving force behind the future endeavors of corporate America. This new reliance on IT has created an entirely new field of computing known as Enterprise Computing.

Enterprise computing is a broad field that encompasses a new breed of applications designed for use in the corporation. These applications span all industries and perform a wide variety of business tasks efficiently, where previous systems failed entirely. Although enterprise systems are created to perform a multitude of functions, they all have certain features in common that define them as enterprise applications. The main features of an enterprise application are robustness, expandability, compatibility, stability, and security.

Robustness is a necessity because of the complex nature of business problems in today's multinational corporations. Business rules governing a process are dependent on company policy, legal concerns, fiscal issues, etc. The application must be suitably complex to model the intricacies of the task that it is to perform. An enterprise application that is not robust enough, cannot hope to capture the essence of the process it is modeling.

Expandability is the feature of enterprise applications that ensures applicability in future endeavors. Since no one can predict the future needs of the users of the application, enterprise applications need to have an easy extension mechanism to incorporate whatever the business demands dictate. Expandable applications have an architecture that invites change. New features are easily incorporated into the existing infrastructure making enterprise applications maintainable in the ever changing corporate environment.

Compatibility refers to the ability of an enterprise application to access data and use the services of existing legacy applications. An application without access to the years of stored information on the company network is severely limited in functionality. Without legacy compatibility, corporations would have to build entirely new applications from the ground up to reap the benefits of enterprise computing. Instead, companies can incrementally add new functionality to existing systems without scrapping the software they invested thousands of man-hours in creating.

Stability is one of the hardest features to implement in an enterprise application. A stable system is one that is highly available, reliable, and is prepared for unexpected failures. Enterprise applications are designed with scalability in mind so if an unexpected boom causes the application to be stressed beyond the normal load, the system does not fail.

Finally, enterprise applications must be secure. All actions taken by users within the application should be monitored. Users should only have access to services that they are authorized to use. These common features have caused architectural standards to evolve that are independent of the intended purpose of the enterprise application.

1.2.2: Enterprise Architecture

Enterprise applications are created to solve the full spectrum of business problems. This diversity in intent makes superficial differences apparent while masking the underlying similarities. In an enterprise system, all applications are created from the same fundamental building block, the component. Components are an evolution of the object-oriented technology of yesteryear. Previous manifestations of OO technology limited the programmer to a particular language, platform, and/or process space. "Unlike traditional objects, components can interoperate across languages, tools, operating systems, and networks. But components are also object like in the sense that they support inheritance, polymorphism and encapsulation."[5] The platform independent nature of components coupled with the inherent benefits of the OO paradigm make component technology the logical choice for the infrastructure supporting the technology of tomorrow.

Components are implemented as objects or as a network of interacting objects that are packaged as a single functional unit. This unit is not an application in itself, rather it is a black box entity that provides some behavior. Assembling components into an appropriate architecture and writing the code to "glue" the components together creates applications. The behavior of a component is defined in a published interface that other components use when making requests of the unit. The implementation of the component need only provide the behavior specified in the interface. Other implementation issues such as programming language, operating system, etc. are made at the discretion of the programmer. This separation of interface and implementation is one of the great strengths of a component architecture. It allows the implementation of the component to be changed without altering its behavior. This benefit is made possible by the interoperable nature of components. Components can communicate across any boundary. A component on a computer across the network is accessed in the same way that one would access a local object. Strict adherence to component standards is the key to maintaining this interoperability.

Middleware is the "glue" that binds components together and allows them to interoperate across network and language boundaries. Without middleware, components are merely disparate objects located somewhere on the network. In essence, middleware is a communications standard that all components must adhere to and an object bus that routes requests between components. Currently, two standards are competing for domination of the middleware market. The Distributed Component Object Model (DCOM) is a proprietary standard developed by Microsoft to provide a middleware solution for Windows environments. The Common Object Request Broker Architecture (CORBA) is the other standard competing for market share. CORBA is a joint effort by over 500 companies to create an industry standard component communications framework. CORBA is discussed in greater detail in Section 1.3. Components targeting a specific middleware standard can be implemented in any language that has a mapping to the middleware architecture.

The architecture of an enterprise application is a departure from the 2-tier client/server model used in many present day systems. "Tiers describe the logical partitioning of an application across clients and servers...In 2 tier client/server systems, the application logic is either buried inside the user interface on the client or within the database of the server."[2] Data access requires explicit knowledge of the structure of the server side data on the client side. Any structural change in the data on the server required changing all of the clients accessing that data. In a large system, a change in data structure could mean altering the code on thousands of clients. This limitation of 2 tier systems was a key driving force behind the creation of an n-tier architecture, the infrastructure underlying all successful enterprise systems. An enterprise system is separated into 3 major tiers, the client, the application layer, and the data layer. The data layer is unchanged in the n-tier approach. It is simply a relational or object database that contains application specific data. The client tier is a "thin" user interface that contains no business logic. The application layer, or server, contains all of the business logic. The server is implemented as any number of interacting components. "Instead of interacting with the database directly, the client calls business logic on the server. The business logic then accesses the database on the client's behalf."[2] The clients access the application layer through the middleware using the published interfaces of the components. In a 2-tier system a change in data structure required a large upgrade in all of the clients. A benefit of the n-tier approach is that all data access is encapsulated within components. Therefore, to accommodate a change in data structure, only the code in the component that encapsulates the data needs to be changed. Thin clients are only dependent on the interface that the component publishes and require no changes.

The Enterprise Architecture is a great leap forward in the design and implementation of systems for corporate use. Components as the fundamental building blocks provide an object-oriented infrastructure with the added benefit of location transparency through the middleware. The n-tier approach solves problems associated with maintainability and management. With these benefits as a foundation, it is extremely likely that the enterprise application will be the de facto methodology in corporate IT projects for the foreseeable future.

1.3: CORBA

1.3.1: Overview

In 1989, a consortium of leading object vendors formed the Object Management Group (OMG), a standards body with the intent to develop an industry standard architecture to provide component interoperability. After 2 years of work the OMG developed the Common Object Request Broker Architecture (CORBA). Since its initial release, CORBA and the OMG have undergone dramatic changes culminating in the 1994 release of CORBA 2.0, a development effort involving over 500 companies. CORBA 3.0 is scheduled for release in the near future and hopefully it will be embraced by industry as much as the previous two releases.

1.3.2: The Object Request Broker

The Object Request Broker (ORB) is the fundamental middleware entity that establishes the client/server relationship between components. The ORB allows clients to make requests of server components whether they are located on the network or on the local machine. "The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results." [5] ORBs use a communications protocol known as the Internet Inter-ORB Protocol (IIOP) to pass requests over the network between components. IIOP is fast becoming the industry standard intercomponent communications protocol, gaining widespread acceptance and use in many enterprise applications. Both Sun and Netscape, industry leaders in their respective fields, incorporate ORBs into their products saving their developers the time and effort that would go into a proprietary solution that would not be compatible with any other products. The use of the ORB as the object bus in an enterprise application guarantees compatibility with all components developed using CORBA, irrespective of the vendor of the ORB.

1.3.3: Component Development

The development of a CORBA-enabled component, or service as they are sometimes known, is a simple process. The first step requires the definition of the service interface in a language neutral format. The CORBA specification defines the Interface Definition Language (IDL) for this task. IDL is very similar to C++ header files in syntax and purpose in that it defines only interfaces, leaving the implementation to associated classes. After the IDL has been defined, it is passed as input to an IDL compiler. IDL compilers are packaged with all commercial ORBs and are used to translate the language neutral IDL interfaces into language specific stub, skeleton, and helper classes. These files provide methods to connect to an object of the IDL defined type through an ORB. This translation is accomplished using the OMG defined mapping of IDL to a particular language. The OMG, drawing on the resources of its large member community, has successfully defined mappings of IDL to all of the mainstream object oriented languages including C++, Smalltalk , and Java. Currently the OMG, mimicking industry, is focusing its attention on Java. The much anticipated CORBA 3.0 specification further incorporates Java into the OMG defined infrastructure allowing the Java language to be used to define interfaces.

The stub and skeleton classes generated by the IDL compiler are the fundamental tools used by the programmer to CORBA enable an application. These classes provide facilities that can bind to and make requests of other CORBA objects by interfacing with the ORB. CORBA supports a dynamic binding and request generation architecture as well, but its use is beyond the scope of this project. When a client of a particular service makes a request, the stub is responsible for marshaling that data. Marshaling entails changing the format of the data from a language specific structure to an IIOP defined format. The stub then sends the request over the wire in that language neutral format. The skeleton reverses the action of the stub on the server side. Requests are received and the skeleton class proceeds to translate the IIOP data into language specific structures and classes. These actions are carried out in a transparent fashion. Neither the server nor the client code interacts directly with the stub and skeleton. The ORB handles interaction with these classes behind the scenes.

The generation of the stub and skeleton classes lays the foundation for the implementation of a CORBA compatible component. After the code is run through the IDL compiler, the developer must implement the interface that the component publishes. The programmer can implement the service in any number of ways, as long as the interface is completely supported. With the interface implementation in place, the final phase is the creation of server programs. These servers are, in essence, object factories. They are responsible for instantiating object implementations and registering these objects with the ORB. Registration mechanisms vary from vendor to vendor, but they usually manifest themselves as single method calls that give the object an address that allows it to be accessed by CORBA clients through the ORB. These addresses, known as Interoperable Object References (IOR), provide a unique name that identifies an object on the network. At this point, the creation of a network accessible, language independent component is complete.

1.3.4: Accessing Components

A client that wishes to access the service must know the IOR associated with the component. Using the `string_to_object` method of the ORB, the client gets a reference to a provider of the service and can make method calls on the component as if it existed in the local process space. The client need only have a local copy of the stub class to interact with the service. Implementation classes are housed solely on the server. The location transparent and language neutral features of CORBA make it an extremely useful tool in the implementation of enterprise applications.

1.4: Intelligent Agents

1.4.1: Overview

Intelligent Agent (IA) technology is a new architectural paradigm based on a model of human interaction. The client/agent relationship is a common form of interaction in today's society. Agents, such as real estate agents, are experts in a particular field. A client seeks the aid of the agent to carry out some duty that he or she has little knowledge of, such as renting an apartment. The client trusts the agent to perform his or her function in a manner that is either equivalent to, or better than, how the client would carry out the same task. IAs mimic the functionality provided by human agents in the software domain. "An intelligent agent is considered to be a computer surrogate for a person or process that fulfills a stated need or activity. This surrogate can be given enough of the persona of a user or the gist of a process to perform a clearly defined or delimited task." [3]

Intelligent Agents have been a "hot" research topic for some time now. Therefore various conflicting definitions have arisen concerning exactly what an IA is. Most researchers merely alter the IA definition of a previous project to make it conform to the problem at hand. However, over the course of time, a basic definition of an Intelligent Agent's capabilities has emerged from the melange of proposals that exist. These capabilities are as follows:

- Autonomy: Agents operate without the direct intervention of humans or others and have some kind of control over their actions and internal state.
- Social Ability: Agents interact with other agents (and possibly humans) via some kind of agent communication language.
- Reactivity: Agents perceive their environment, and respond in a timely fashion to changes that occur in it.
- Proactivity: Agents do not simply act in response to their environment, they are able to exhibit goal directed behavior by taking the initiative. [4]

1.4.2: Agency

The true power of IAs stems from the social characteristics of the agents. A single agent is basically a goal driven process. Societies of agents, however, are much more complex. Societies, or agencies, are created by connecting agents together into a network of autonomous entities that communicate with each other. This communication network allows agents to share information with others, allowing a single agent to draw on the wealth of information already collected by IAs in the agency. This sharing of information makes IAs much more efficient by reducing the effort that goes into collecting the data. Collection is the job of the community, not the individual. In addition to the sharing of information, agencies can provide other features. Some agent systems implement a bidding architecture where only the agent who can accomplish the goal in the most effective way is assigned the task. In such an architecture, agents are capable of fulfilling a multitude of tasks but are fine tuned for a specific subset of jobs. When a job becomes available, the agents are asked to bid, thereby rating their ability at the task at hand. The IAs that are fine-tuned to a particular job bid better than those who's skills lie elsewhere. In many cases, agencies exhibit unexpected behavior stemming from unseen facets of the problem that the society of agents focuses on. There are many other examples of agent interaction and architecture, but they are beyond the scope of this paper. Many software packages are beginning to incorporate IA technology into their product and industry is examining the benefits of using IAs to act as surrogates for humans in their enterprise applications.

1.5: The Universal Modeling Language

1.5.1: Overview

As the size of an application increases it becomes increasingly difficult for any single developer to conceive the system in its entirety. A thorough software model is the key to success for any large-scale object-oriented system. The Universal Modeling Language (UML) provides a standardized notation that allows developers to communicate the design of an object-oriented system in a visual and concise manner that is readily interpreted by anyone familiar with the UML representation of an application's components. Much of the UML notation is irrelevant to the project at hand. This section will explore the subset of the UML that was used in the development process that guided this project.

1.5.2: Use Case Diagrams

"A use case scenario is a description, typically written in structured English or point form, of a potential business situation that an application may or may not be able to handle." [1] The business analyst, touting the necessity of the proposed system, typically assembles these scenarios. Figure 1.1 is an outline of a use case scenario describing the act of placing an order in a telephone catalog application. The format of a scenario depends upon the complexity of the use case that is being described. The figure is merely an example, it is not uncommon for a real-world use case scenario to consume pages of documentation. The more precise a scenario is, the less likely a miscommunication will occur between the business analyst and the application developers. The most important part of the scenario is the sequence section which presents a high level view of the actions that must be taken to complete the scenario. The developer uses this section to model the interactions between the entities of the system. The scenarios are visualized in the Use Case diagram which models how real-world elements are expected to use the system and what functionality the system is expected to provide. The use case diagram effectively replaces the business requirement document by conveying the gap in business application software that the application will fill.

Telephone Catalog: Place Order Scenario

```
DESCRIPTION:
This use case scenario takes place when a customer,
with the aid of a salesperson, places an order to be
placed in the catalog.

PROTOCOLS:
In:
Customer Information (i.e. phone number, etc.)
Out:
Confirmation or rejection

SEQUENCE:
The customer gives his information to a sales associate.
After the validity of the customer's information is
ascertained, and the method of payment is approved, the
customer is added to the catalog.

EXCEPTIONS:
CustomerAlreadyCataloguedException is thrown when the customer
has already purchased an entry in the catalog
```

Figure 1.1: A use case scenario describing the "place order" scenario

The use case diagram visually presents the interaction between non-system entities and the system itself by associating particular use cases with individual actors (the non-system entity). Figure 1.2 depicts a typical use case diagram representing a telephone catalog system.

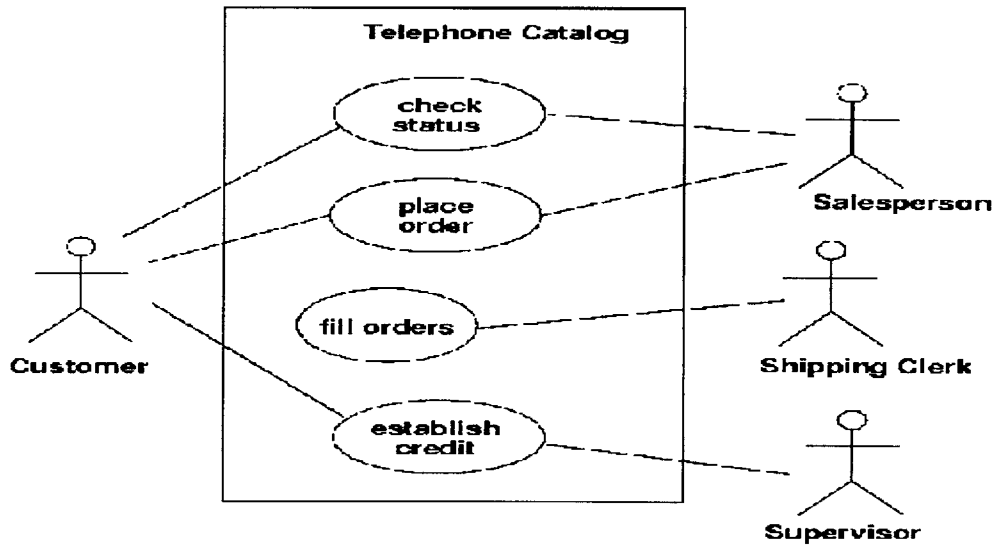

Figure 1.2: An example use case diagram depicting a telephone catalog system

The actors external to the system are the salesperson, shipping clerk, supervisor and customer. The ellipses represent the use cases and the rectangle that the use cases reside in is the entire system. A line between an actor and a use case represents an association. The type of association depends upon the context of the particular use case and the business domain. In the example diagram, the "place order" use case is associated with both the customer and the salesperson actors. One possible interpretation of these associations, based on the business problem at hand, is that the customer initiates and places an order with the assistance of the salesperson. The use case diagram in Figure 1.2 adequately models the business responsibilities of the proposed system. The remaining diagrams that are explored in this section concretely model the application's structure and behavior.

1.5.3: Class Diagrams

The class diagram models the static structure of the application. This diagram isolates the system entities and how they are related to one another. The class diagram, as it evolves, becomes the object model for the application. The classes are composed of two elements. Class attributes are similar to instance variables in the OO realm. These attributes represent the

internal data that every instance of a particular class has in common. Operations describe the particular actions that a class can perform. In object-oriented terms, operations are realized as methods within an object that implements the class.

The relationships between the classes of the model, for the purposes of this project, are members of three basic sets. Relationships of the aggregation type symbolize that one class "contains" instances of another class. Aggregations are a type of "has-a" relationship. Object inheritance relationships are depicted in the model using a generalization. Generalizations can be defined as an "is-a-kind-of" relationship. Relationships that are neither generalizations nor aggregations fall into the category of associations. Classes that share an association are weakly linked. For example, if a class operation returned an instance of another class, the two classes would have an association relationship. The association is best explained as a "uses" relationship.

The image depicted in Figure 1.3 is a class diagram corresponding to the aforementioned Telephone Catalog application. Although the example model contains only five classes, real world enterprise applications can be composed of hundreds of classes with complex relationships. Classes are represented as compartmentalized rectangles in the diagram. The compartment below the name contains the attributes of a class. The bottom compartment

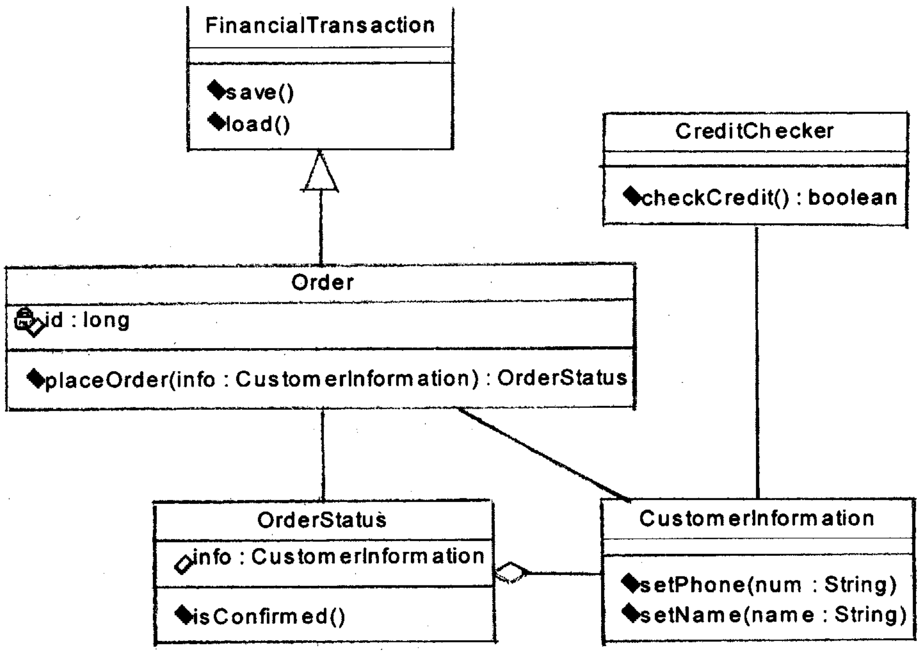

Figure 1.3: An example of a class diagram

houses the class' operations. According to the diagram, the Order class has one attribute and one operation. Attributes are specified as <visibility><name>: <type>. The visibility term corresponds to the OO notion of visibility (i.e. public, private, etc.). Therefore all instances of the order class have a private attribute named id that is a `long`. The proper format for an operation signature is <visibility><name>(<arg1>, <arg2>,..., <argN>): <return-type>. Applying these guidelines, the placeOrder operation accepts one argument of the CutomerInformation type, returns an OrderStatus object, and has public visibility.

Relationships in a class diagram are represented as a line connecting two or more of the previously described class rectangles. An aggregation is visually displayed as a line with a diamond at its head. The diamond is closest to the class that is the client in the aggregation relationship. Figure 1.3 contains an aggregation relationship between OrderStatus and CustomerInformation. The client in this example is the OrderStatus class signifying that it contains one or more instances of the CustomerInformation class. Generalizations are represented in the model as lines with a triangle at the parent class. Therefore, according to the example model, the Order class is a child of FinancialTransaction. The remaining relationship that is presented in the diagram is the association. These are represented as lines with no adornment. Since the placeOrder operation of the order class uses both the OrderStatus and CustomerInformation objects, the three classes share associations.

1.5.4: Collaboration Diagrams

The use case and class diagrams are effective in modeling the business requirements and the architecture of a system. Collaboration diagrams visually present the interactions between the classes of the application thereby modeling the behavior of the system. Collaborations are typically associated with a particular use case and are used to explain how the classes of the system fulfill the requirements of said use case. If a collaboration diagram cannot be created that can complete the tasks dictated by a scenario, then the class structure is lacking functionality. This feature of collaboration diagrams makes them useful as an initial feasibility test when a developer is attempting to ascertain the validity of a proposed architecture.

A collaboration diagram is shown in Figure 1.4 that is associated with the "place order" use case discussed in the previous sections. Instances of a class are represented as rectangles. The arrows in the diagram symbolize usage of a particular class' operations by the object at the tail end of the arrow. The "Sales Program" object is a non-system entity that acts as a client of the telephone catalog system. The collaboration begins when the sales program calls setPhone and setName on the CustomerInformation object. This scenario may occur when a salesperson is collecting information from the customer over the phone. After the CustomerInformation object is contains the correct values, the salesperson initiates the order process by calling the placeOrder operation, with the CustomerInformation argument, on the Order object. The Order uses the CreditChecker object to validate the transaction. The final phase of the collaboration occurs when the sales program queries the OrderStatus object, returned from the placeOrder operation, to see if the order has been confirmed. The creation of a collaboration diagram from the class diagram in the previous section suggests that the proposed structure is an adequate architecture for the requirements dictated in the "place order" use case.

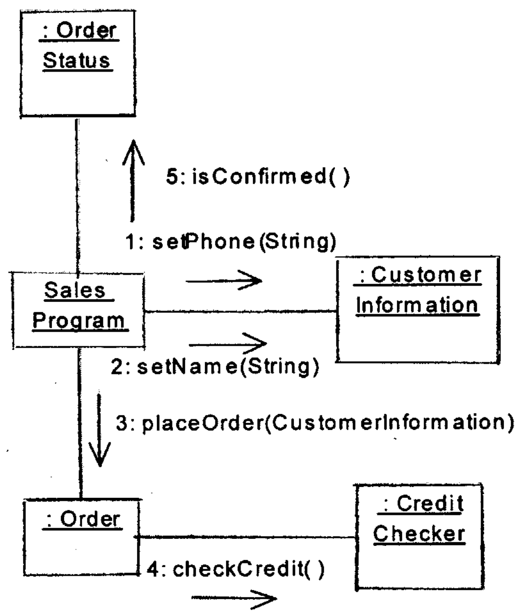

Figure 1.4: The "place order" collaboration

1.6: Component Development Services Infrastructure

American International Underwriters (AIU) is a division of American International Group primarily concerned with the sale of insurance products in foreign countries. The Component Development Services (CDS) group at AIU is responsible for the creation and maintenance of components for use in enterprise applications. CDS develops components for IT infrastructure as well as for business purposes. The infrastructure components are the focus of this project.

CDS maintains a library of reusable infrastructure components that are applicable for nearly any enterprise application. An example of such a component is the Data Access Layer (DAL). The DAL provides efficient and secure access to corporate databases by encapsulating vendor specific access mechanisms and business specific security concerns within a single component. Any application that requires access to a database needs to incorporate the DAL into its architecture. Since almost every application requires some form of data access, the DAL is one of the most important components that CDS provides.

The DAL, and other infrastructure components analogous to it, share certain characteristics. All CDS components use CORBA as the middleware infrastructure. Use of CORBA provides OS and language independence that is necessary in an international industry targeting the different platforms in use around the world. All infrastructure components are persistent services that are available to the network at all times. Server down-time is minimized because the unavailability of a component of this type can cause many applications to grind to a halt. Connections with infrastructure components tend to be of a long duration. Often, an application component maintains a reference to an infrastructure component from the beginning of its execution to the very end. Finally, the major CDS infrastructure components are stateless. The main benefit of stateless components is that in the event of a component crash, the service does not lose any information. Restarting the component server will fully restore the component without any loss of functionality. All instances of stateless components are equivalent as long as they implement the same interface. This project builds upon this infrastructure by taking advantage of the common characteristics of CDS components while adding new functionality.

Chapter 2: Design and Implementation

2.1: Methodology

2.1.1: Overview

The complex nature of object-oriented programs has led to many innovations in the methods employed in the design and implementation of such systems. High-level visual models, such as the UML diagrams presented in Section 1.5, represent the object relations and interactions in a more concise and descriptive manner than the low-level specification documents used as design templates for procedural language based programs in the past. In addition to these advances, much work has been expended on defining methodologies that guide the development of an object-oriented system from its initial conception to its eventual deployment. These modeling techniques and design methodologies will guide the development of the Dynamic Connection Framework.

2.1.2: Modeling Procedure

The initial step in the design process of any system is a thorough analysis of the business requirements that the application must fulfill. Section 1.1 explained the motivational forces that necessitate the creation of the DCF. These motivations are analyzed and use cases are created from them. The DCF is deemed fully functional when it can fulfill the requirements dictated by the use cases.

After the initial use cases have been defined, the implementor identifies the system entities that would need to exist to fulfill the requirements of the use case. The general method employed in this phase is to select the reoccurring nouns that appear in the use cases and create classes from them. For example, the "Place Order" use case discussed throughout the UML overview dealt with orders placed by a customer. It follows from this observation that a good candidate for a class in this scenario is an order. Class behavior is defined using a similar approach. The verbs that appear in use cases are conceptually linked to operations performed on, or with the help of, a particular system entity. In the case of the "Place Order" use case, a possible behavior of the order class would involve the act of "placing" or "processing" itself. Therefore, an operation was defined that is responsible for placing an order.

The final step in the creation of the models is to map out the behavior of the system as a whole. This requires the creation of collaboration diagrams that use the system entities defined in the previous phase. Each collaboration should implement a particular use case. This is the most complicated step of the design process. No methodology exists that can guide developers through this portion. If a collaboration successfully models the completion of a use case, the portion of the system is implemented.

2.2: Requirements Analysis

2.2.1: Overview

The DCF must compensate for the weaknesses in the CDS architecture by providing a functional layer that transparently complements the existing infrastructure by adding new functionality. As discussed in Section 1.1, the main areas of functionality that need to be implemented in the DCF are failure recovery mechanisms, load balancing of CORBA services, simplified connection creation facilities, the querying of connections, and the integration of agent technology. This section documents the translation of these requirements into use cases.

2.2.2: Use Cases

The main function of the DCF is to act as an intermediary between client programs and CORBA services. Therefore, client programs are the only non-system entities that interact with the DCF making them the only viable choice for the actor in the use case diagram. With the actor identified, the next phase of analysis examines the different ways the client programs interact with the system. Figure 2.1 presents the complete use case diagram for the DCF representing the business requirements imposed on the system. The remainder of this section explains how the use cases were generated.

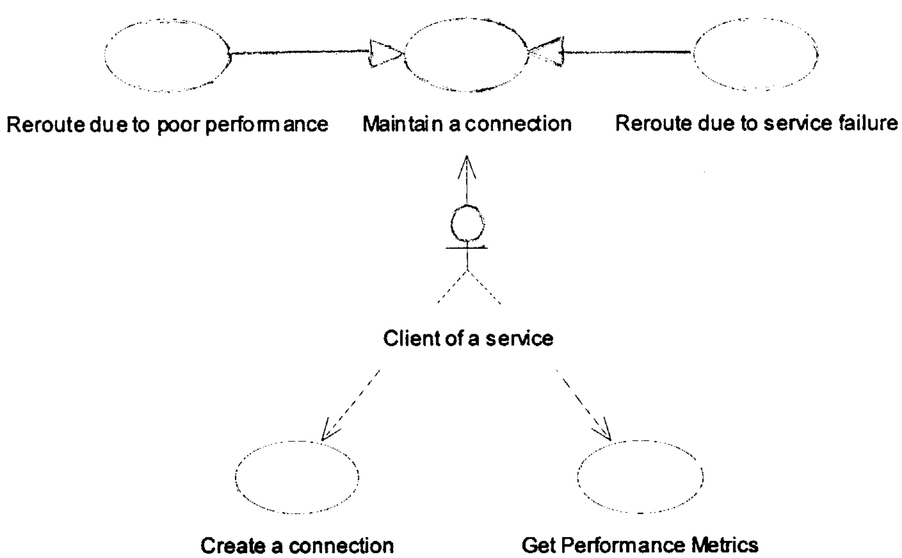

Figure 2.1: DCF use cases

Client programs use the DCF to create connections to the CORBA services that they need to interact with. Therefore the first use case of the DCF is "Create a Connection". The load balancing features mandated in the requirements act as a constraint on this particular use case. More specifically, this use case dictates that the DCF must be able to create connections to services and that the method employed in creating these connections must take into account the load of the service.

A second piece of functionality that a client uses the DCF to obtain is real-time connection maintenance. Maintaining connections entails providing the facilities to reroute to equivalent services in the event of service failure or extreme performance decrease. These requirements translate to the "Maintain a connection" use case. Employing the generalization relationship further specifies this use case. The children use cases present the actions that the system is responsible for carrying out in order to maintain a connection.

The third and final use case that the DCF will be required to provide for is the querying of performance metrics from a connection. The types of metrics that are available depend on the CORBA service that the client is interacting with. An example of such a metric is the number of database connections open by the DAL service. In the following sections, the implementation of the requirements set forth by these use cases will be presented.

2.3: System Architecture

2.3.1: Implementation Overview

The DCF is a system of interacting CORBA components implemented in the Java Programming Language. Java was chosen as the language for the DCF because of its platform independent nature. In a heterogeneous computing environment, such as the one that exists at AIU, applications must be able to run on multiple platforms. This fact makes the use of Java very appealing because it saves the developer from expending time and energy on porting the application to the numerous platforms deployed throughout the world.

The DCF is comprised of two major components that through mutual interaction satisfy the business requirements mandated by the use cases presented in the previous section. The Trader, the first component that will be explored in this section provides location and discovery services that enable client applications to determine the types of CORBA services that are available to the network. The Connection Agency, the other major subsystem implementing the DCF functionality, is a factory for agents that provide connection creation and maintenance services. In the proceeding sections, the components that implement the DCF will be explained and the methods employed to fulfill the use cases will be examined.

2.3.2: The Trader

The Trader is a central repository of server location information. It contains data concerning components that are currently available to the network and those that lie dormant. This information is used by entities in the DCF to locate services for clients. When a DCF aware component is started, it registers the interfaces it implements with the Trader. When it shuts down, the Trader is notified that the services are no longer available. When a request for a service is made, the Trader checks its database of running components. If no service of a requested type is currently available when a request is made, the Trader will start one of the dormant components to facilitate the needs of a client. In the DCF, the Trader is used to allow agents to "research" all available components that implement a particular interface.

TraderService

(from AIG.DCF.CORBA)

- getTargetsForInterface(interfaceName : String) : AIG.DCF.CORBA.DynamicConnectionTarget[]
- _deref() : Object
- addManager(manager : AIG.DCF.CORBA.TraderManager) : void
- registerIORForInterface(interfaceName : String, ior : String) : void
- removeManager(manager : AIG.DCF.CORBA.TraderManager) : void
- unregisterIORForInterface(interfaceName : String, ior : String) : void
- registerObjectInRepository(interfaceName : String, objInfo : AIG.DCF.CORBA.ObjectInfo) : void
- unregisterObjectInRepository(interfaceName : String, objInfo : AIG.DCF.CORBA.ObjectInfo) : void
- getRepositoryInfo() : AIG.DCF.CORBA.InterfaceInfo[]

Figure 2.2: TraderService Class

Figure 2.2 depicts the published interface of the TraderService. CORBA services use this interface to make the Trader aware of their availability. When a service is started it calls the `registerIORForInterface` operation supplying the interface that it implements and the service's IOR as arguments. This method call causes the Trader to add the service to its database of components that are currently available to the network. When the service is terminated, a call is made to `unregisterIORForInterface` that removes the service from the available component database. The `getTargetsForInterface` operation is used to query the Trader for a list of the components currently available to the network that provide implementations of a particular interface. Any one of these components provide the same functionality because of the stateless nature of the CDS infrastructure services. These components are returned from the operation as an array of DynamicConnectionTargets. The DynamicConnectionTarget is a wrapper class that contains all of the information necessary to create a connection to a particular CORBA service. An example of such a piece of information is the IOR of the service. If no component that implements the desired interface is in the available component database, the Trader activates a service that lies dormant and returns it to the client process. The remaining operations in the TraderService class manage this repository of dormant components.

2.3.3: The Dynamic Connection Agency

Whereas the Trader is a centralized repository of server knowledge, the Dynamic Connection Agency is the center of activity in the DCF. It is within this component that the desired functionality of the DCF is implemented. This component also acts as a testing ground for agent technology.

A client that desires a connection to a CORBA service interacts with the Agency through a proxy that is generated with the aid of a utility program, the DCF compiler. The proxy classes are created to connect to a particular service. For example, a client that desires a connection to the Data Access Layer component would employ a DALProxy to obtain such a connection. Figure 2.3 depicts the main classes that comprise the Dynamic Connection Agency. When a client instantiates a proxy, the proxy requests a connection to a service through the agency component using the `getConnection` operation. The arguments that are passed into this operation are the requested interface (i.e. the DAL) and a reference to the proxy that can be used as a callback.

In essence, the Dynamic Connection Agency is a factory for Connection Agent objects. These agents act as surrogates for a human programmer by finding a CORBA service that implements a particular interface and creating a DynamicConnection object. When a client proxy requests a connection from the Agency, the Agency instantiates an agent and calls the `createConnection()` operation on it. The agent acts to create the connection and return it to the Agency. This result is made accessible through the ORB to the proxy allowing the client to interact with the component. At this point the agent has no task delegated to it by any external party. In this state, the agent begins to monitor the connection that it created in an autonomous fashion. In this manner all of the connection creation and maintenace duties are delegated to the DynamicConnectionAgent by the Agency.

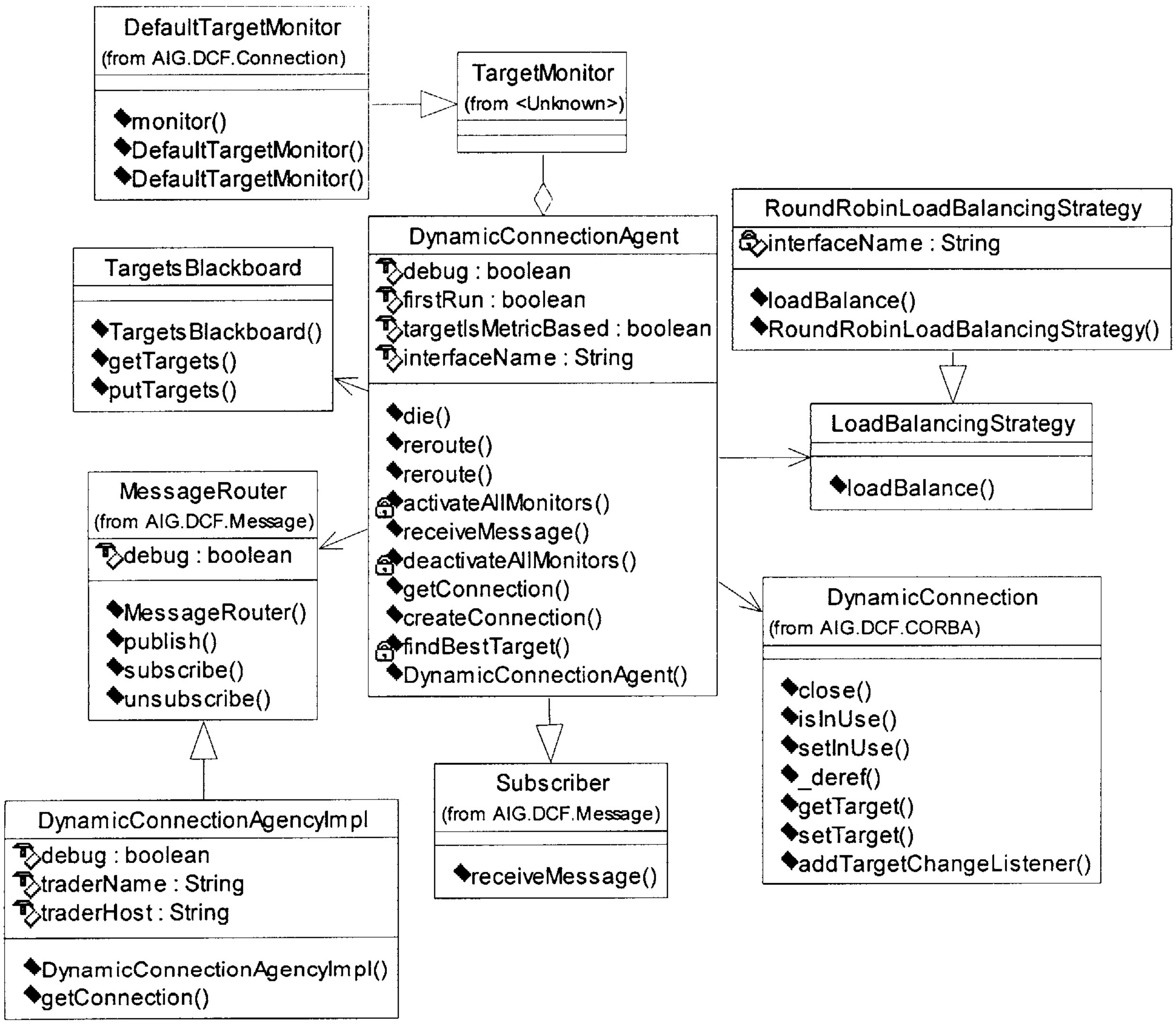

Figure 2.3: The major classes that make up the Dynamic Connection Agency

2.3.4: The Connection Agent

The Connection Agent is an IA that is responsible for creating connections and carrying out the requests of the monitors. Mimicking a human programmer, all an agent "cares" about is finding a component that implements a desired interface. Since all CDS components are stateless, any service that provides the desired interface is a suitable candidate. Which component it connects to is dependent on the current state of the system and the amount of knowledge it has about the components available to the network. The intelligence of the agent lies in its ability to create the best possible connection to a desired component at any time.

Figure 2.4 presents a collaboration associated with the "Create A Connection" use case. It provides insight into the internal workings of the connection creation mechanism. Recall from the previous section that a client proxy requesting a connection from the Agency begins the use case. When the agent is making a connection, its first goal is to find all of the appropriate components that implement the interface that is desired by the client. To find an appropriate target, the agent consults other Connection Agents first to ascertain if a similar problem has already been solved by the agency. This consultation occurs by querying the TargetBlackboard with the `getTargets` operation. The blackboard is a local repository of server information that mimics the functionality of the Trader. Its purpose is to prevent the agent from having to query the Trader for information about an interface that another agent has already obtained. If the agency has no knowledge of the desired interface, the agent consults the Trader. Using the `getTargetsForInterface` operation, the agent can request a list of components that implement the interface of the component in question. This information is then placed on the TargetBlackboard with the `putTargets` operation to help in future agent endeavors.

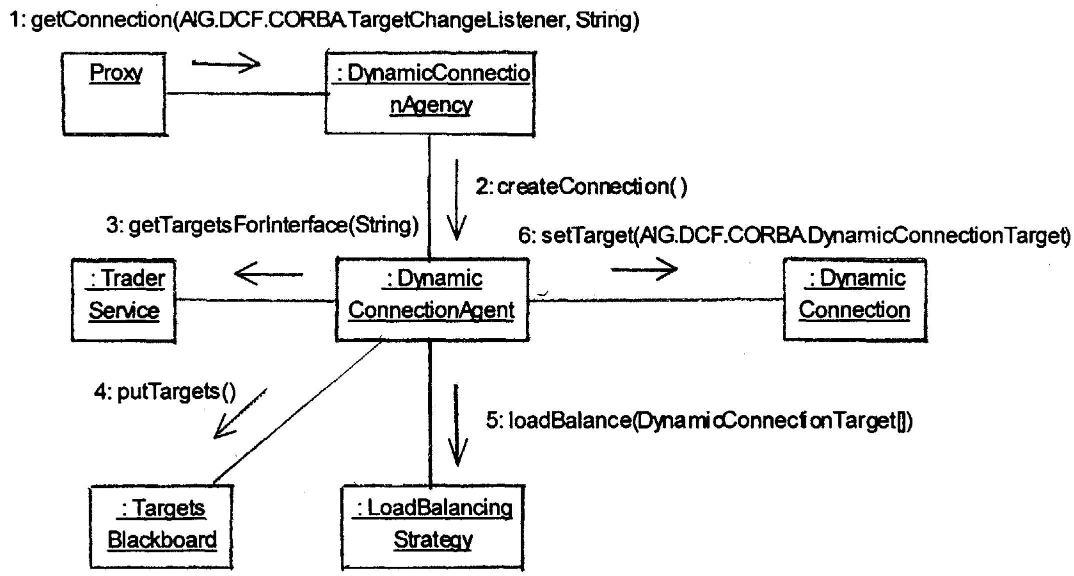

Figure 2.4: Create a Connection Collaboration

After consulting either the agency or the Trader, the agent has access to a list of suitable components to fulfill the client's need. If only one such component exists, a DynamicConnection is created with that sole component as the target. If more than one candidate exists, the agent employs a LoadBalancer to decide between the components. Load Balancers are interchangeable algorithms that take a set of candidate components and returns the best target. The DCF supports both metric and round robin based load balancing. Metric based load balancing is most effective because it takes into account interface specific information about the component when deciding between candidates.

The load-balancing algorithm returns the appropriate target to the agent for connection construction. The final stage of creation sets the connection's target and affixes the appropriate TargetMonitors. A Target Monitor is an object that uses the connection to ascertain the current state of the target. After the connection is instantiated, the monitors poll the target at regular intervals and interpret the results to decide upon a course of action. The connection is then made accessible to the client process where it is used to access the target. The monitors have references to the agent that are be used to call methods to remedy error conditions in the target. The most important of these methods is `reroute()`. In the event of an irrecoverable error, the monitor will call `reroute()` on the agent. This method causes the agent to reset the target object of the connection to a valid service. Figure 2.5 presents a collaboration associated with a "Reroute due to service failure" use case. The performance based rerouting occurs in an analogous fashion. The only difference is the event that triggers the scenario. These functions are carried out in a completely transparent fashion. The client is never aware that it is dealing with a different component. All the client is concerned with is the interface that the component implements.

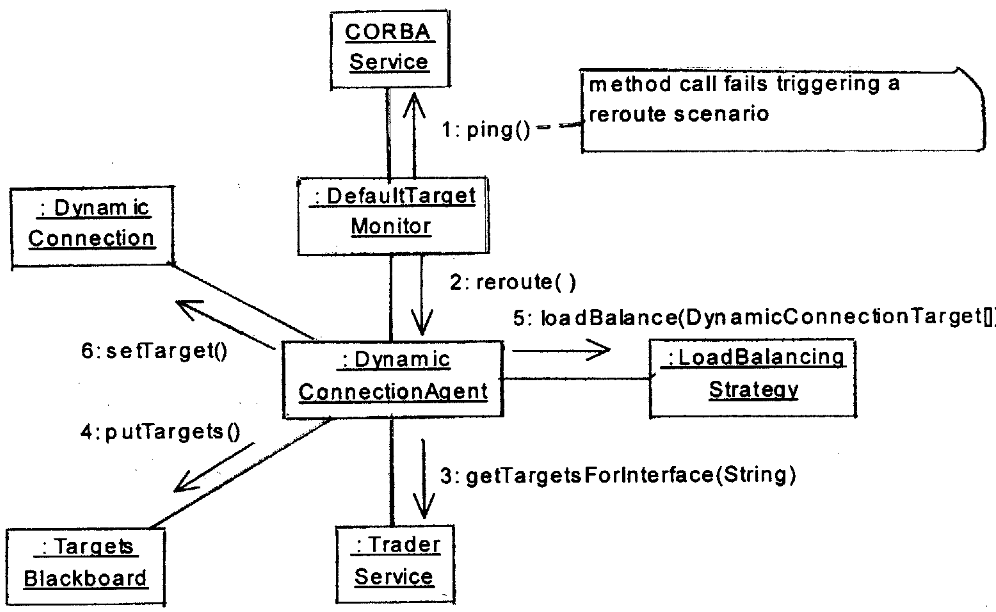

Figure 2.5: Rerouting Collaboration

2.3.5: The Dynamic Connection

The Dynamic Connection provides operations that allow a client component to access the desired CORBA service and to retrieve metrics from it. The connection contains a DynamicConectionTarget that is set by the agent using the `setTarget` operation called in both the connection creation and reroute scenarios. The component is accessed through this target object. More specifically, the target is provides an operation `getIOR` that contains the address of the component. Once the proxy obtains an IOR, a connection to the service can be created using the CORBA-specific methods presented in Section 1.1.

The DynamicConnectionTarget provides the `getMetricTable` operation that allows an agent or a client program to access a table that contains metrics concerning the current performance of a service. This information is used by the agent to perform metric-based load balancing and performance-based rerouting. The main drawback of the metric based services is that the algorithms that implement these pieces of functionality are applicable to particular services. For example, a DALLoadBalancingStrategy class would contain the necessary logic to perform metric-based load balancing on Data Access Layer components. Therefore, metric-based strategies should be used sparingly due to the additional complexity they introduce into the application logic.

2.3.6: Summation

With the addition of the metric facilities of the DynamicConnectionTarget the architecture of the DCF is fully described. The Agency and the Trader, acting in conjunction, fulfill all of the goals set forth by the use cases for the Dynamic Connection Framework. The Connection Agent creates connections in an intelligent fashion by using a load-balancing algorithm. Failure recovery is made possible by the interaction between the Target Monitors and the Connection Agent. Relevant statistics concerning service performance are acquired through the DynamicConnectionTarget object. Finally, the DCF tests the viability of Intelligent Agents within an enterprise component.

Chapter 3: Conclusions

3.1: Satisfaction of Requirements

3.1.1: CORBA Enhancements

The main motivational force driving the creation of the DCF was the need to enhance the existing CORBA middleware architecture used at AIU. Failure Recovery was implemented, as described in the previous chapter, by monitoring the targeted CORBA service at regular intervals. The moment a communication error or a performance decrease is detected, the agent proactively acts to remedy the situation by dynamically rerouting the connection to an equivalent service.

Load Balancing was the other major CORBA enhancement that the DCF implemented. Connections that are created through the DCF can be routed to several different services that provide the same functionality. This minimizes the load on each individual service by increasing the number of services available to the network at a given time. This functionality was made possible by the interaction of the Dynamic Connection Agent with the Trader Service.

3.1.2: Simplified Client Creation

Clients interact with the DCF through proxies that allow a developer to use a component without incorporating any CORBA-specific code. The proxy creates a reference to a correctly cast version of the connection's target. All method calls are made through this reference. When the target of the connection changes, the proxy is notified and the reference is refreshed. This process is transparent to the client that uses the proxy and allows the DCF to greatly reduce the effort associated with client construction. The DCF version of the DAL example in section 1.1 is presented in Figure 3.1. As can be seen in the figure, the CORBA specific method invocations are no longer present in the client's code. The proxy now contains all of the method invocations that are geared towards interacting with the middleware. The client simply needs to instantiate the proper class that is generated by the DCF compiler.

```
DAL dal=new DALProxy(); //instantiate the DAL Proxy class
dal.createConnectionToDatabase("livtrans1"); //the DAL can now be used
```

Figure 3.1: Client code to create a Connection to the DAL

The DCF compiler generates these proxy classes. The compiler is implemented using Java reflection. Reflection is a very powerful feature of Java that allows programs to discover information about a particular class at run-time. An example of the kind of information used by the DCF compiler is the return type and argument lists of all of the methods of a service. Using this information, the compiler generates the code for a service proxy.

3.1.3: Obtaining Service Statistics

The DynamicConnection class provides a means to obtain performance metrics pertaining to the service that the connection targets. By accessing the DynamicConnectionTarget object available from the connection class, the client is able to retrieve a table of service metrics. The entries in this table differ depending on the component that the DynamicConnectionTarget represents. For example, the number of database connections is a valid metric that is defined for all DAL components.

3.1.4: Intelligent Agents

The operation of the DCF is dependent on the effectiveness of the DynamicConnectionAgents. These agents satisfy the four conditions that define agency presented in Section 1.4. They are autonomous in that no other class controls theirs actions. The social ability feature is satisfied by the blackboard communications mechanism employed to share Trader information. The agents exhibit reactive behavior in that they detect and remedy failures in a service. An example of proactive behavior is the way in which monitors are affixed. Depending on the number of agents currently active, the agents will increase or decrease the level of monitoring on the connections that they are maintaining. If the system is very busy (i.e. many agents are operating) the agent will only attempt to identify failure conditions. When the system returns to a less stressed state, performance is monitored as well. This behavior is intended to have the agents act to alleviate the stress on the system.

3.2: Future Enhancements

3.2.1: Stateful Services

The restriction most hindering the widespread acceptance of the DCF is the inability to provide the same functionality for services with state. One of the main benefits of CORBA is the ability to create components that have internal state. Without this feature, the scope of the DCF is not general enough for corporate wide use.

One approach to addressing state concerns within the DCF would be to devise a strategy in which services wrote state information to a centralized repository at regular intervals. Using this information, an agent could create an equivalent service using the state information contained in said repository. The main disadvantage of this strategy is the performance decrease involved with centrally managing the state of a component. With hundreds of components available to the network, the increase in traffic due to the constant recording of state would be an unacceptable quantity.

3.2.2: Failure Recovery

A system designed to prevent failure should itself be protected from such occurences. A useful enhancement to the DCF would be internal failure recovery capabilities. Currently, the DCF is mirrored throughout the network providing redundant entry points to the system. Should one component fail, others exist that can take the place of the failed service. The main problem is that there is no process in place that provides failure recovery functionality if a DCF component encounters a fatal error.

A possible solution to this problem would involve removing the agents from the process space of the Dynamic Connection Agency and distributing them around the network. In this architecture, if a DCF component encounters an error, the agents that are still functioning can act to remedy the situation without being hindered by the error itself. The main drawback of this solution is the added complexity of the distributed agent architecture.

3.3: Reflections

The DCF was very well received by the CDS department at American International Underwriters. Upon completion, projects centering about its integration into the current architecture were underway. The Data Access Layer component is being redesigned with inherent DCF support. The DCF is being used for an unexpected purpose as well. It has been integrated into the OO training class as an example of the use case driven design process.

Bibliography

1. Ambler, Scott. "How the UML Models Fit Together." *Software Development Magazine,* March 1998.
2. Edwards, J. *3-Tier Client/Server At Work,* Wiley, New York, 1997.
3. Knapik, Michael and Johnson, Jay. *Developing Intelligent Agents for Distributed System,* McGraw-Hill, New York, 1998.
4. Majewski, S.D. Mail message to agents@sun.com mailing list in "Clarifications and Additions for AI: A Modern Approach," Web page at URL: http://www.cs.berkeley.edu/~7Erussell/clarify.html, 1996.
5. Orfali, Robert et al. *The Essential Distributed Object Survival Guide*, Wiley, New York, 1996.
6. Si Alhir, Sinan. *UML in a Nutshell: A Desktop Quick Reference,* O'Reilly & Associates, Cambridge, 1998.