INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa • University of Ottawa

Federated Name Resolution

by

Mircea Trifan

A thesis submitted to the

School of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

School of Information Technology and Engineering Faculty of Engineering

University of Ottawa

May, 1999

I hereby declare that I am the sole author of this thesis.

I authorize the University of Ottawa to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Mircea Trifan

I further authorize the University of Ottawa to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Mircea Trifan

ABSTRACT

The naming service is a central component for a distributed system. In this thesis it is investigated the creation of a highly scalable, federated, distributed and load balanced name resolution system supporting persistent names in a flat namespace. A new federation technique is proposed and applied for solving the name resolution problem in a flat namespace. The new federation algorithm proposed in this thesis is based on mapping a set of servers (and their names) into a set of nodes in a graph. Another mapping assigns names to a geometric net where each server manages a local set of nodes representing neighbor names. To retrieve a server reference or resolve a name, a hill climbing like technique is applied. The set of federated servers exposing the same interfaces and having identical implementation operate on the namespace distributed database. Mobile agents are used for the federation server's deployment, link construction and link update. In this way as opposed to a centralized server that uses replication to achieve scalability, there is no need to store the whole database of names at each server node. Also, any bottlenecks related to the root node server access that is present in any alternative hierarchical architecture, are avoided.

Two implementations of a subset of this federated name resolution system one using Orbix, a CORBA compliant product, and another in Voyager, a standard neutral ORB for mobile computing, are provided as a proof of concept. Immediate applications are in the DIATEMS network management project for the federation of the finder object used for bootstrapping on agent and event reporting sides. Other applications of this system that require further adaptations span from the creation of an alternative CORBA flat name service to the deployment of a uniform resource name (URN) service over the

Internet or the topology control for multihop packet radio networks.

ACKNOWLEDGMENT

First and foremost, I would like to express my sincere thanks to my advisor, Dr. Dan Ionescu, for his guidance, encouragement, friendship and support in quite a few key moments and decisions of my life. I was very fortunate to have the opportunity to work with him for more than two years. He ignited this dissertation by introducing me the federation concept long before it became mainstream with SUN's JINI initiative.

I wish to thank the members of my qualifying and dissertation committees: Dr. Dorina Petriu and Dr. Gregor v. Bochmann, for their helpful comments and discussions.

I wish also to acknowledge Dr. Voicu Groza for his friendship and for recommending me to the distributed computing interest group at the Machine Intelligence Group (MIG) laboratory. I was fortunate enough to be a member of this friendly, enjoying and stimulating research community. Thanks to Radu Ocica, who acted as a friend and research partner. My acknowledgements also go to the rest of the MIG group, old and new: Eugenia and Chris Lambiri, Phill Lavoie, Vlad Vieru, Monica and Virgil Cimpu.

I will always be grateful to my parents: Maria and Nicolae, for providing a supporting environment in which to grow up and for sending me far away to Ottawa. Finally, I would like to thank my fiance, Maria for the love, support and encouragement which made it possible for me to overcome the obstacles in the life of a graduate student.

Contents

List of Tables

List of Figures

Acronyms

Various abbreviations used frequently in this thesis are summarized below.

BNF	Backus Naur Form
CORBA	Common Object Request Broker
DBMS	Database Management System
DIATEMS	Distributed Interactive Application for Telecommunication Network Management and Supervision
DNS	Domain Name Server
DTP	Distributed Transaction Processing
FIPA	Foundation for Intelligent and Physical Agents
JIDM	Joint Interdomain Management
MASIF	Mobile Agent System Interoperability Facility
MIG	Machine Intelligence Group
OMG	Object Management Group
ORB	Object Request Broker
UML	Unified Modeling Language
URN	Uniform Resource Name

URL Uniform Resource Locator

VM JAVA Virtual Machine

Chapter 1

Introduction and Problem Definition

A naming service is one of the simplest and more useful services in a distributed object oriented environment. Its role is to allow a name to be associated with an object and to allow that object to be found subsequently by resolving that name within the naming service. The naming service provides operations to resolve a name, operations to create new bindings, delete existing bindings and to list the bound names.

Complexity appears when there is no hierarchy organising the namespace and a scalable solution is searched. This thesis is concerned with the definition of a very large, distributed name resolution system for a flat namespace.

1.1 Brief Literature Review

Previous name resolution systems have provided valuable insight into the creation of a global service, including necessary requirements, design features, and operational experience. Grapevine [53] and its successor, Clearinghouse [77], served primarily as messag-

ing systems. Email addresses were resolved into inbox locations, allowing users to move and have multiple inboxes. Scaling of this systems were limited by the fact that every server contained the structure of the entire name database. A larger system would have needed more space on each server and would have led to a higher frequency of structural changes. Clearinghouse had better scaling characteristics at the expense of adding a level to the naming hierarchy.

Lampson [34] highlighted the differences in requirements between a name service and a more general database in the description of his global name service. This name service used hierarchy as its chief method of dealing with growth. The deeper the hierarchy in a naming scheme the less likely names will remain constant as the people will cross partition boundaries more often. This system may be good for routing mail or authenticating users but failed to prove its ability to meet the combined goals of arbitrarily large size and long-lived names, both important requirements for a global name resolution system.

The Domain Name System [38] serves as an example of a successful, large, distributed service with extensive operational experience. However, like Lampson's global name service, DNS names are strictly partitioned into a hierarchy.

A class of names with some characteristics similar to the domain names is the Uniform Resource Name (URN). The Internet community has adopted this term for a name that identifies a resource or unit of information independent of its location. URNs are globally unique, persistent, and accessible over the network. A description of the functional requirements for Uniform Resource Names is given in [60] and an implementation of such a system developed in the Advanced Network Architecture group at MIT is presented in [56]. In order to achieve scalability independent of its contents, the distributed database that stores the names uses a B-Tree structure, a generalized version of

a balanced binary search tree that can have an arbitrary branching factor. The B-Tree has some shortcomings when applied to a distributed system. A root splitting algorithm achieves the tree growth. The root needs to remain constant because a central authority maintains it. All the tree nodes have the same maximum amount of information, but servers can vary in size. The branching factor limits the number of names per server and cannot be changed without rebuilding the entire tree.

1.2 Motivation and Research Objective

A novel federation technique is proposed and applied for the solving of the name resolution problem in a flat namespace. The federation concept is a modern trend in databases, distributed systems and network management. The concept of federation in computer systems was introduced by Heimbigner and McLeod and was based on the federation of information bases [29]. Their approach was directed to achieve coordinated sharing and interchange of computerized information, rather than relying on centralized control by a centralized (logical) organization.

In this technique a set of servers are mapped to a set of nodes in a graph. In the case of the name resolution system, each server manages a local set of names and it is linked to a number of similar neighbour servers managing their own local set of names. The set of servers exposing the same interfaces and having identical implementations operate on the namespace distributed database.

The main operations of the federated servers are: *bind* that associates names to values: object references, URLs, etc: *resolve* that returns the associated value to a name, and finally, *unbind* that breaks the association between the name and the corresponding value. Also the federation is able to insert or delete new resolver servers and update its

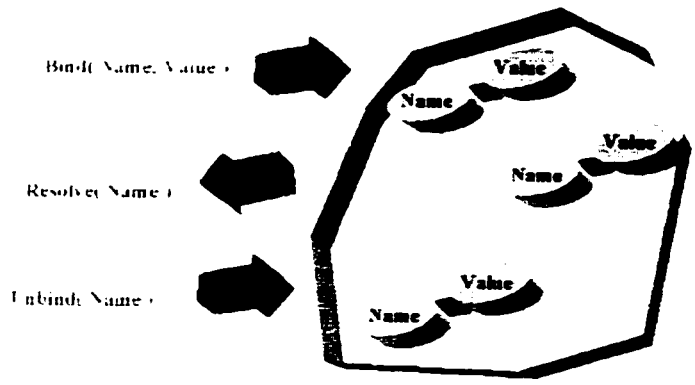

Figure 1.1: Name Resolution Main Functions

links.

Whenever a client issues a name resolution request addressed to a server in its proximity, the federation graph is traversed from node to node until a server object which can satisfy the request is reached and the associated information from the name binding is returned to the client.

At each intermediary node a decision has to be made to determine the next node in the resolution path. A similar graph-searching problem is the point location in the Delaunay triangulation from the field of computational geometry [49] where the selection among the node neighbours is done by computing the distances from them to the searched node and by choosing the one that is closer to the target.

A Delaunay network has several properties that make it interesting for the name resolution system. Scalability in terms of number of nodes is easily achieved. It is not unusual for a Delaunay mesh to contain millions of nodes like in the finite element

method, terrain modeling, biology and nuclear physics. The graph edges are load balanced in the sense that the minimum angles of all triangles in the planar triangulation are maximised by means of edge switching to improve connectivity. Thus, the nodal degree parameter of each node is negotiated among neighbours and the links are evenly balanced on the whole structure. The final topology is degree-bounded, it has a rather regular and uniform structure, and its throughput and reliability are greater than that of a number of alternative topologies. Inserting new nodes can easily refine the structure that is affected only locally. There are well-established algorithms for point insertion, deletion and searching that can be adapted for a distributed execution. Finally, there are no bottlenecks related to the root node access associated to a tree structure.

In addition, its dual structure, the Voronoi diagram [73], provides a way of partitioning the plane into a set of domains each of them being identifiable with the local partition of the namespace that is stored per federation server.

By definition, a Voronoi diagram of a set of points partitions the plane into a set of polygons representing the geometrical locus of points closest to each point from the initial set. If the points that have neighbouring Voronoi polygons are linked, its dual structure, the Delaunay mesh, is obtained.

The Delaunay network algorithms are adapted in order to perform the functions of a federated name resolution system. The set of federation servers and their links are mapped to the nodes and edges in the Delaunay mesh. Another mapping assigns names to geometric points where each server manages the local set of names that are included in its associated Voronoi partition. To retrieve a server reference or resolve a name a modified version of the point location algorithm is used.

Any server in a distributed application that holds an object reference can register it with the Federated Name Resolution Service, giving it a name that can be used by

other components of the system to subsequently find the object. One of the advantages of the naming service is that the names it associates with objects are independent of any properties of the objects they refer to: in particular, a name is independent of an object interface, server or host name. There are two ways in which an application can use the federated name service. It can be used to name a significant number of objects in the system in much the same way that all files in a system are named by a filing system. Alternatively, some important objects in the application can be named, and these objects can act as entry points for the other objects. The federated name service does not dictate which of these two models is used.

Two implementations of a subset of this federated name resolution system, one using Orbix, an implementation of the CORBA standard and another in Voyager, standard neutral ORB for mobile computing, are provided as a proof of concept. The CORBA implementation allows the components of any distributed application to use the new naming services, its interfaces being defined in the IDL language. Mobile code, as a new programming paradigm that enhances the traditional client-server model is used for: federation servers deployment, link construction and update, name resolution and bulk insertions. In the classical client/server paradigm, the application communicates requests and waits for the replies across the network. In mobile environments, the client migrates to the server to directly invoke requests.

The advantages motivating mobile agent-based computation are many-fold. First, mobile agents provide an efficient, asynchronous searching method for information: mobile clients could be launched into the federation nodes and roam around according to the resolution protocol until the targeted name is found. Second, mobile agents solve the client/server network bandwidth problem. By moving the federation link update transactions associated to any node insertions or deletions from the client to the server,

the repetitive request/response handshake is eliminated. Third, mobile agents are robust and fault-tolerant. If a federation server is being shut down, all agents executing there will be warned and given time to dispatch and continue their operation on another node in the federation. Fourth, agents reduce design risk. They allow decisions about the code location (client vs. server) to be pushed toward the end of the development cycle when more is known about how the application will perform. In fact, the architecture even allows for changes after the system is built and in operation. Finally, in the context of server deployment, it avoids the use of the factory pattern in a distributed environment. Mobile agents parked at the federation nodes are used as name resolution servers. For the implementation of this name service, the mobile agents are not required. Mobile agents seen as an extension of the stored procedure from the client server structured databases have the same advantages and disadvantages as the stored procedure concept.

1.3 Applications

Immediate applications of the flat naming service are in the DIATEMS network management project for the federation of the ProxyAgentFinders and EventPortFinders. This implements a CORBA network management system along the specifications of the JIDM group from the X/Open organization. The DIATEMS project is an JIDM standard extension and implementation developed in the Machine Intelligence Group (MIG)laboratory at University of Ottawa under the direct supervision of professor Dan Ionescu. The federated name resolution system could be an alternative CORBA name service. A scaled up version could form a solution for the uniform resolution name (URN) problem, where resources can move freely over Internet and are not linked to

a specific location by their URL. Future plans that require more design adaptations would target the wireless domain, in the topology control for multi-hop packet radio networks, the Voronoi diagram being a structure that abstracts naturally the wireless cells in mobile communications.

1.4 Contributions

The contributions of this thesis are as follows:

- The design of the flat name resolution server federation based on the Delaunay mesh and of the node location algorithms.
- The use of mobile agents for the federation server's deployment.
- A performance comparison of method invocations in the mobile agent and the traditional client/server paradigm.
- Two implementations: Voyager and CORBA based serving as proof of concept

Chapter 2

Federation, Name Resolution and Mobile Agents

This chapter examines the current research related to the federation pattern, reviews some of the existing solutions to the name resolution problem and describes a couple of contemporary mobile agents.

2.1 Federation of Distributed Applications

The concept of federation in computer systems was introduced by Heimbigner and McLeod and was based on the federation of information bases [29]. Their approach was directed to achieve coordinated sharing and interchange of computerized information, rather than relying on centralized control by a centralized (logical) organization. Federation appears central to the construction of systems which allow evolution and can operate in heterogeneous environments with independent administrations.

Heimbigner and McLeod identify four principles that characterise a federal approach to interconnecting computing components [29]:

1. "A component must not be forced to perform an activity for another component. The role of centralised authority must be replaced by cooperative activity among the components supporting protocols."

2. "A component must have 'freedom of association' with respect to the federation. Since the federation is a dynamic entity, components must be able to dynamically enter or leave the federation. Further, a component must be able to modify its shared data interface, adding new data and withdrawing access to previously shared data."

3. "Each component determines the data it wishes to share with other components. Since partial, controlled sharing is a fundamental goal of the federated approach, each component must be able to specify the information to be made available as well as specify which other components may access it and in what ways."

4. "Each component determines how it will view and combine existing data. In a composite system, all access to the underlying data is mediated by a global schema. In a federation, each component must be able to, in effect, build its own "global" schema that is best suited to its needs."

[69] introduces the common abstractions and terminology necessary when attempting to construct a federated system. Two members of the same family of systems: A1 and A2 are said to be federated if A1 and A2 can interoperate in such a way that work together to accomplish the function of A as if they were one system.

If a software system is constructed as a monolith, it embeds a certain family of functionality, scales to a certain extent and is extensible in prespecified ways. But it may be difficult for others than the original design team to extend its capabilities, scale the system for flexible unforeseen needs, or evolve the system. There is a need to have

more flexible architectures to meet future needs and preserve architectural properties. If like systems can interoperate as if they formed a larger whole that provides the same functions of the original systems, then the system's architecture is constructed using the *federation pattern*. Federations are a principal means of scaling systems.

There is a distinction between:

- homogeneous federations composed of similar subsystems with at least the same interfaces and usually the same implementations and expose the same interface as the composed system and
- heterogeneous federations where some amount of mediation is required to coerce some subsystem

In what follows, examples of the federation pattern, where two or more of a kind of a service or system interoperate in a scalable manner are presented. In all cases, they must expose some kinds of protocols formerly used inside themselves on the outside such that their neighbours can interface to them. For instance: if two systems contain objects and relationships internally but then there is a need to have relationships that span the two systems then a relationships protocol is defined that allows objects in one system to be related to objects in another.

In the CORBA architecture the following services are federated:

- naming - such that multiple name spaces can be federated
- transactions - to allow nested transactions along the distributed transaction processing (DTP) reference model of the X/Open Company, Ltd. for prepare to commit protocol
- relationships and proposed topology service, also various hypertext linking services - these services separate the relationships from the data so the data can be viewed

with different collections of relationships. Also, workflow and compound documents use relationships as a kind of federation glue for hooking together diverse kinds of applications.

- query - so a query can be broken into subqueries and operated on by different DBMS.

Typically, interfaces to a system for the purpose of federation need to be exposed. This has not been done in a consistent manner in the CORBA architecture. There is a need for a Federation Design Principle. Federation, in this sense, composes similar systems into a larger like system or alternatively it permits a system to be replicated in function but still provide a uniform interface. A major consideration of federated systems in the DBMS sense is interoperability. This may mean that a parent DBMS with a federated schema provides a uniform interface to new applications. Internally, it may process queries itself and store data but it may also partition some queries to call other leaf (or federated) DBMSs. The DBMS literature [33] often uses the term federation with the connotation that the systems involved are relatively autonomous: they participate in the federation to an extent agreed upon by the federation, but retain autonomy in other respects.

2.2 The Federated CORBA Name Service

The CORBA Naming Service provides the principal mechanism through which most clients of an ORB-based system locate objects that they intend to use. Given an initial naming context, clients navigate naming contexts retrieving lists of the names bound to that context. In conjunction with properties and security services, clients look for objects with certain "externally visible" characteristics, for example, for objects with

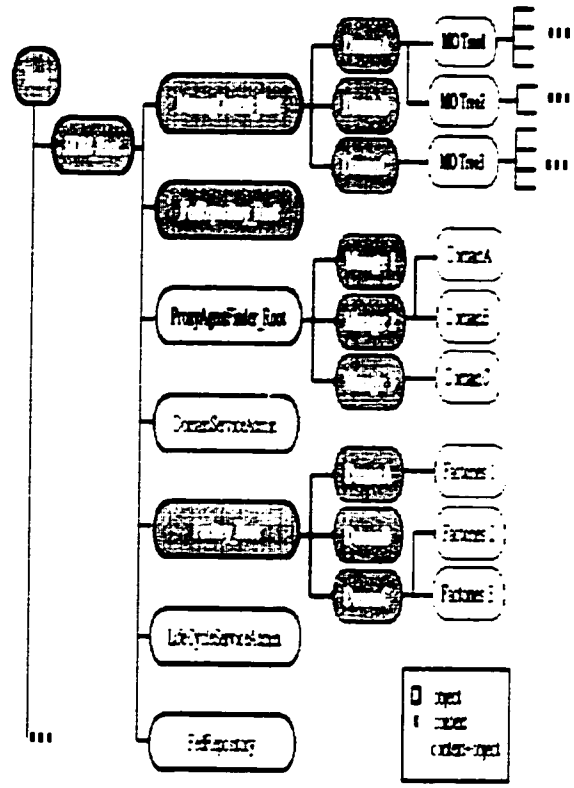

Figure 2.1: JIDM Naming Space Structure

recognized names or objects with a certain time-last-modified (all subject to security considerations). All objects used in such a scheme register their externally visible characteristics with other services (a name service, a properties service, and so on).

The Naming Service provides the ability to bind a name to an object relative to a *naming context*. A naming context is an object that contains a set of name bindings in which each name is unique. To resolve a name is to determine the object associated with the name in a given context.

An example of organization of the namespace for a network management application is presented in (Figure 2.1). There are different naming contexts and object names registered for the managed object, domains, manager objects and their corresponding

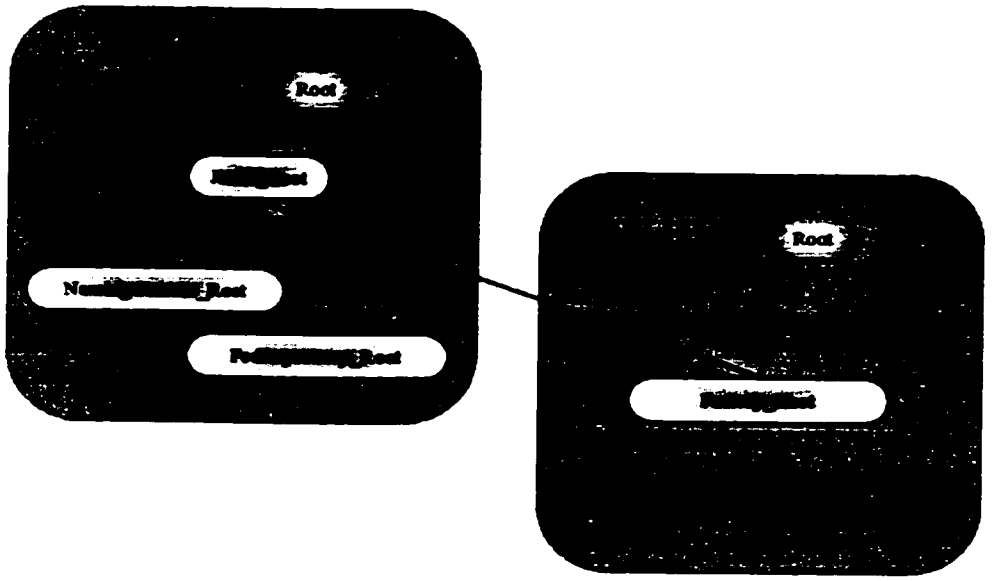

Figure 2.2: Naming Graph Spanning Different Name Servers

factories.

Graphs of naming contexts can be supported in a distributed, federated fashion. The scalable design allows the distributed, heterogeneous implementation and administration of names and name contexts.

The collection of all valid names recognized by the Naming Service is called a *name space*. A name space is not necessarily located on a single name server (since a context in one name server can be bound to a context in another name server on the same host or on a different host). The name space provided by a Naming Service is the association or *federation* of the name spaces of each individual name server that comprises the Naming Service.

Figure 2.2 shows a Naming Service federation that comprises two name servers

running on different hosts. In this example, names relating to the NamingContext_Root and FedRepository_Root contexts are served by one server and names relating to the Factory_Root context are served by a separate server. Client's requests to look up names start in one name server but may continue in another name server's database. Clients do not have to be aware that more than one name server is involved in the resolution of a name, and they do not need to know which server interprets which part of a compound name.

2.3 Lampson's Global Name Service

[34] describes a name service that is intended to be the basis for resource location, mail addressing, and authentication in a distributed computing system. The experience gained in previously deployed similar systems: Grapevine [53] and the Xerox Clearinghouse [77] formed the basis of this design.

The peculiarities of such a system are emphasised in [34] for the first time. A name service is not a general database: the set of names changes slowly, and the properties of a given name also change slowly. Furthermore, the integrity constraints of a useful name service are much weaker then those of a database. Nor is it like a file directory system, which must create and look up names much faster than a name service, but need not be as large or as available. Either a database or a file system can be named by the service, though.

Lampson stated the requirements for such a service: arbitrarily large size, long life, high availability, fault isolation, and tolerance of mistrust. He viewed hierarchy as the fundamental method for accommodating growth, isolating faults and proposed a design similar to a file system. The leaves in the global tree are the values of the

name properties and the intermediary node names are used to encode the path toward a specific child. Distribution and replication were treated in the same manner as in the Grapevine system. Replication is used at the subtree level. Updates originated at directory copies and were propagated by sweeps. A sweep collected the changes made to each directory copy and then distributed the combined changes back to each of them. The directory copies were organized in a ring topology, and if one failed during a sweep the update could be lost. After a failure, the copies needed to be reorganized into a new ring manually.

This name service used hierarchy as its chief method of dealing with growth. The deeper the hierarchy in a naming scheme the less likely names will remain constant as the people will cross partition boundaries more often. The names in this system were aligned with the topology of the servers. This system may be good for routing mail or authenticating users but failed to prove its ability to meet the combined goals of arbitrarily large size and long-lived names. Lampson's global name service articulated the needs of a global resolution system and contributed several ideas, primarily the importance of replication and distribution, but also viable consistency and update methods.

All of the name services mentioned achieved scaling with a hierarchical namespace reflecting the system's structure. This sort of design inhibits the longevity of names which can move from one organizational branch to another and can not be reused for the implementation of a flat system.

2.4 The Domain Name System

The Domain Name System [38], the standard way for resolving the name of any host connected to the Internet, is the most widely deployed resolution system. Due to its extensive field-testing and numerous implementations, DNS has grown to be a mature, successful resolution system that has demonstrated what does and does not work. The design goals for DNS differ from those of Grapevine and Lampson's service, perhaps due to DNS's application and the experience its creators had with existing networks and host naming. Among the goals are:

- Consistency: The namespace should be consistent and free from network specifics.
- Size: DNS replaced a previous resolution system that could not handle the growing number of Internet hosts. Given the frequency of updates, the host name needed to be stored and maintained in a distributed manner.
- Generality: Implementing a global service is costly so DNS is not restricted to one type of data, such as a host's IP address.
- Protocol Independence: While the Internet protocols are most prevalent, a disjoint set of information for each class of network should exist for a name so that it can be resolved to network specific information.

The Domain Name System is a generalized resolution system for hierarchical names primarily used to identify Internet hosts. The DNS name servers are distributed and maintained along administrative boundaries, so names typically reflect organizational structure. A simple delegation of naming authority and maintenance responsibility is obtained by coupling the topology of the name servers to the structure of the namespace.

DNS is composed of a tree of name servers. Each name server has authority over a zone, a logical portion of the namespace replicated on several servers. Each zone contains a list of names and their attributes. If the name is a host, then it will have an address attribute. If it is another zone, then its attribute will be that zone's name server. At the top of the tree is the set of servers that contain the root zone.

The names in the root zone, such as EDU, COM, GOV and CA are commonly known as the top-level domains. A name server at the next level down, for example the CA domain, authoritatively knows all of the name servers that serve the CA zone. It also has a list of zones in the CA domain and their name servers. For example, the root zone contains the CA domain name, which contains the UOTTAWA domain name, and so on. The name servers for the UOTTAWA domain contain addresses of the university's hosts, such as Web, the Web server, and information about subzones, such as the GENIE domain.

A name is the concatenation of delegation, so the name of the university of Ottawa's Web server is *www.uottawa.ca*. The method of replication employed by DNS is simpler than that of Grapevine and Lampson's global name service. Those services were administered remotely by a few trained people. With widely distributed replicas, this required more complex update protocols and structural overhead.

In the DNS model, each server is administered locally by its maintainer. An update to the names in a zone can only be performed on a master server, which reads the name information out of a file. The other zone replicas are secondary servers that either read their information from the same file or use a simple protocol to poll the master server for updates. To discover the attributes of a name, an application queries a resolver. The resolver can reside on the same computer or on a remote one. It knows the address of at least one name server that it can query, possibly receiving referrals to other name

servers.

Through a repetition of queries the resolver will encounter a name server that can resolve the name. Like Lampson's global name service, DNS names are strictly partitioned into a hierarchy. However, a computer moves less frequently than information, and when it does the disruption is not as great.

The Domain Name System serves as an example of a highly successful name resolution system. While it demonstrates proven design choices, it also benefits from extensive operational experience in areas not covered by its technical details. Among these is the administration of a global name database, legal issues surrounding name assignment, and demands by those wishing to own names.

2.5 Uniform Resource Names

The Uniform Resource Names or URNs [58] are globally unique, persistent identifiers that offer a solution to the fragility and limited life of the Web links. An architectural framework for the URNs is defined in the IETF's RFC 2276, [62]. The purpose or function of such a system is to provide a globally unique, persistent identifier used for recognition, for access to characteristics of the resource or for access to the resource itself.

Unlike URLs, URNs are designed to be location-independent. To find the location of a resource named by a URN, the URN needs to be resolved by a service that knows the mapping between URNs and locations. This service is essentially a global lookup table. By changing a mapping in the table, a resource can move without affecting its URN. While some types of URNs may indicate how to find the appropriate resolution service, Web objects should be named by URNs that are pure identifiers, that is, names

with an extremely general syntax giving no indication of how to find resolvers. The less information a name contains the longer it will last and the broader its applicability.

The global scope of URNs requires a global resolution system to enable pure identifiers to be usable. The global resolution system can be divided into two major parts. The first part consists of resolvers, a highly distributed set of servers that can resolve URNs. The other part is the Resolution Discovery Service (RDS), which finds the appropriate resolver for a URN. Because the global resolution system will be an integral, highly utilized part of any information infrastructure, it will need to scale well with Internet growth while remaining responsive and available.

A description of the functional requirements for Uniform Resource Names is given in [59] and an implementation of such a system developed in the Advanced Network Architecture group at MIT is presented in [56]. In order to achieve scalability independent of its contents, the distributed database that stores the names uses a B-Tree structure, a generalized version of a balanced binary search tree that can have an arbitrary branching factor.

The B-Tree has some shortcomings when applied to a distributed system. A root splitting algorithm achieves the tree growth. The root needs to remain constant because a central authority maintains it. All the tree nodes have the same maximum amount of information, but servers can vary in size. The branching factor cannot be changed without rebuilding the entire tree. Like any other hierarchical structure, the B-tree is architecturally limited for obtaining the federation effect. As it will be seen, the Delaunay based solution is better suited to achieve this goal.

2.6 Mobile Code

Mobile code is used for implementation purposes as the new programming paradigm that enhances the traditional client-server model. In the classical client/server paradigm, the application communicates requests and waits for the replies across the network. In mobile environments, the client migrates to the server to directly invoke requests.

Mobile agents are used in the context of the federated resolution system for: federation servers deployment, link construction and update, name resolution and bulk insertions. The driving force motivating mobile agent-based computation is many-fold. First, mobile agents provide an efficient, *asynchronous searching method* for information: mobile clients are launched into the federation nodes and roam around according to the resolution protocol until the targeted name is found as compared with the stored procedure concept from which they developed. Second, mobile agents solve the client/server *network bandwidth* problem. By moving the federation link update transactions associated to any node insertions or deletions from the client to the server, the repetitive request/response handshake is eliminated. Third, mobile agents are robust and fault-tolerant. If a federation server is being shut down, all agents executing there will be warned and given time to dispatch and continue their operation on another node in the federation. Fourth, agents reduce design risk. They allow decisions about the code location (client vs. server) to be pushed toward the end of the development cycle when more is known about how the application will perform. In fact, the architecture even allows for changes after the system is built and in operation. Finally, in the context of server deployment, it avoids the use of the factory pattern in a distributed environment. Mobile agents parked at the federation nodes are used as resolver servers.

This field is evolving so dynamically and so fast that any attempt to map the agent systems will be outdated quickly. However, a few interesting Java and other language

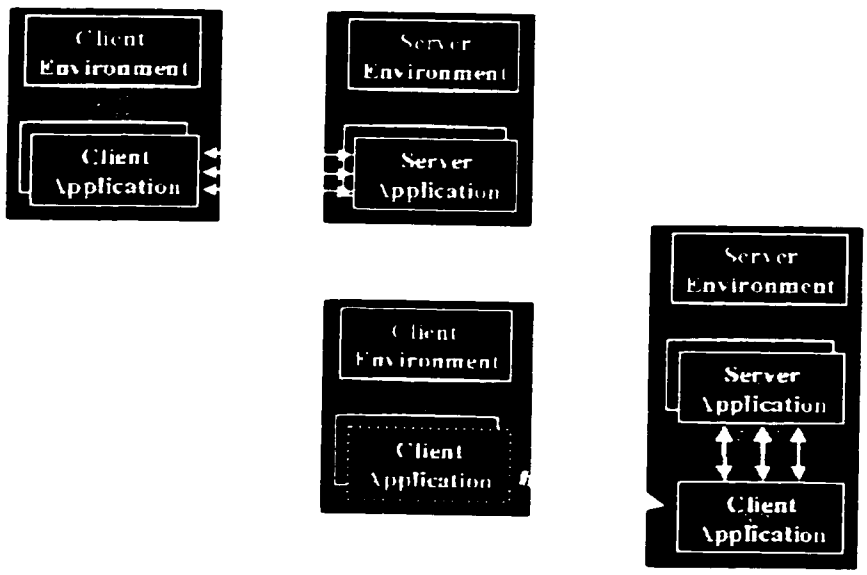

Figure 2.3: Client/Server Versus Mobility

based mobile agent systems are reviewed in the following.

General Magic Inc invented the mobile agent and created the first commercial mobile agent system called *Telescript*. Being based on a proprietary language and network architecture, Telescript had a short life. In response to the popularity of the Internet and later the success of the Java language, General Magic decided to reimplement the mobile agent paradigm in its Java based *Odyssey* [16]. This system effectively implements the Telescript concepts in the shape of Java classes. *Aglets*, [35] created by a research group from IBM mirrors the applet model in Java. The goal was to bring the flavour of mobility to the applet. Aglets are Java objects that can move from one host on the Internet to another. That is, an aglet that executes on one host can suddenly halt execution, dispatch itself to a remote host, and resume execution there. When the aglet

moves, it takes along its program code as well as its data. The aglet is a mobile Java agent that supports the concepts of autonomous execution and dynamic routing on its itinerary.

Mitsubishi's *Concordia* [19] is a framework for the development and management of mobile agent applications which extend to any system supporting Java. Concordia consists of multiple components, written all in Java, which are combined together to provide a complete environment for distributed applications. A Concordia system, at its simplest, is made up of a standard Java Virtual Machine (VM), a Server, and a set of agents.

ObjectSpace's *Voyager* [43] is a platform for agent-enhanced distributed computing in Java. While Voyager provides an extensive set of object messaging capabilities it also allows objects to move as agents in the network. Voyager combines the properties of a Java-based object request broker with those of a mobile agent system. Because this facility that enables the creation of applications using both traditional and agent-enhanced distributed programming techniques this platform was selected for the implementation of the federated name resolution system.

Several organizations promote standards for mobile agent systems. Two of them are the Object Management Group (OMG) and the Foundation for Intelligent and Physical Agents (FIPA). Together they represent almost 1,000 corporate members. OMG endorses the Mobile Agent System Interoperability Facility (MASIF); while FIPA has just called for proposals for agent mobility. It is expected that these standardization efforts have a positive effect on the deployment of mobile agents in commercial systems.

Chapter 3

A Delaunay Based Federated Name Resolution System

3.1 Requirements

Any system that resolves names must satisfy a number of requirements to be successful. It must be designed such that the names are *long lived*. The names should contain very little information that can *change* over time. The name resolution service must be able to change without any visible changes to the names. The system must *scale* to very large proportions. It will need to service hundreds of millions of requests to resolve hundreds of millions of names, and it will need to do so quickly. The resolution system should be *modular* enough so that various parts (resolution algorithms, database structure, protocols, etc.) can be replaced independently of the other parts. It should be *decentralised* (or federated) so that one entity cannot yield absolute control over the entire namespace.

3.2 System Architecture

This chapter describes the federated name resolution system architecture, the mechanisms used for binding and resolving names, for insertion and deletion of servers in the federation and the subsequent adaptive federation link reorganisation. The name resolution function is completely distributed in *Resolver* servers while the federation link update operations that occur during Resolver insertions or deletions are centralised, computed in *Configurator* servers. This is necessary due to the complexity of the Delaunay node insertion algorithm and the fact that the operations on the federation links occur many times less frequently than the naming operations. Also, in the mobile implementation, the Reolvers have to be lightweight components that retain minimum functionality on top of the name related operations.

From the functional decomposition point of view there are three layers, each of them using the services of the one below it. The layered functional decomposition is captured in the table below.

Layer	Function
Adaptive	Join, spawn, bulk insertions, replication
Configurator	Federation discovery, add, delete Resolvers for the Delaunay net
Resolver	Name resolution, add, delete for general graph nodes and links

Table 3.1: Functional Layer Decomposition View

The base layer called the *Resolver* level contains mobile classes injected on several machines. These classes are equipped with facilities for building a general graph connecting the federation nodes and for implementing the distributed Delaunay name resolution algorithm. The Resolvers are managing a local resource set consisting of name bindings

kept in a mapp table or in a database and have links implemented as distributed object references to a number of similar neighbour Resolvers.

Using the generic graph update functions provided by the Resolver level, the *Configurator* layer organises the federation links in a Delaunay mesh. There is a mapping of the Resolvers and their links into the Euclidean two dimensional nodes and edges of a Delaunay mesh. Another mapping assigns names to geometric points and each Resolver server manages the local set of names mapped into points included in its associated Voronoi polygon.

The motivation for using the Delaunay structure for the server federation is given by the following reasons:

1. The adapted searching algorithms are completely distributed among the federation server nodes.

2. There are no root node bottlenecks associated to any alternative tree based hierarchical structures.

3. A balanced distribution of the federation links per federation nodes guarantees a degree-bounded topology.

4. The number of federation nodes can scale to very large proportions. All the server insertion operations affect only locally the federation.

5. The Voronoi cells provide a way of partitioning the namespace and determine the names that are stored in each federation server.

The Configurator server generates the commands necessary to update the federation links when Resolver servers are entering or leaving the federation. In order to accomplish this functionality, the Configurators construct and store the Delaunay graph model of

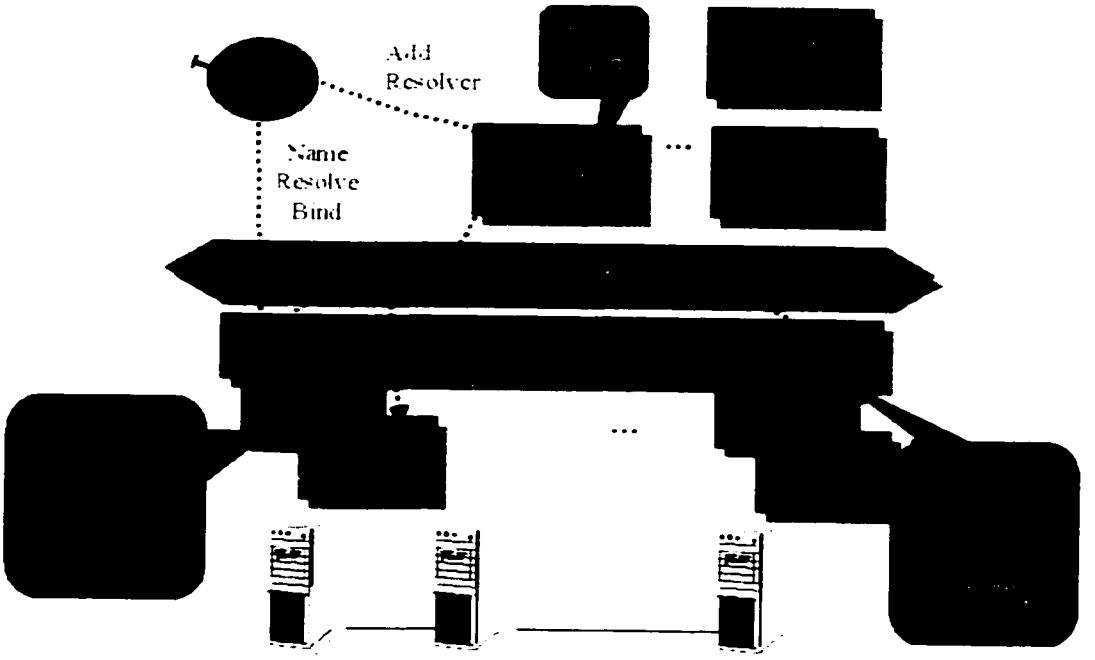

Figure 3.1: CORBA Based Architecture

the whole federation or of the minimal subset that is impacted by the operations for inserting or deleting Resolvers. It performs locally the insertion or deletion in the Delaunay model and generates the sequence of commands that are necessary to be executed to update the federation links. The set of commands consists of add and delete nodes, links or name bindings. The commands are stored in a pool and their execution triggers the invocation of the corresponding remote Resolver methods for link updates. The Resolvers and Configurators are CORBA distributed objects and remote method invocations are used for executing the link update commands. In order to be able to remotely create new Resolvers, the corresponding factories are deployed at system initialisation at each networked workstation.

An alternative implementation for carrying out the Configurators actions that uses mobile agents is presented in Figure 3.2. All elementary link update commands are grouped together to form a single logical unit of interaction forming the *agent load*. The agent (one or multiple) visits each impacted Resolver location and performs locally the actions in its load. For a complete class structure description see 5.1.2.

The motivation for using mobile agents as compared to the CORBA approach for the implementation is manifold.

1. The Resolver servers are directly deployed at the desired workstations, no factory objects being necessary in the process.

2. The link update commands are performed locally such that the repetitive request/response handshake is eliminated.

3. Intermittent connectivity due to slow or unreliable network connection between the Configurator and the Resolver is achieved. Once the agent is launched it can work "off-line" and report the link update result when the connection is re-established again.

A configuration scenario is presented in the general architecture Figure 3.2 as a distinct path from the name resolution. Both CORBA and Voyager Java clients can use the naming service implemented by the Resolvers. In the first case, the requests are translated by the Voyager gateway to CORBA.

Our approach to distribute the name resolution algorithm and centralize the computation of federation link updates is necessary for the following reasons. First, the Resolver class concentrates only the modified Delaunay name resolution logic because it has to have a lightweight implementation for efficient mobility reasons. The other functionality consisting of the modified Delaunay mesh construction and the node in-

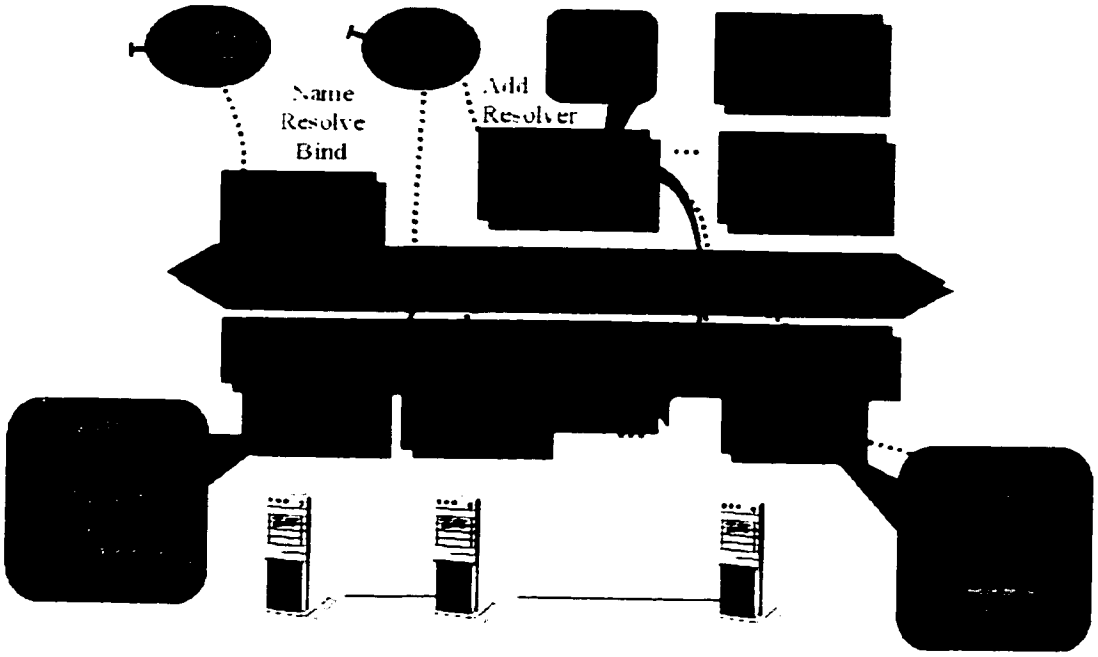

Figure 3.2: Mobile Agents Based Architecture

sertion algorithm are complex and cluttered with special case treatment and had to be concentrated in the Configurator layer. Second, the name resolution operation occurs more frequently than the federation updates. Finally, design changes more specifically a new graph link organization other then Delaunay can be easily obtained within such an architectural decomposition.

The upper layer of the functional decomposition, called the Adaptive level, implements more esoteric features like adaptive Resolver joins or spawns function of the Resolver *health state*. The Resolver health state is parameterized by number of the requests it has to serve, and by its local resource occupancy status. Server replication and mobile bulk binding updates: insertions or deletions are also included in this layer.

There are multiple strategies for the federation server deployment function of the relation between the number of resolvers and the numbers of processors. Let M be the number of machines and N the number of servers. If $M < N$ there are multiple servers on the same machine. If $M = N$ a one to one mapping is achieved. If $M > N$ server replication is used to take advantage of the processing power. An extreme would be to have a fully-replicated federation in which each machine contains replicated servers for all resolvers. The advantages of the last case are enhanced fault tolerance, response time and easier load balancing algorithms.

In the subsequent chapters the mapping function, a description of the Delaunay network pseudocode, of the adapted algorithms, other design alternatives and the motivation for choosing the current implementation through advantages and possible drawbacks of the decision making are all introduced.

3.3 The Delaunay Triangulation and the Voronoi Diagram

This section takes a closer look at the two fundamental geometric constructs, the Voronoi diagram and the Delaunay network that will be used to partition the namespace and organize the federation links. These geometric structures are closely related, and one of them can be extracted from the other. The Voronoi diagram is named after the mathematician M. G. Voronoi who explored this geometric construction in 1908 [73]. However, as early as in 1850, G. L Dirichlet, studied the problem [22]. Accordingly the Voronoi diagram is sometimes named Dirichlet tessellation. The Voronoi diagram is one of the most useful constructs defined by irregular lattices. As it is a widely-used geometrical construction, the diagram has several names.

The construction is used in many distinct fields, and is often named after the person who first used it in a particular field. Geographical interest in the Voronoi diagram originates from the climatologist A. H. Thiessen. He used the Thiessen polygons to define regions that surround unevenly distributed weather stations. Data from each weather station could be presented in the enclosing polygon of the station. The polygons are constructed so that "regions be enclosed by a line midway between the station under consideration and surrounding stations" [68]. The construction is also known as Wigner-Seitz cells [76] (metallurgy) or Blum's transform [14] (biological shape and visual science).

In the two dimensional domain the following geometrical locus defines formally the Voronoi polygon.

Definition 1 *Given a set S of N points in the plane, the Voronoi cell $V(i)$ for a point P_i in S is the convex polygonal region representing the locus of points (x, y) in the plane that are closer to P_i than to any other point P_j in S.*

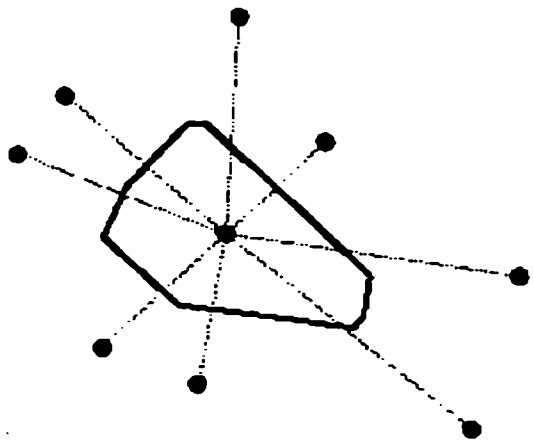

Figure 3.3: Voronoi Polygon

Given two points, P_i and P_j, the set of points that are closer to P_i than to P_j is

given by the half-plane $H(P_i, P_j)$ shown in Figure 3.3 (a).

$$d(N, P_i) < d(N, P_j) : \forall N \in H(P_i, P_j)$$

The half-plane $H(P_i, P_j)$ is defined by the perpendicular bisector b_{ij} of $\overline{P_i P_j}$ and containing P_i. Function d denotes the distance in the Euclidean space.

The locus of points closer to P_i than to any other point in S is given by the intersection of all such half-planes, that is,

$$V(i) = \bigcap_{i \neq j} H(P_i, P_j)$$

As it can be seen in Figure 3.3, a single Voronoi polygon is defined by the lines that bisect the lines between the center point P_i and its surrounding points. The bisecting lines and the connection lines are perpendicular to each other. When this rule is used for every point in the area, the area will be completely covered by adjacent polygons. These polygons partition the plane into regions forming a convex net referred as the Voronoi diagram:

$$VD(i) = \bigcup_{1 \leq i \leq N} V(i)$$

For a set of randomly distributed points the Voronoi diagram is shown in Figure 3.4. It can be noticed that the polygons on the boundary of the area are "open" because they have no neighbouring points in that direction.

The Delaunay triangulation is closely related to the Voronoi diagram. The triangulation is named after B. Delaunay, who first made use of this relationship and called the two structures dual [21].

Definition 2 *The dual of the Voronoi diagram is the graph embedded in the plane obtained by adding a straight-line segment between each pair of points in S whose Voronoi polygons share an edge.*

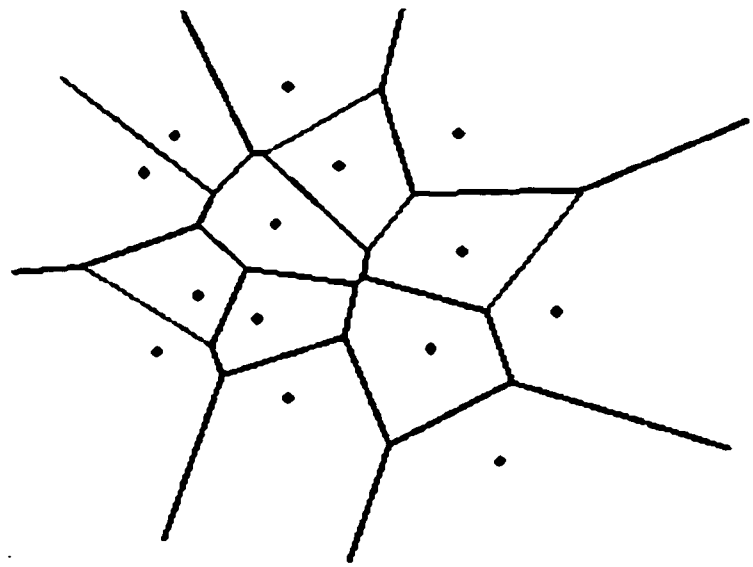

Figure 3.4: Voronoi Diagram

If the Voronoi diagram is used as a basis, the Delaunay mesh is constructed by drawing the lines between the points in adjacent polygons. When the construction is finished a triangular network that covers the whole area is obtained. This result was proved by Delaunay and is stated by the following theorem.

Theorem 1 *The straight-line dual of the Voronoi diagram is a triangulation of S. [21]*

The relationship between the Voronoi diagram and the Delaunay network is shown in Figure 3.5.

In the previous definitions and results the Euclidean distance is used as the metric.

$$d(P_i, P_j) = \sqrt{(x(P_i) - x(P_j))^2 + (y(P_i) - y(P_j))^2}$$

Different other metrics like the Minkowski or Manhattan distance can be considered as well and are relevant to various applications.

The Voronoi diagram is an end in itself in a number of fields. In archaeology, Voronoi polygons are used to map the spread of the use of tools in ancient cultures and for

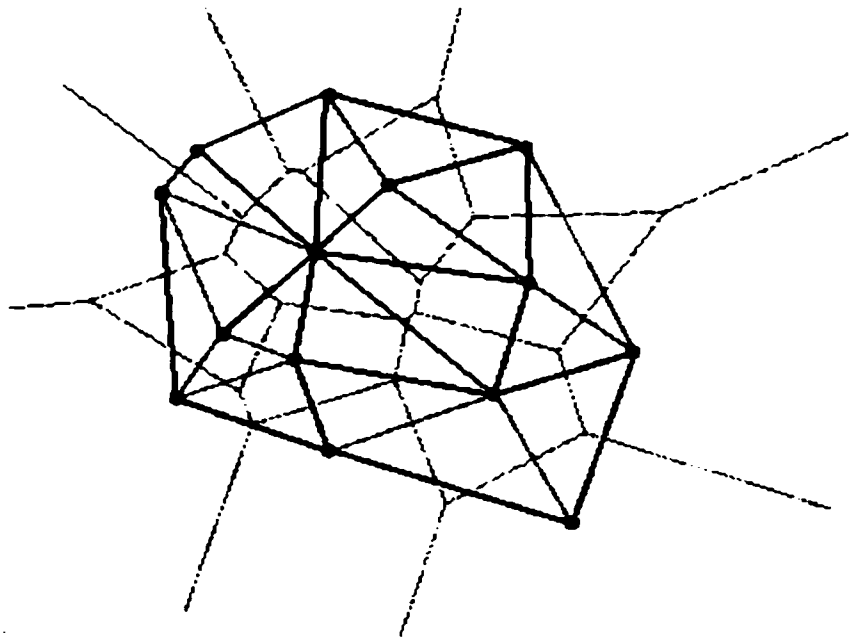

Figure 3.5: The Delaunay Net in Conjunction with the Voronoi Diagram

studying the influence of rival centers of commerce. In ecology, the survival of an organism depends on the number of neighbours it must compete with for food and light, and the Voronoi diagram of forest species and territorial animals is used to investigate the effect of overcrowding. The structure of a molecule is determined by the combined influence of electrical and short-range forces, which have been probed by constructing elaborate Voronoi diagrams.

For many purposes the Voronoi diagram is not needed, but it is desirable to make the Delaunay triangulation directly. The Delaunay triangulation is extensively used in terrain modeling and in the finite element method as an adaptive and scalable structure. There are parallel implementations that link in a mesh millions of nodes. Actually the generation of a Delaunay network in a computer is simpler than the construction of a Voronoi diagram.

3.3.1 The Lawson Criterion

The main rule for the construction of a Delaunay triangulation is formulated in the Lawson criterion. It is also known as the maximum angle-sum test or the circle criterion. According to Lawson a Delaunay triangulation can be incrementally constructed by using a max-min angle test. A theorem states that the maximum angle-sum test determines the diagonal of a quadrilateral that is valid in the Delaunay triangulation. The quadrilateral is formed by two adjacent triangles that are part of the Delaunay mesh and the diagonal is their common edge.

Theorem 2 *In a quadrilateral formed by two adjacent triangles in a Delaunay triangulation, the diagonal goes between two opposite vertices where the sum of the interior angles is greater or equal to* π.

Another equivalent criterion considers the maximum angle-sum determined by cosine. It is useful to observe that, in the quadrilaterals of the mesh, only the angles of the vertices of the interior diagonal can exceed π. This can be used to form an algorithm where only the cosine of the inside angles of the quadrilateral determines whether the diagonal is to be swapped. The angles are calculated in the "diagonal free" corners. By investigating the nature of cosine for these angles, it can be concluded when a diagonal has to be swapped. If the cosine test computed by the inequality below is true, then the diagonal has to be swapped.

$$cos(\alpha) + cos(\beta) < 0$$

The maximum angle-sum test is equivalent to a circle test. It states that a Delaunay mesh consists of non-overlapping triangles where no vertexes in the network are enclosed by the circumscribing circles of any triangle formed by the graph edges. This circle test and invalid diagonal swapping are depicted in the figure below.

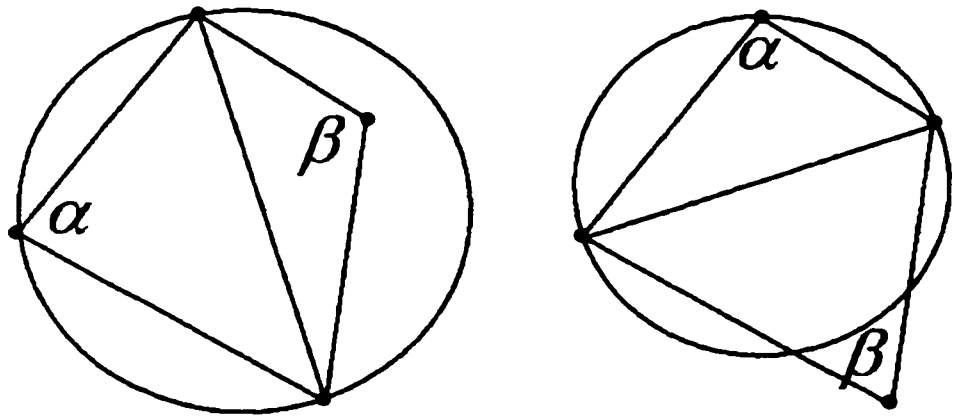

Figure 3.6: Diagonal Swapping

3.3.2 Incremental Construction of a Delaunay Mesh

There are several algorithms for the generation of Delaunay networks and Voronoi diagrams. The incremental algorithm as presented in [37] is described below. The algorithm uses the swapping mechanism from the previous chapter and is incrementally growing the Delaunay network by inserting points into it and swapping the links that do not satisfy the Lawson criterion. The "initial value" for the triangulation is a valid Delaunay network (at least one triangle). For each point included in the network, the network will be rearranged until the Lawson criterion is met for all the triangles in the network.

The algorithm consists of four steps.

1. An initial triangular network is created. The triangular network of the convex

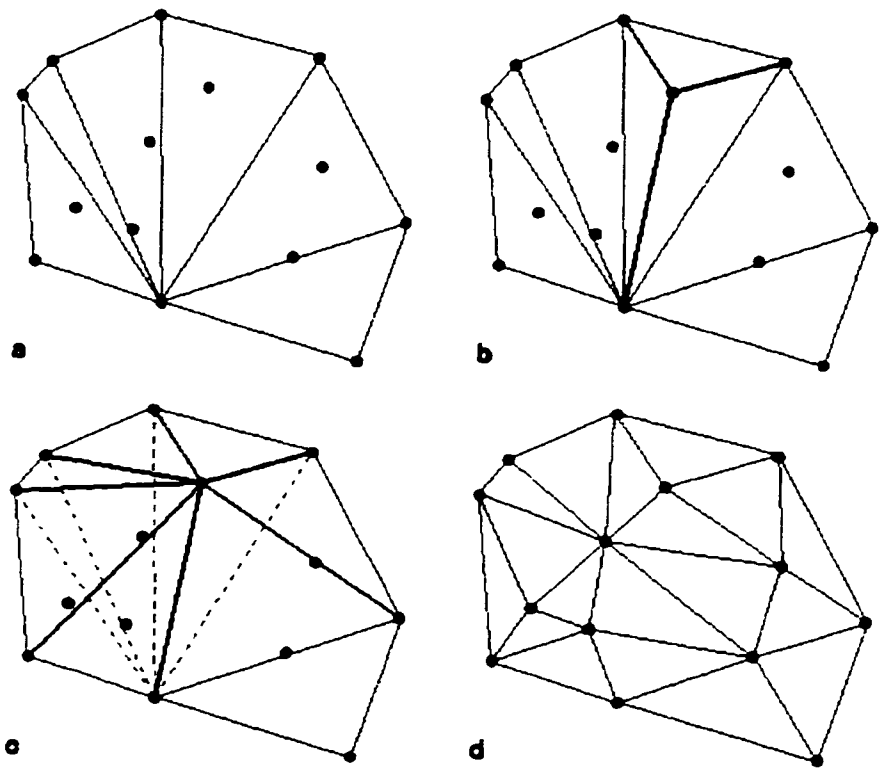

Figure 3.7: (a) An initial network. (b) Insertion of the first point. (c) The inconvenient edges are swapped. (d) Final lattice after the last point is included.

hull is used for this purpose (Figure 3.7.a). This network has to meet the circle criterion (Lawson).

2. The first point of the interior area is included in the network. The point is connected to its enclosing triangle by three new triangle edges between the point and the vertices of the triangle (Figure 3.7.b).

3. The quadrilaterals, which have got the "old" edges of the enclosing triangle as a diagonal, have to be tested by the maximum angle-sum rule. If they do not meet the criterion, their diagonals are swapped and the new, opposite edges to the inserted point will be examined as diagonals in their quadrilaterals. The results

of the swapping operations are shown in Figure 3.7.c.

4. The Delaunay network now contains one more point. All the remaining points will be included in the network in exactly the same way (Figure 3.7.d).

This section is concluded with the following theorem from [49] giving the complexity of the construction algorithm.

Theorem 3 *The Delaunay network of a set of N points in the plan can be constructed in $\mathcal{O}(N log N)$ time and this is optimal.*

3.3.3 Name Resolution

Name resolution is the key function of the federated server system. Given an initial resolver server reference, it is desired to traverse the federation graph until the resolver server that contains the name is retrieved. To resolve a name or retrieve a server object reference, a hill climbing like technique is applied.

Definition 3 *A hill climbing algorithm is a graph search algorithm where the current path is extended with a successor node which is closer to the solution than the end of the current path [23].*

In simple hill climbing, the first closer node is chosen whereas in steepest ascent hill climbing all successors are compared and the closest to the solution is chosen. Both forms fail if there is no closer node. This may happen if there are local maxima in the search space, which are not solutions. There are no local maxima for the searching algorithm used in the name resolution system.

The mapping of the names into points and of the resolver federation into the Delaunay network is used to translate the name resolution into the point location problem in

the computational geometry domain. For many searching problems it is necessary to move from one node to another in the Delaunay mesh. This section introduces some theory and algorithms for moving between two nodes in the Delaunay network.

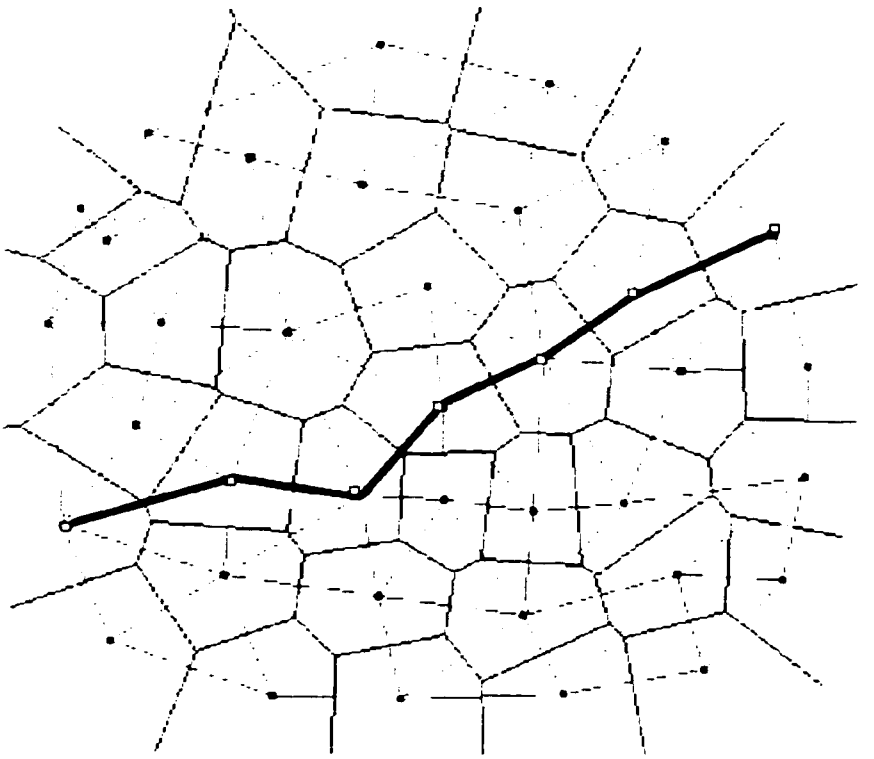

Figure 3.8: Name Resolution Path

There are two variants for the computation of a short path between two nodes in a Delaunay triangulation using hill-climbing algorithms, the direction search and the distance search. Both methods implement a step by step move to find the path between two endpoints. At each stage a point in the Delaunay network is reached that is closer to the point we are searching for. The two methods differ only by the way in which a criterion minimizing distance or angle (direction) toward the target makes the next

closest node selection.

In the first case, distances from the initial point to its neighboring nodes are calculated. At least one of the neighboring points lies closer to, or is equal to the searched point. The neighboring node that is the closest to the searched point is chosen as the next initial point. The routine repeats itself until the searched point is reached. When the distances from the points around the initial point are calculated successively in trigonometric order, the calculations may be terminated when the distance has reached a "minimum" and starts growing again. This algorithm is summarized below:

1. Compute distances from the current node in the path to the target point.

2. Select as the current node the neighbor node that is the closest to the target.

3. Stop if the target is reached, else go to step 1

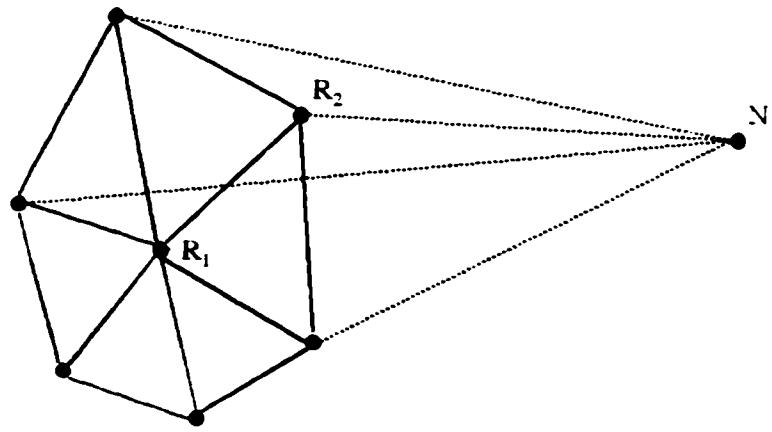

Figure 3.9: Moving One Step Closer in a Distance Search

In Figure 3.9, R_2 is the node among the neighbors of N that has the shortest distance to the searched point N. Consequently, R_2 is added to the path.

In the second variant, the direction to the searched point is calculated. Further, directions from the target to the neighbor points are calculated. When the two directions that span the direction to the searched point are localized, further calculation is superfluous. The point opposite to the initial point at one of the neighboring direction lines is chosen as the next initial point. The routine repeats itself until the searched point is reached.

Among the two variants for point location, the distance algorithm is selected and adapted for name resolution due to the overhead induced by the calculation of the angles. From a computational point of view it is cheaper to compute distances than angles. An outline of the pseudocode is given below.

```
Resolve( InitialResolver, key )
1 if  key is in this InitialResolver mapp table
2   return associated value
3 else
4   (x, y) = mapp( key )
5   for InitialResolver and for all its neighbours
6     compute the square distance to (x, y)
7   NextResolver = select the minimum value
8   if this Resolver is the closest one
9     throw NotFound
10  else
11    Resolve( NextResolver, key )
```

To resolve a name, meaning to obtain the associated value there are 2 actions to be done: obtain the names corresponding point using the mapping function and then, starting from an initial proxy Resolver server object reference, the next Resolver in the resolution path is computed. This recursive process will finally lead to the resolver that contains the name in its local database. If the name is not registered in the system the traversal occurs in the same way as if it would exist. The path arrives to the final Resolver that is supposed to contain the node and since the local search fails an exception is thrown as in step 8.

3.4 The Mapping Function

The role of the mapping function is to provide a global mapping between the names seen as plain strings and the two dimensional Euclidean points. All the names that are represented by points inside a Voronoi polygon are stored in the Resolver that corresponds to the associated Delaunay vertex in the federation. In order to have a locality property in the namespace, the mapping function should translate names that differ only by a limitted number of letters to neighboring points. An example of such a function extracts all odd indexed characters from the initial name string (function 3.1) and concatenates them into a new string (function ??) that is used for the computation of the X value of the corresponding point. Any string to integer mapping (function ??) can finalize the translation of each substring. Similarly, the even chars are processed for the Y projection.

$$Map_x : \ Integer \mapsto STRING_X \tag{3.1}$$

$$Map_y : \ Integer \mapsto STRING_Y \tag{3.2}$$

The mapping function is known by the federation Resolver servers and it provides information used to uniquely select the next hop in the name resolution path. One of its advantages is that it does not link any name by the physical server location that contains the resolver object, and if the server is deactivated, all the locally impacted names will be redistributed among neighbor federation Resolvers. This approach takes for granted the assumption that the communication overhead time between different servers is approximately the same that is the case in a LAN. Over WANs or Internet, where this assumption is not true, server replication will be used to compensate for the differences between node communication time.

3.5 Federation Update and Link Load Balancing

This chapter describes the features included in the Configurator layer. Pseudocode is given for the edge reorganisation or swapping, for the centralised annotated Delaunay model node insertion and deletion and finally for the distributed Resolver insertion and deletion.

From the Resolver federation point of view, the theorems and considerations presented in this chapter are very important to demonstrate that the Resolver insertion process perturbs the network only locally thus making the algorithm suitable for a distributed implementation. Only a small number of federation links implemented by Resolver object references have to be updated when a new Resolver is introduced.

3.5.1 Adaptive Federation Link Swapping

This section presents a detailed discussion of the reorganisation process that is necessary when a new node is included in a network. It is a recursive process for edge reorganisation that needs to be invoked after the insertion of a new point that usually results in some swapped edges. Every quadrilateral that is adjacent to the inserted point has to be tested by the maximum angle-sum rule, the Lawson test or some other criteria. If the diagonal is swapped, two new adjacent quadrilaterals appear and must be tested.

The algorithm can be formulated short and precise by a recursive procedure not cluttered with degeneracy or any other special cases:

```
SwapTest( diagonal )
1 Quadrilateral = EnclosingQuadrilateral( diagonal )
2 if Quadrilateral.isSwapped( diagonal )
3   SwapDiagonal( diagonal )
4   SwapTest( Quadrilateral.edge1 )
5   SwapTest( Quadrilateral.edge2 )
6   SwapTest( Quadrilateral.edge3 )
7   SwapTest( Quadrilateral.edge4 )
```

The parameter in the function call represents the diagonal of the quadrilateral to check. In line 2: Quadrilateral.isSwapped(diagonal) performs the maximum angle-sum test or the Lawson criteria as it is called in computational geometry language.

SwapDiagonal deletes the current diagonal of the quadrilateral and adds the other one as a link in the federation mesh. Edges on the convex hull are marked and are not submitted to the swapping test.

3.5.2 Node Insertion in a Delaunay Mesh

From the name resolution federation point of view, in order to have a scalable solution, there must be a possibility of connecting new Resolver servers to the system such that an increasing number of name bindings can be stored.

Node insertion is a fundamental operation of the incremental Delaunay construction algorithm described in section 3.3.2. While the previous section discusses the geometrical conditions for edge swapping, this section describes the algorithms, methods and special features related to the insertion.

The point insertion is of vital importance for the incremental algorithm used in the construction of the mesh. But the same rules can also be used to update Delaunay nets constructed by other methods. The conditions are that the mesh has to meet the circle criterion before the inclusion, and that the data structure can be updated according to the rules. This discussion on the geometrical consequences of the point insertion is of major importance for the understanding of the Delaunay network. The theorems explored in this section are also useful concerning the merging of triangular networks and the deletion of points from a mesh.

The insertion of a point interior to the mesh influences the edges in the triangular network. The orientation of the new and swapped edges after the new node D is inserted into the triangular cell formed by ABC can be viewed in the following figure. The dotted lines show the triangular network before the node insertion while the solid ones represent the final mesh after node D is added.

In the Resolver federation case the links are implemented by distributed object references stored in the nodes adjacency structure and the following theorems show how they are modified by the insertion of a new Resolver.

The point insertion generates the refinement of the triangulation. When D is inserted

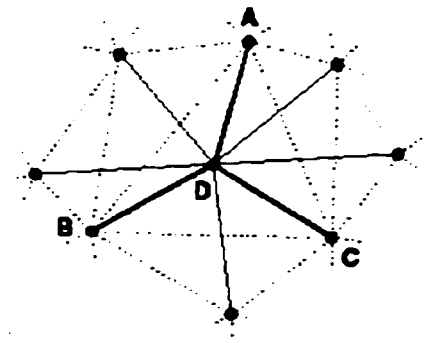

Figure 3.10: New Links in a Node Insertion in a Delaunay Network

into the initial mesh, three new edges are constructed. This means that the old triangle is split into three new ones. These edges are all valid in the newly created triangulation. Some of the surrounding edges have to be swapped and located with one end in D. A swapped edge is a member of the new triangulation, and is consequently only swapped once.

When the point D is inserted into ABC, three new edges are constructed to connect D to the initial mesh. The original triangle ABC will be split into three new triangles ADB, ACD and DCB. The circumscribing circles of these new triangles have to be examined to localise any interior points. It is known from the circle criterion that no fourth point, except D, can be situated inside the circumscribing circle of ABC. The effects on the three new inserted edges are described by the following theorem [37].

Theorem 4 *The three new edges constructed to connect the new point to the triangular network never change during the edge reorganisation which is caused by the insertion of the point.*

There are also effects on the existing edges of the network as described in the next theorems whose proofs are presented in [37].

Theorem 5 *An edge influenced by an insertion point will always be located with one end in the inserted point.*

Theorem 6 *An edge influenced by an insertion point is never swapped more than once as the result of the current point insertion.*

An important result of these observations is that in each quadrilateral only the edges opposite to the inserted point have to be examined. These edges have to be checked by the maximum angle-sum test on their enclosing quadrilaterals. This observation is of great value when efficient algorithms are to be implemented.

Concerning the extension of the region influenced by an inserted point, various authors claim that the insertion of a point into the Delaunay network is a rather local process, without showing how local it is. The size of the area, influenced by an inserted point, depends very much on the distribution of the points in the mesh. The extension of the influenced area, and how the point distribution influences the extension is discussed next. The term distribution means distribution in the xy-plane. An upper limitation is given by the following theorem.

Theorem 7 *The area influenced by an inserted point never exceeds the circumscribing circles of the three new initially created triangles formed when the old triangle is split.*

An exact limitation of the influenced region is shown in order to answer more precisely the question of localizing the triangles that are influenced by the insertion of D. The calculation can be done before D is inserted.

Theorem 8 *All triangles of a Delaunay triangulation which have the inserted point interior to their circumscribing circles are influenced by the point insertion.*

When the triangles influenced by D are determined, they can be deleted. The enclosing polygon of D in the new triangulation shown in Figure 3.11 is the result of this deletion. This polygon indicates the limitation of the influenced region. The dotted edges are identical to the edges swapped during the point insertion of D.

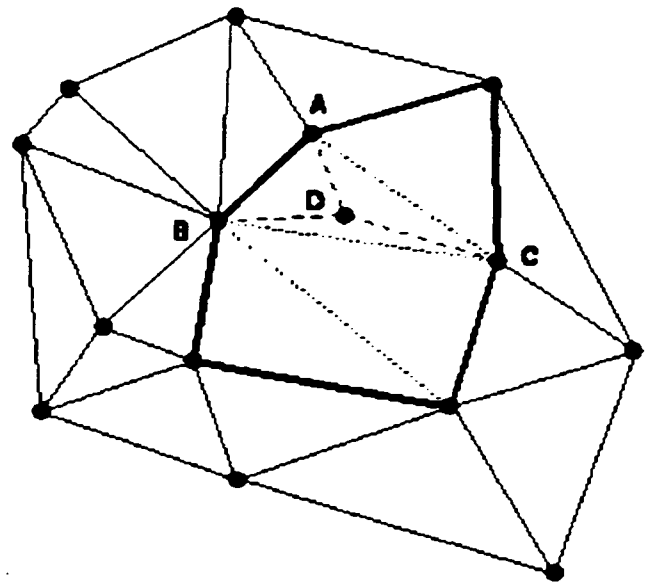

Figure 3.11: The Enclosing Influence Polygon of D

A different issue concerns the fact that the succession of the inserted points has no influence on the final result. On the other hand the succession may influence the time consumption to some degree. Because of the built-in point handling structure, it is advantageous to split the large triangles early in the process. However, any calculation for the determination of the most favorable split point takes time. In the program the edge table decides the succession of the points to insert. The first edge of a triangle that encloses additional points is chosen. When a point is inserted, the algorithm moves on to the next edge. The localization of the next point to insert is very fast by this random method, and practical experiments show that the insertion spreads over the

triangulated area quite fast.The pseudocode for the annotated insertion in a Delaunay mesh is presented next.

The theorems and considerations presented before are very important to demonstrate that the insertion process perturbs the network only locally and make the algorithm suitable for a distributed implementation. The number of federation links that have to be udated when a new node is inserted or deleted does not depend on the number of federation nodes. The insertion operation occurs in $\mathcal{O}(1)$ time.

```
[Commands, TouchedNodes] Insert( DelaunayMesh, Point )
1  Commands = null // start with an empty set
2  TouchedNodes = null
3  DelaunayMesh.addNode( Point )
4  Commands.insert( "ADD Resolver" )
5  if DelaunayMesh.nodes.size < 2
6    return
7  else if DelaunayMesh.nodes.size equal 2
8    edge = DelaunayMesh.addEdge( nodes[0], point )
9    Commands.insert( "ADD edge" )
10   return
11 else if DelaunayMesh.nodes.size equal 3
12   edge0 = DelaunayMesh.addEdge( nodes[0], point )
13   edge1 = DelaunayMesh.addEdge( nodes[1], point )
14   Commands.insert( "ADD edge0 edge1" )
15   return
16 else
17   Triangle = FindEnclosingTriangle( point )
```

```
18   edge0 = DelaunayMesh.addEdge( Triangle.node[0], point )
19   edge1 = DelaunayMesh.addEdge( Triangle.node[1], point )
20   edge2 = DelaunayMesh.addEdge( Triangle.node[2], point )
21   Commands.insert( "ADD edge0 edge1 edge2" )
22   TouchedNodes.insert( Triangle.node[0],
23                        Triangle.node[1],
24                        Triangle.node[2] )
25   SwapTest( edge0 )
26   SwapTest( edge1 )
27   SwapTest( edge2 )
28   return [Commands, TouchedNodes]
```

In procedure SwapTest, the links that lie on the convex hull are marked and are not to be swapped. Another modification for accomplishing the name resolution was the reorganization: deletion and reinjection of all the names in the touched nodes. This ensures that each node manages exactly the names in it's associated Voronoi cell. The Voronoi diagram changes with a node insertion and the impacted names have to be reassigned to the touched resolvers accordingly.

3.5.3 Delaunay Mesh Node Deletion

The removing of nodes is an important feature of a program system that handles meshes. Nevertheless, many existing implementation programs do not support this feature. The insertion of points into triangular meshes has been well studied, while the problem of removing points has been given less attention according to [37]. Static triangulation methods usually rebuild the complete network when there are any changes in the data set, insertion or deletion.

The only algorithm for point deletion that was found is one that is briefly described by Midtbo [37]. It is named the basis algorithm, and it is described in detail. There are also two variants of the basis algorithm where no edges are removed, but are swapped according to the constraints from the basis algorithm. These algorithms are named the reversion algorithm and the modified reversion algorithm.

When a point is inserted into a Delaunay network, it is known from the theorems in the previous chapter that the edges influenced by the insertion will be located with one end in the inserted point. These edges stretch out like spokes to an enclosing polygon. This is not necessarily a convex polygon. Still, the triangular structure of a Delaunay mesh usually results in a figure that is pretty close to a convex polygon. Because the mesh that lies outside the polygon is untouched by the point insertion, it is obvious that deletion of the same point will have no influence on this mesh.

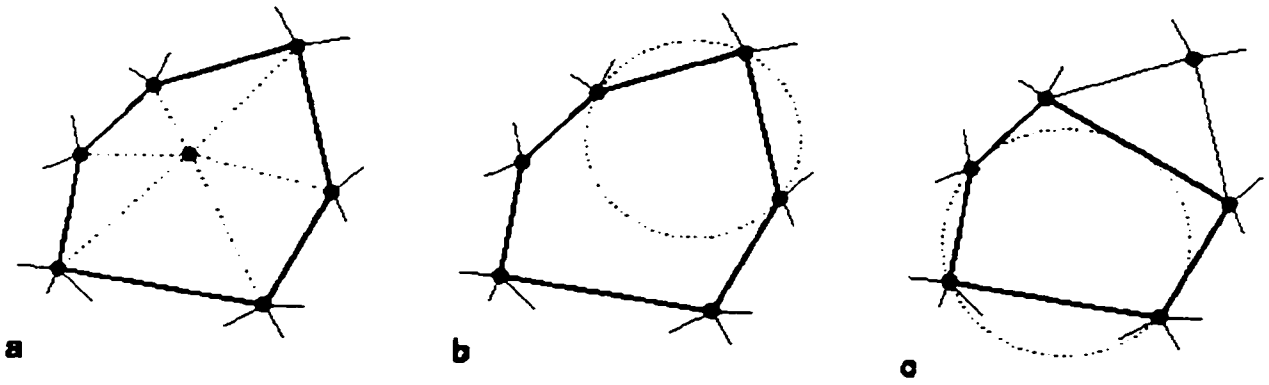

Figure 3.12: Node Removal from a Delaunay Mesh

The basis algorithm is introduced below.

1. The enclosing polygon of the current point is found and its interior triangles are removed (Figure 3.12.a).

2. Two consecutive edges along the polygon form a triangle. The circumscribing

circle is calculated for each consecutive pair of edges (Figure 3.12.b).

3. The triangle, corresponding to the smallest circle, is chosen. The triangle edges are added as new federation links and the polygon is adjusted.

4. Go to step 2 if the mesh that lies inside the polygon is not complete.

It is easy to see that the smallest circle along the polygon does not contain any other points of the same polygon. This is because if the circle covers one of the other points, it is possible to make a smaller circle. The same constraint prevails for each of the shrunken polygons during the process. The last polygon will be a single triangle. The final network meets the circle criterion for Delaunay triangulation because none of the smallest circles during the operation cover any of the other points in the network.

```
[Commands, TouchedNodes] Delete( DelaunayMesh, point )
1 if DelaunayMesh.nodes.size <= 3
2   return
3 node = DelaunayMesh.findNearest( point )
4 DelaunayMesh.removeNode( node )
5 TouchedNodes = allNeighbors( node )
6 for all node[i] in TouchedNodes
7   DelaunayMesh.removeEdge( node[i], node )
8 Edges = TouchedNodes.retriangulate
9 return [Commands, TouchedNodes]
```

The pseudocode for this function is presented above. The commands to be invoked for updating the distributed Resolvers links are generated in the SwapTest procedure as in the insertion case. Deletions from a Delaunay network that has three nodes or

less are not allowed. Again, like in the insertion case, the links on the convex hull are marked and are not supposed to be swapped.

Chapter 4

Distributed and Mobile Federation Service

This chapter discusses the issues related to the federation server deployment using distributed and mobile code, federation discovery and update. Federation discovery is used to build the model contained in Configurators. Commands are generated by the centralised execution of the update algorithms operating on the federation model inside Configurators. The commands are used to update the names and links of the distributed Resolver objects composing the federation.

4.1 Server Deployment Using Mobile Code

This section shows how a remote object is instantiated through the use of mobile code, how messages are sent to it via proxies and how this remote creation is used with a standard neutral ORB, the ObjectSpace's Voyager product. The Java language supports a feature called interfaces. An interface contains no code or data. Instead, it defines a set of method signatures that must be defined by any class that implements the interface.

A variable whose type is an interface can refer to any object whose class implements the interface. For example, if the class Resolver implements IResolver, it is legal to write:

```
IResolver resolver = new Resolver(); // resolver refers to local object
```

Voyager leverages this language feature to simplify distributed computing. When a remote object is constructed using Voyager.construct(String implementation, String destination), a proxy object whose class implements the same interfaces as the remote object is returned. Voyager dynamically generates the proxy class at run time. The proxy can receive messages, forward them to the object, receive the return value, and pass the return value on to the original sender. If the remote object throws an exception, the exception is caught and passed back to the proxy, which throws it to the caller. For example, after the following line of code is executed, the resolver variable references a proxy that implements IResolver and is associated with the remote Resolver object.

```
// resolver refers to local proxy associated with a remote object
IResolver resolver = (IResolver) Voyager.construct( "Resolver",
                                                    "remotehost:8000" );
```

The Figure 4.1 illustrates this effect.

Any messages sent to the proxy via the resolver variable are automatically sent to the remote object as it is demonstrated below.

```
// executed by the remote object as if the object were local
resolver.bind( "Sun", "java" );
```

The Voyager implementation of mobile agents has the property that the header stub classes containing the marshalling code for the proxy classes are invisible to the user and generated on the fly in the daemon, in this way reducing the application complexity.

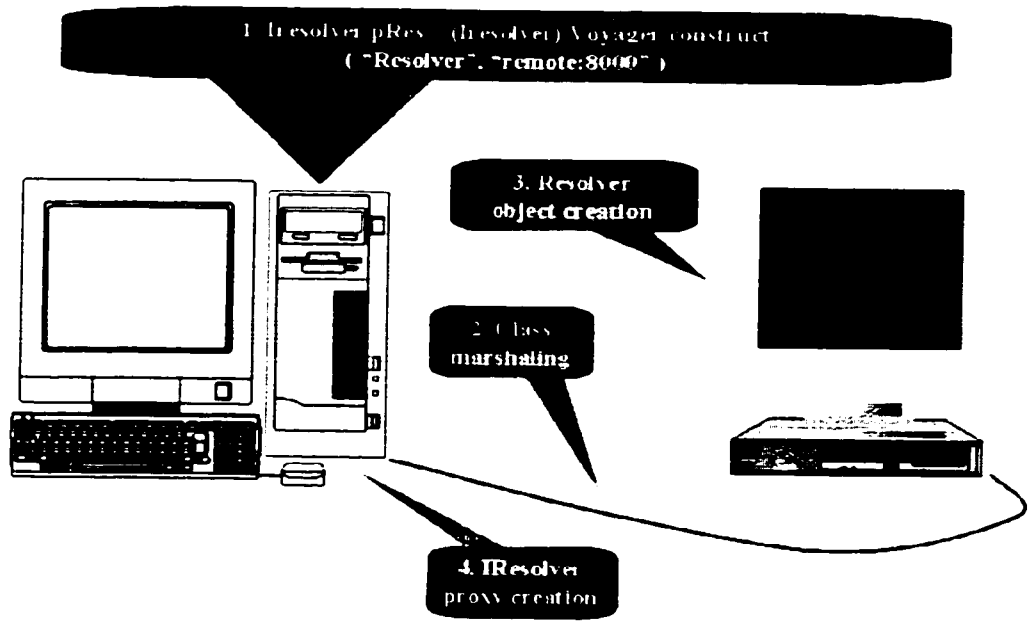

Figure 4.1: Server Deployment Using Mobile Code

The only parameters that are present in a remote server object instantiation are the class name to be created and the remote environment: machine name and port on which the server object is to be instantiated. A prerequisite for efficiency reasons is that the marshaled class should have a lightweight binary footprint. The actions that are actually performed are given in pseudocode:

```
ProxyReference Construct( ClassName, Location )
1 bytecodes = Serialize( ClassName )
2 Marshall( bytecodes, Location )
3 Create( server object )
4 Create( proxy object )
```

```
5 return reference to proxy object
```

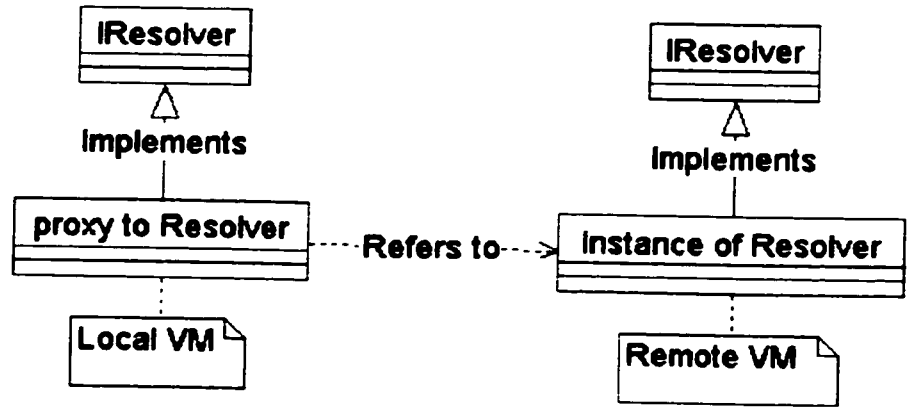

Figure 4.2: The Proxy Pattern

In this way, it is avoided to use the factory pattern which raises the problem of factory object classes deployment at remote locations.

4.2 Federation Discovery

Sometimes it might be necessary to have a centralised view of the system. The system should have features similar to an air-traffic controller environment that allows real-time visualization and control of distributed objects in the federation. This is especially useful when debugging the program or when a superuser wants to have a global view of the resolver servers and their states.

There are several ways to achieve this goal. The first option is a modified classical breadth-first algorithm such that it achieves distribution through the use of the proxy pattern. A more elaborate version considers a mobile agent that visits the nodes in a breadth-first way and updates a centralised graph model of the federation. The graph

model can be carried along with the agent or be placed at a given location and accessed through proxy references by the agent. An outline of this modified breadth-first algorithm is presented below.

```
[Graph] FederationDiscovery( InitialResolver )
1 Resolver = InitialResolver
2 Graph = null
3 VisitorQueue.insert( Resolver )
4 while VisitorQueue.notEmpty
5     Resolver = VisitorQueue.get
6     MobileAgent.moveTo( Resolver )
7     Graph.addNode( Resolver )
8     Graph.addEdges( allNeighbors( Resolver ) )
9     for allNeighbors( Resolver )
10        if NeighborResolver.notTouched
11          NeighborResolver.Touch
12          VisitorQueue.put( NeighborResolver )
13 return Graph
```

For speeding up the discovery process, multiple mobile agents could be used. They start from a set of initial resolvers and follow different edges, constructing together the shared graph model. In this case, the access to the shared resource consisting of the queue and the graph model is guarded by semaphores. A high degree of parallelism is achieved.

It is necessary for the mobile agent to have a lightweight implementation such that the serialization and marshalling time does not become prohibitively long.

4.3 Resolver Insertion

This section presents an outline of the Resolver insertion algorithm. Before the insertion, a model of the federation is built using the federation discovery feature. This model consists of a graph corresponding to the deployed Delaunay federation mesh. Actually, it is sufficient if all the Resolver's mapped points are known. The graph links can then be determined from the nodes. Though the Delaunay triangulation for a set of points is not singular, conventions can be added such that the federation links are uniquely determined. The only indeterminate case is when four points from the initial set are on the same circumscribing circle. In this situation both diagonals of the quadrilateral formed by the points are satisfying the Lawson criterion and consequently are valid Delaunay edges. An example of such a convention selects between the two diagonals the one that makes the smallest angle with the horizontal axis, angle measured in the trigonometric sense.

Once the federation model is constructed, new Resolvers are inserted as points in the model and the subsequent federation link update commands are generated. These commands are then executed on the different remote server nodes as shown in lines 2 and 3 from the pseudocode presented below. In line 1 the insert method returns as output parameters the set of commands generated and the nodes that have to update the links.

Apart from the update commands, the insertion in the model generates also the set of nodes whose adjacent links were modified by the algorithm. All the names contained in the Resolver servers corresponding to this *touched nodes* have to be reinjected into the federation. This is due to the fact that any Resolver contains all the names that are mapped in it's associated Voronoi polygon and for the touched nodes the Voronoi cells are modified. Lines 5 to 10 describe the reinjection process. In line 6, the referred

binding is a name-value pair and getAllBindings returns all local bindings contained in the specified resolver.

```
InsertResolver( Federation, InitialResolver, Point )
1  [Commands, TouchedNodes] = DelaunayMesh.insert( Point )
2  for each Command in Commands // adds node, creates or
3    execute( Command )          // adjusts links as appropriate
4  TouchedBindingsList = null
5  for each Resolver in TouchedNodes
6    TouchedBindings = Resolver.getAllBindings
7    TouchedNamesList.add( TochedNames )
8    Resolver.removeAllNames
9  for each Binding in TouchedBindingsList
10   InitialResolver.bind( Binding.Name, Binding.Value )
```

In Figure 4.3 are presented the link updates that occur when a new Resolver is inserted in the federation. The link between Resolver 2 and 3 is deleted and new edges connecting Resolver5 to Resolvers 1, 2, 3 and 4 are created. Also the figure depicts the generated commands and the touched nodes list.

Another design option is to include the Delaunay insertion logic into the Resolvers. This variant avoids the centralized approach and does not construct any federation model. However, the insertion algorithm is complex and is better to be executed in the Configurator server. Also, the Resolver classes being mobile have to be lightweight components retain only the resolution function. Finally, it is estimated that the name resolution requests that are implemented by a distributed algorithm outnumber the resolver insertions and deletions that can be seen as an administrative-provisioning operation. A mixed approach in which the FederationDiscovery only returns the federation

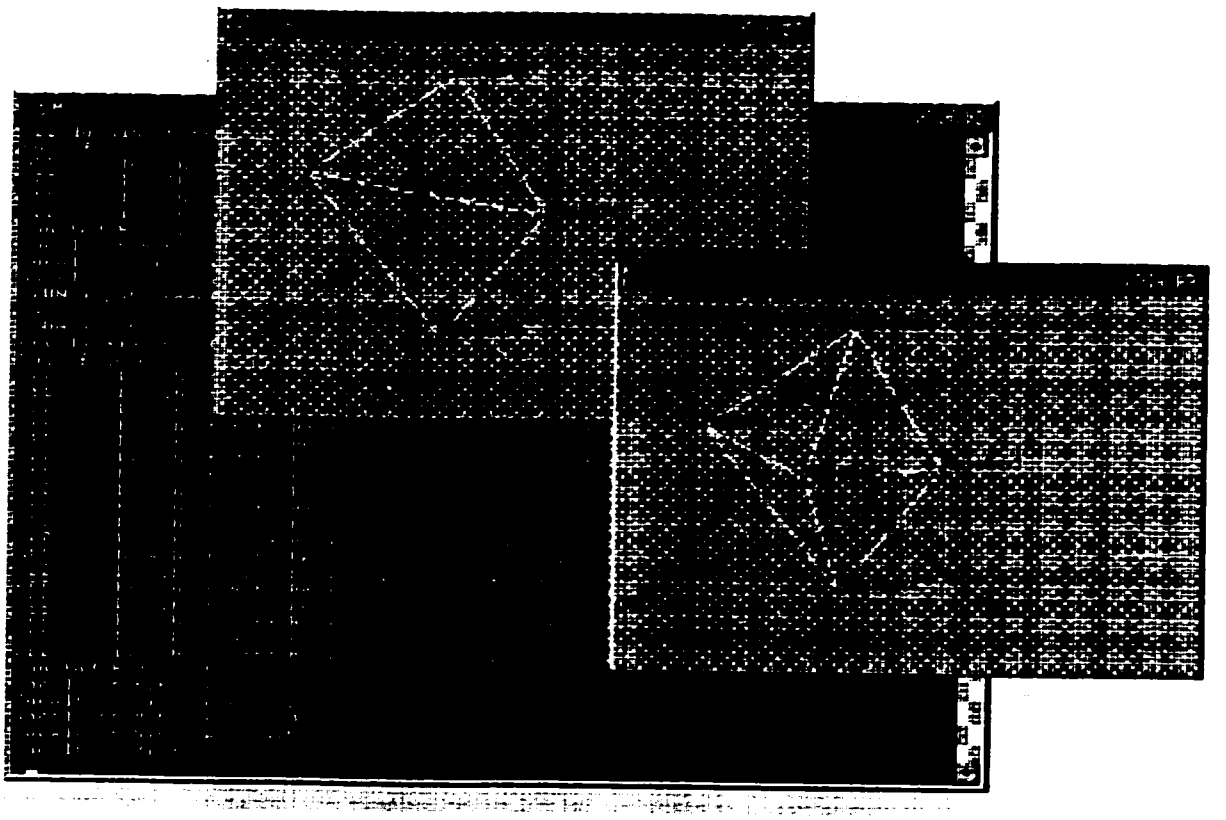

Figure 4.3: Link Updates at Resolver Insertion

subgraph that is impacted by the insertion, determined by the theorems in the Delaunay node insertion chapter is the best solution.

The Delaunay insertion algorithm has some notable advantages. Insertion does not require a downward traversal of any tree thus avoiding the root node bottlenecks associated with such a structure. Federation link update is a localized operation that takes $\mathcal{O}(1)$ time. Furthermore, storing a variable number of name bindings in all nodes of the federation is beneficial for resource utilization and load balancing.

4.4 Resolver Deletion

The Resolver deletion has an analogous treatment as the previous Resolver Insertion algorithm. The only difference is that the virtual commands are generated by the augmented deletion as described in the Delaunay federation node deletion. The lines between 2 and 10 are common between the InsertionResolver and the DeletionResolver and could form a common procedure.

```
DeleteResolver( Federation, InitialResolver, Point )
1  [Commands, TouchedNodes] = DelaunayMesh.delete( Point )
2  for each Command in Commands
3    execute( Command ) // creates or adjust links as appropriate
4  TouchedBindingsList = null
5  for each Resolver in TouchedNodes
6    TouchedBindings = Resolver.getAllBindings
7    TouchedNamesList.add( TochedNames )
8    Resolver.removeAllNames
9  for each Binding in TouchedBindingsList
10   InitialResolver.bind( Binding.Name, Binding.Value )
```

Although the delete operation might not be used as frequently as the insertion of a new resolver it is still very important. Scalability is achieved by injecting a large number of resolvers in the federation. When a machine is taken off line and consequently all contained resolvers are destucted there must be a way of maintaining the integrity of the federation links. This is the main purpose of the delete method.

Chapter 5

Test Cases and Applications

Two implementations of a subset of this federated name resolution service are provided as a proof of concept. Voyager - a standard neutral ORB for mobile computing and Orbix - CORBA 2.0 standard compliant implementation, are used as the underlying communication mechanisms. This service was used in the DIATEM network management project for the federation of the finder objects used for domain connection on agent and event reporting side [31].

5.1 Flat Name Resolution Service

The service IDL definitions, implementation class diagram, user interface, important features like persistency and the CORBA-Voyager interaction are described in the subsequent sections.

5.1.1 CORBA IDL Definition

The typical CORBA name service interfaces are adapted for a flat namespace. The federated naming service maintains a 'database' of bindings between names and object

references. It provides operations to resolve a name, operations to create new bindings, delete existing bindings and to list the bound names.

The difference between this service and the CORBA Name Service is that in the latter a name is always resolved within a given naming context. The naming contexts are organized into a naming graph, which may form a naming hierarchy, much like that of a filing system. This gives rise to the notion of a compound name. The first component of a compound name gives the name of a NamingContext, in which the last name in the compound name is looked up. For example, in the compound name represented by the string: "JIDM_Root.Factory_Root.DomainA.Factory", the object reference for the name "Factory" is looked up in the naming context corresponding to "JIDM_Root.Factory_Root.DomainA".

In the flat name resolution system there are no naming contexts and consequently no hierarchies. All names are contained in a unique, global naming context.

The interfaces that are provided by the federated naming service are defined within the FlatNaming IDL module. A full listing of the definitions within this module is provided below.

```
// CORBA IDL
module FlatNaming {
  typedef string Istring;
  struct Name {
    Istring id;
    Istring kind;
  };
  typedef sequence <Name> NameList;
  interface BindingIterator;
```

```
exception CannotProceed {
  Name name;
};
exception InvalidName {};
exception AlreadyBound {};
exception NotFound {};

interface Resolver {
  void bind(in Name n, in Object obj)
    raises (CannotProceed, InvalidName, AlreadyBound);
  void rebind(in Name n, in Object obj)
    raises (NotFound, CannotProceed, InvalidName);
  Object resolve(in Name n)
    raises (NotFound, CannotProceed, InvalidName);
  void unbind(in Name n)
    raises (NotFound, CannotProceed, InvalidName);
  void list(in unsigned long how_many, out NameList nl, out BindingIterator bi);
};

interface BindingIterator {
  boolean next_one(out Name n);
  boolean next_n(in unsigned long how_many, out NameList nl);
  void destroy();
};
```

```
}; // module FlatNaming
```

Name resolution is the process of looking up a name to obtain an object reference. The *resolve()* operation returns the object reference bound to the specified name in the global namespace. The return type is an IDL Object, which translates to type *org.omg.CORBA.Object* in Java. The result must therefore be narrowed, using the appropriate narrow() function, before it can be properly used by an application.

The *bind()* operation creates a binding between a name and an object. The *bind()* operation raises an exception if the specified name is already bound. The *rebind()* operation creates a binding between a name that is already bound and an object. The previous name is unbound and the new binding is made in its place. A *NotFound* exception will be thrown if the name is not already in use. The operation *unbind()* removes the binding between the specified name and the object it is resolved to.

The operations *bind()* and *rebind()* create bindings, but they can be extended with API calls like *bind_URL()* and *rebind_URL()* in order to create more specialized forms of binding than the general CORBA object reference. The functions *bind()* and *rebind()* allow a name to be bound to any object, while *bind_URL()* and *rebind_URL()* are special constructs used in the context of the naming network supported by the URN service.

The operation *list()* obtains a list of the name bindings in the flat namespace. The parameter *how_many* specifies the maximum number of bindings that should be returned in the *NameList* parameter *nl*. The *NameList* parameter is a sequence of names containing the *id* and *kind* fields. The *kind* attribute is usually set to indicate the type of the object that is referenced by the given name (possibly an URL object). If more than the requested number (*how_many*) of names are to be returned, the *list()* operation constructs a *BindingIterator* with the remaining bindings in the parameter *bi* (the first *how_many* bindings will be in parameter *nl*). If the naming context does not contain

any additional names, the parameter bi will be a nil object reference.

The operations *next_one()* and *next_n()* can be used to access the additional entries (that is, the entries other than those returned by the out *NameList nl* parameter of the *list()* operation). Each entry will be returned at most once: hence consecutive calls to *next_one()* and/or *next_n()* can be used to retrieve all of the additional entries. The operation *next_n()* returns at most n entries: it may return fewer. The operation *next_one()* returns true if an entry can be returned, otherwise it returns false. The operation *next_n()* returns true if n (or fewer) entries can be returned: if no entries can be returned, *next_n()* returns false. A *BindingIterator* object can be deleted by calling its *destroy()* operation.

The exceptions in FlatNaming module are raised under different conditions. *NotFound* indicates that the specified name is not bound. *CannotProceed* indicates that the federated naming service cannot continue with the operation request for some reason. *InvalidName* indicates that the specified name is invalid. A Name that contains an id member that is zero or is an empty string is invalid. *AlreadyBound* indicates that an object is already bound to the specified name. At any time, only one object can be bound to a given name.

5.1.2 Voyager Implementation Class Diagram

The first implementation provided as a proof-of-concept uses Voyager - a standard neutral ORB with facilities for mobile computing. The classes and the relations among them are represented in the UML modeling language in Figure 5.1.

The *Delaunay* class contains the graph model of the federation. It uses the *Triangle*, *Edge* and *Node* classes. The *Delaunay* class has operations that perform locally the node insertion or deletion and the subsequent link updates. These operations generate

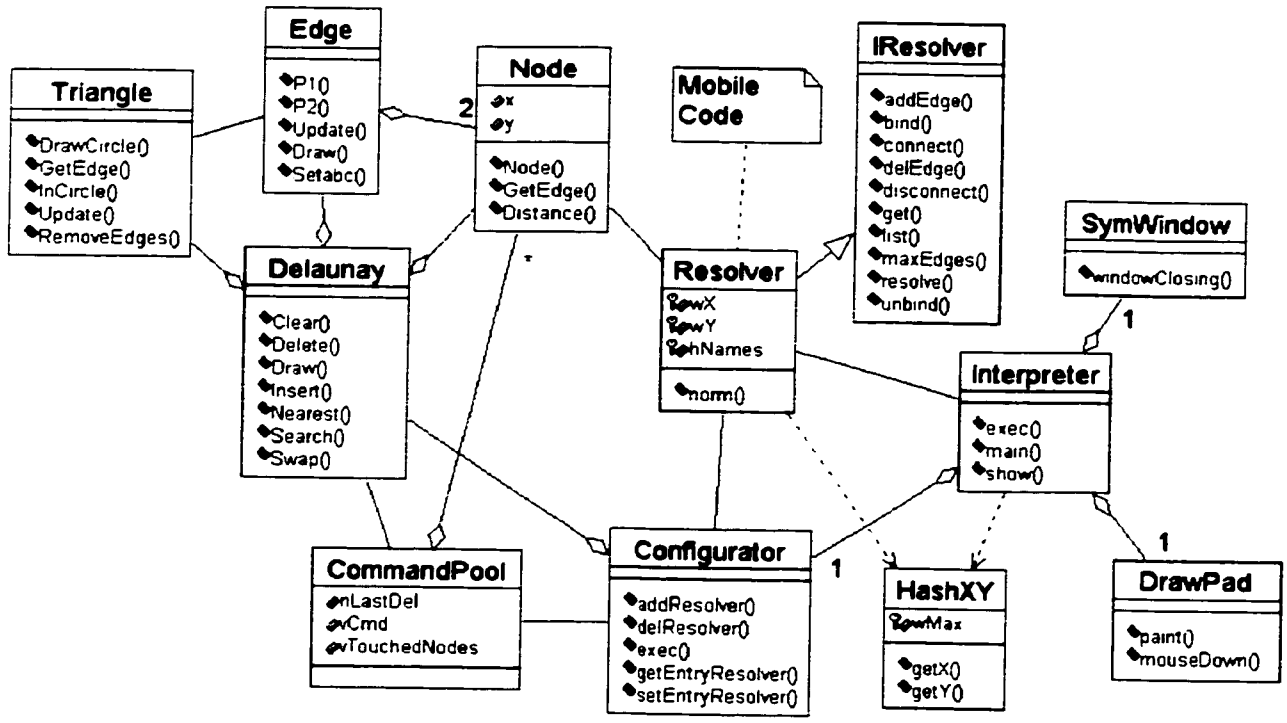

Figure 5.1: Voyager Based Class Diagram

the sequence of commands that are necessary to be executed to update the distributed federation server links. The set of commands consists of add and delete federation nodes, links or name bindings. These commands are stored in the *CommandPool* class and are executed by the *Configurator*. The *Resolvers* are the federation distributed name server nodes. They are mobile classes equipped with facilities for linking themselves in a general graph, for implementing the distributed name resolution algorithm and for managing the local set of names.

The *Resolver* server class implements the *IResolver* interface. The *Resolver* proxy class implements also the *IResolver* interface. It is generated automatically by the Voyager daemon and contains information for packing the parameters and marshalling any client requests to the server.

The hash class embodies the mapping from string names to 2D points. Finally, the user interaction is managed by the *Interpreter*, *DrawPad* and *SymWindow* classes.

This approach that distributes the resolution algorithm and centralizes the federation configuration update process is necessary for the multiple reasons. The number of names and naming operations that the system supports is many times greater than the federation topology changes. System administrators need a centralized view of the federation topology. Also, it is important for the resolution servers to be lightweight for assuring their efficient migration. Adding topology update logic to the *Resolver* classes would increase dramatically their sizes. The complexities of the modified Delaunay algorithms that update the federation topology also suggest centralizing their execution. Finally, changes in the design, more specifically a new graph link organization, other then Delaunay, can be easily obtained by modifying only the Configurators.

5.1.3 Orbix Based Class Diagram

The second implementation is based on Orbix - a CORBA 2.0 standard compliant product. The motivation for providing yet another implementation apart from the Voyager based one is for obtaining a tighter integration within the DIATEM framework that is developed entirely in Orbix.

The classes that are introduced or changed by porting the application from Voyager to Orbix are presented in figure 5.2.

The IDL compiler produces several Java constructs which correspond to the IDL definition. The mapped constructs may be divided into those that allow a client to access an object through the class interface and another set of constructs which allow an object to be implemented in a server.

The *_ResolverImplBase* and *_ResolverFactoryImplBase* classes are abstract Java classes

generated by the IDL compiler for the server-side implementation of the *Resolver* and *ResolverFactory* interfaces. The *_ResolverHelper* and *_ResolverFactoryHelper* classes are also automatically generated by Orbix and are used on the client-side to marshal the requests to the servers. The implementation of the server methods is provided in the *ResolverImpl* and *ResolverFactoryImpl* classes.

At runtime, the *ResolverServer* containing the main method instantiates one *ResolverFactory* object that allows the creation of *Resolvers* at the same server location by invoking the *construct* method.

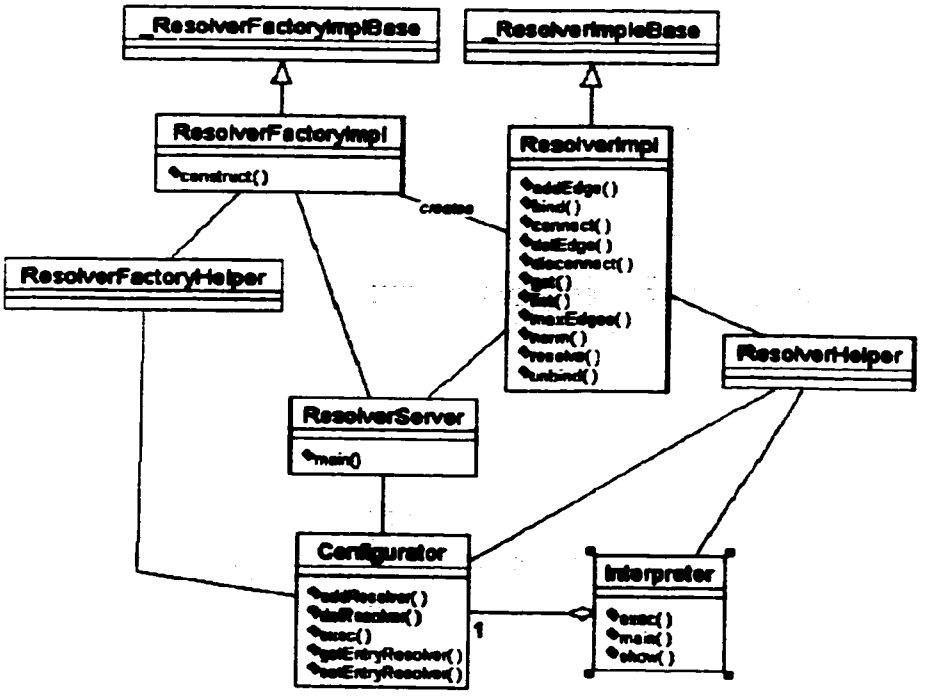

Figure 5.2: Orbix Based Class Diagram

5.1.4 User Interface

A command line interpreter and a graphical federation viewer are provided for the user to interact with the system. A runtime example containing the interpreter window and the associated commands, as well as the graphical window depicting a five-node federation of Resolvers are presented in the figure below.

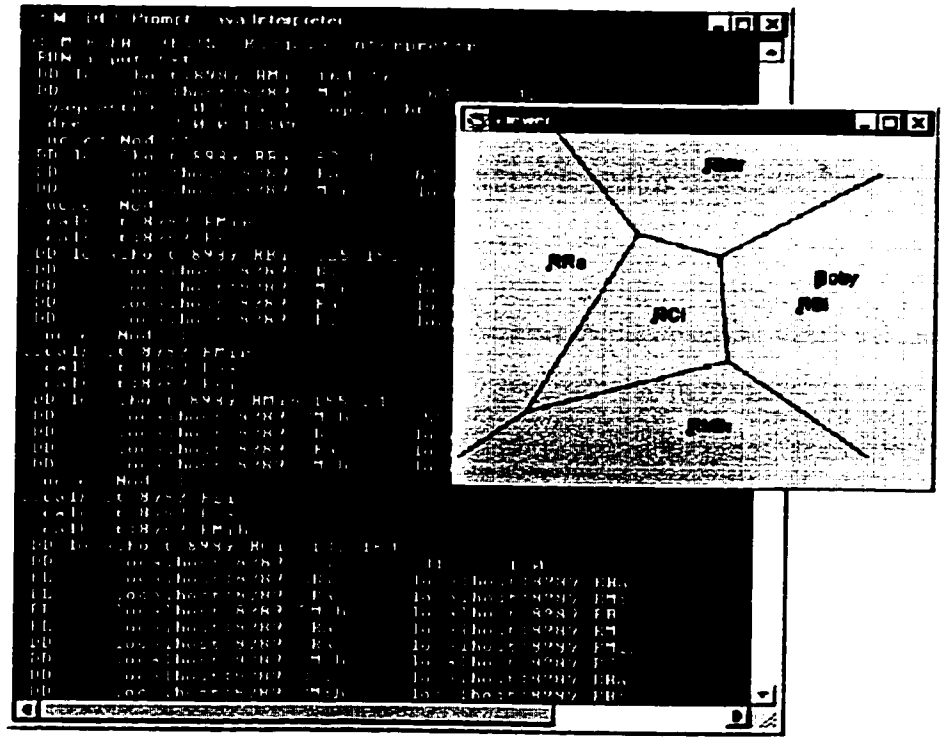

Figure 5.3: The Interpreter and Federation Graphical Viewer Windows

There are commands for adding Resolver nodes at specified network locations. Existing Resolvers can be deleted. Links between Resolvers can also be explicitly created or deleted. The name-related operations are bind, unbind, resolve and list with the known semantics. The load and save commands add persistency to the local set of names stored in the Resolver given as parameter. The view command represents the name's

corresponding point in the plane after the mapping function is applied. By determining in which Voronoi polygon a name is included, the Resolver server that contains it is found. The command syntax is described using the extended BNF notation as follows:

```
<LaunchSyntax> ::= "java" "Interpreter" ["-l" <LogFileName>]
                                        [<BatchFileName>]

<Command> ::= <AddNode> | <AddLink> | <DeleteNode> | <DeleteLink> |
              <Bind> | <Unbind> | <List> | <Resolve> | <Discover> |
               <Run> | <Save> | <Load> | <Comment> | <Help> | <Quit>
<AddNode> ::= "ADD" <HostName>:<Port>/<ResolverAlias> <IntX> <IntY>
<AddLink> ::= "ADD" <ResolverAlias1> <ResolverAlias2>
<DeleteNode> ::= "DEL" <ResolverAlias>
<DeleteLink> ::= "DEL" <ResolverAlias1> <ResolverAlias2>
<Bind> ::= "BIND" <Name> <ObjectReference> ["-a" <ResolverAlias>]
           ["-t" <TimeOut>]
<UnBind> ::= "UNBIND" <Name> ["-a" <ResolverAlias>] ["-t" <TimeOut>]
<Resolve> ::= "RESOLVE" <Name> ["-a" <ResolverAlias>] ["-t" <TimeOut>]
<List> ::= "LIST" [<ResolverAlias>]
<Save> ::= "SAVE" <ResolverAlias> [<FileName>]
<Load> ::= "LOAD" <ResolverAlias> [<FileName>]
<View> ::= "VIEW" [Name]
<Run> ::= "RUN" <BatchFileName>
<Help> ::= "HELP" [<CommandName>]
<Comment> ::= "//" <comment>*
<Quit> ::= "QUIT"
```

A log file for all entered operations and their output is created if the "-l" option is used. In the Voyager daemon activity log window a method invocation trace is printed out. The constructor invocation after the mobile class arrived at this node followed by list and bind calls can be observed in Figure 5.4.

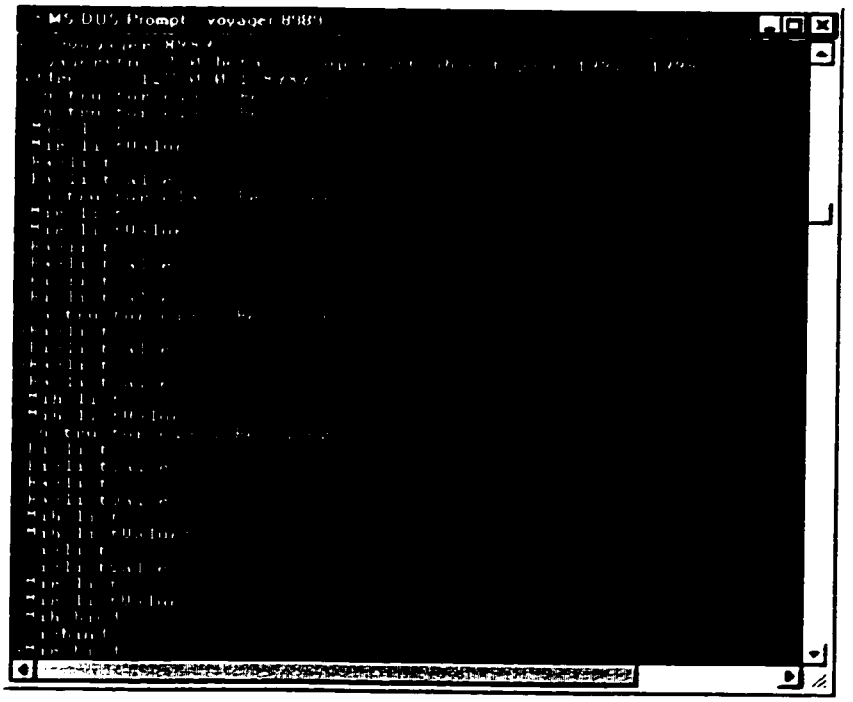

Figure 5.4: The Voyager Demon and Resolver Log Window

A more complex federation containing a larger number of Resolvers is constructed in Figure 5.5. It renders both the Voronoi diagram and the Delaunay network superposed.

The FindNearest procedure from the Delaunay mesh node deletion algorithm is used to retrieve the closest mesh node to be deleted from the user mouse input click. This is necessary for supporting a user-friendly GUI interface when to click exactly on the point that is to be deleted is difficult.

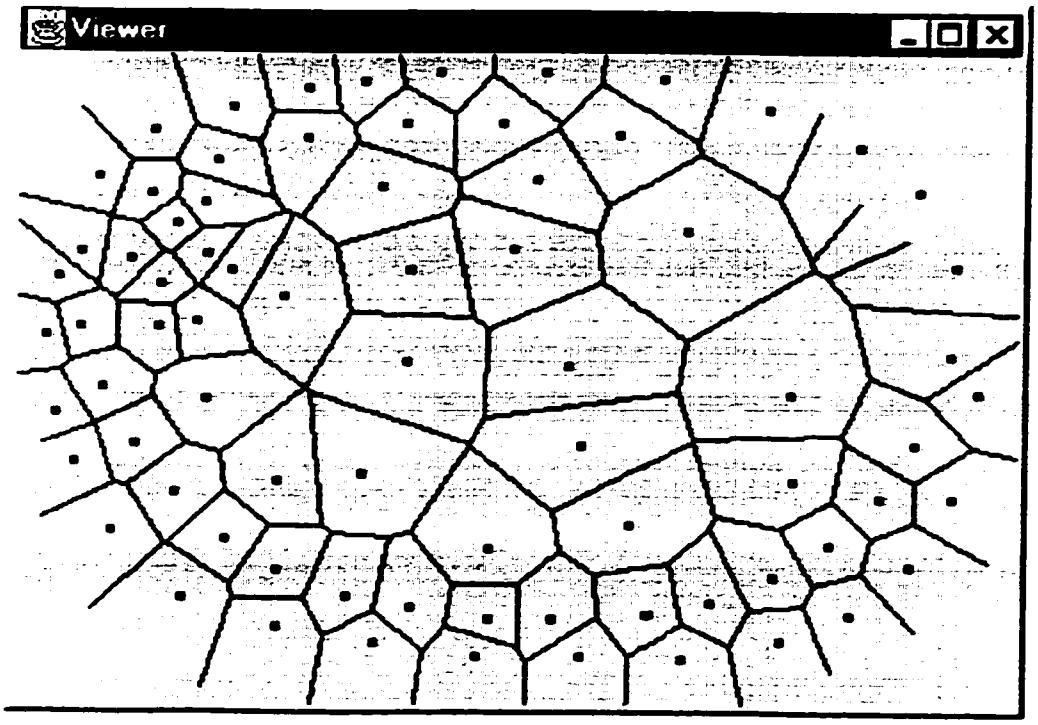

Figure 5.5: Scaled-up Federation

5.1.5 Voyager - CORBA Interaction

The federated Resolvers can serve remote CORBA clients using the ObjectSpace Voyager core technology features. Once a remote CORBA reference is connected to a local Voyager object, Voyager automatically translates communications to and from IIOP. It allows an unmodified Java object to receive IIOP messages from a CORBA client.

For a CORBA client to communicate with a Java object in a Voyager server, the client must have access to its IDL file. Voyager provides a command that generates automatically IDL files from the JAVA definitions. This command creates resolver.idl and configurator.idl according to the Java-to-IDL mapping rules. Once created, these files can be processed by another CORBA vendor's product to produce client-side CORBA stubs. In this way, an instance of a regular Java class like resolver.java is exported as

a CORBA object. To export a Voyager object to a CORBA client, either convert its virtual reference to a stringified IOR using Corba.asIOR() and write it to a shared file or place the virtual reference into a CORBA naming service. A CORBA client can then obtain the IOR and communicate with the resolver object in the Voyager server.

A virtual reference returned from the Voyager server to the CORBA client is automatically converted to an equivalent CORBA reference. Similarly, if a CORBA reference is sent as a parameter from a CORBA client to a Voyager server, the CORBA reference is automatically converted to a Voyager virtual reference.

The VoyagerException and VoyagerRuntimeException exceptions are each thrown as a CorbaSystemException on the CORBA client.

5.1.6 Adding Persistence

The federated name resolution system like most programs requires the ability to create objects, in particular names that have a long life span. Because the program is volatile and will lose all the names from its memory if the host crashes, a database is employed for saving objects on a secondary, nonvolatile medium such as a hard disk.

ObjectSpace Voyager core technology supports database-independent, distributed-object persistence. Any adapter that implements the Voyager database interface can be used for storing and retrieving objects. Voyager includes an object storage system, VoyagerDb, that implements the database adapter and uses the Java serialization mechanism to persist any serializable object without modification. Implementations of the database adapter can be created to support most of the popular database systems. The federated resolution system uses the native VoyagerDb for object storage.

When any Resolver server is started, it is assigned to an optional database using the setDb() API call. Two servers should not share the same database even if they reside on

the same machine. Once a Resolver server is assigned to a database, the server should not be reassigned to a different database.

The initial version of the system used a memory resident hash-table for storing the name database. In order to add persistence to this structure, a special Voyager component class is used. A Voyager component extends any object with special interfaces (IIdentity, IMobility, ILifecycle, and IProperty) that add value to the object.

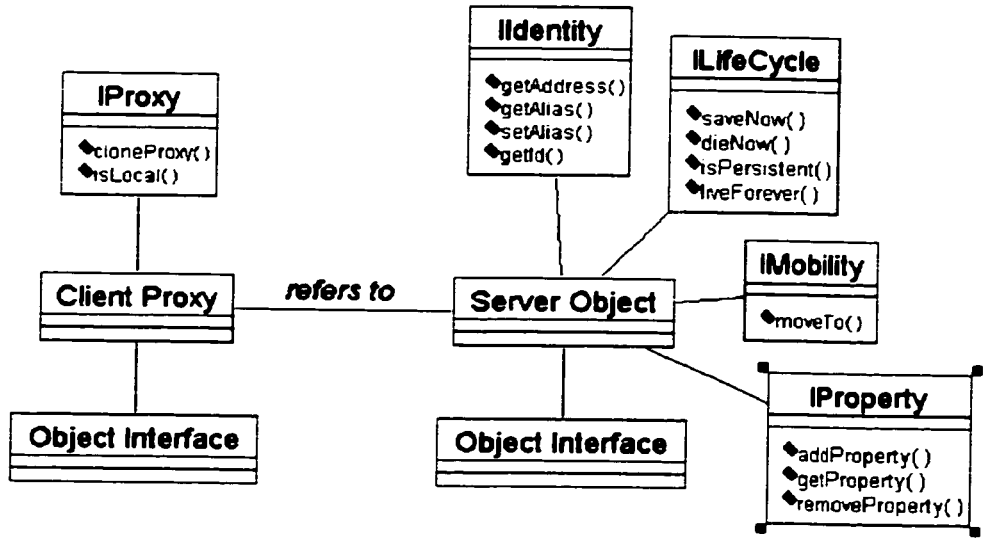

Figure 5.6: Value Added Interfaces

The value-added interfaces allows the Resolver class to be remote-enabled, to become mobile and persistent. There is no need to modify the class in any way, nor to access its class source code.

Persistence is added using the ILifeCycle interface. To save the Resolver in its database, the saveNow() method is invoked via a virtual reference.

```
// save copy to database of server 8000
Lifecycle.of( Resolver ).saveNow();
```

The Resolver is copied into its program's database, overwriting the previous version, if present. If not already persistent, the Resolver is made persistent before it is saved. If the program is shut down and then restarted, the persistent Resolver is initially left in the database and does not consume any memory. The save() and flush() families of methods allow objects to be transparently saved and flushed from memory to persistent storage.

Any attempt to communicate with a persistent Resolver not currently in memory causes the Resolver to be automatically loaded from its database. This feature is called autoloading. Persistent Resolvers can also be autoloaded at program startup by using the setAutoload() method.

5.2 DIATEMS Network Management Application

A general architecture for a CORBA-based Telecommunications Network Management environment has been designed and is under development at the University of Ottawa, in the Machine Intelligence Research Laboratory [31]. A prototype implementing the concepts of actively managing and supervising elements of communications networks is available. There are three basic entities that were considered: managers controlling managed objects representing the network resources partitioned into flat domains defined as sets of resources sharing common characteristics and supporting common rules. Thus, a manager can connect to a domain using its unique title and protocol type: CORBA, CMIP or SNMP, create new managed objects, obtain existing managed objects references, invoke actions on individual or sets of resources satisfying eventually certain criteria. Also, the managers can receive reports on events happening in the supervised domains and change the destination of event reporting.

The main advantage of using CORBA for Network Management stems from the possibility of inter-working with SNMP and CMIP domains through gateways, which dynamically translate invocations and notifications from one domain to another. This translation is described in [32] as bi-directional in the case of CORBA-CMIP inter-working, and as unidirectional in the case of the inter-working with SNMP. The process of translating the static, i.e. syntactic constructs of two different domains is referred to as "specification translation" whereas the one of translating the dynamic constructs (invocations and notifications) is known as "interaction translation". For these translations, a formal approach is defined. Other features included in this architecture are the possibility to add new managed object classes, and to add new methods and attributes to managed objects without shutting down the network management application.

The name resolution system is used in the domain namespace i.e. the federation of ProxyAgentFinders and in the event reporting i.e. the federation of the EventPortFinders.

The name of an initial resolver node from the federation is introduced in the host of the federation repository field as it can be seen in Figure 5.7. Also, the manager title and the host of the interface repository are entered before any other operation is invoked.

In Figure 5.8, the name for the new domain to be created, MIRLAB in this case, is entered. Also, the hostname or IP address where the new domain is to be located and the manager names that will have access rights are introduces in a popup window. A bind operation between the name and the domain in executed.

When a manager wants to connect to a specific domain he enters the domain name and the type of environment: CORBA, SNMP or CMIP and a resolve operation is invoked as shown in Figure 5.9. The result is presented in Figure 5.10. The domain

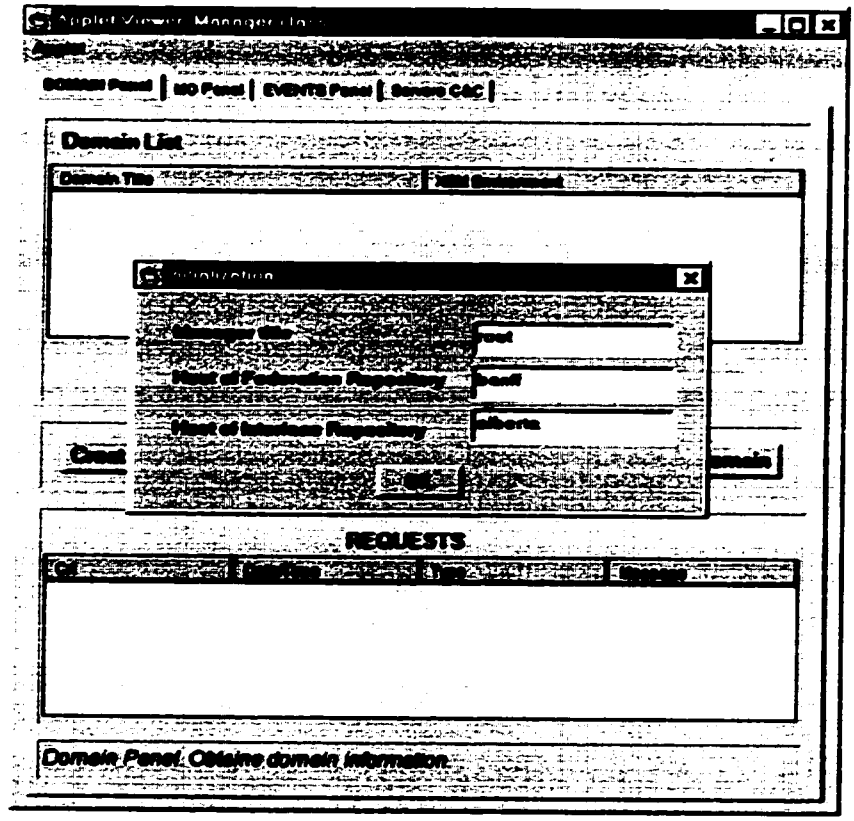

Figure 5.7: Connecting to the Initial Resolver

is added to the domain list and from now on the manager can switch to the MO panel, browse through the set of managed objects contained in this domain or invoke actions on them.

The salient features of the system are its openness and scalability. Due to its multi-tier architecture the system can be extended by writing new sets of managed objects dynamically discovered at run time, without the need of recompiling any part of the application. In the case of a hierarchical naming service, if the names are considered together with their contexts the flat name resolution system can be reused as well.

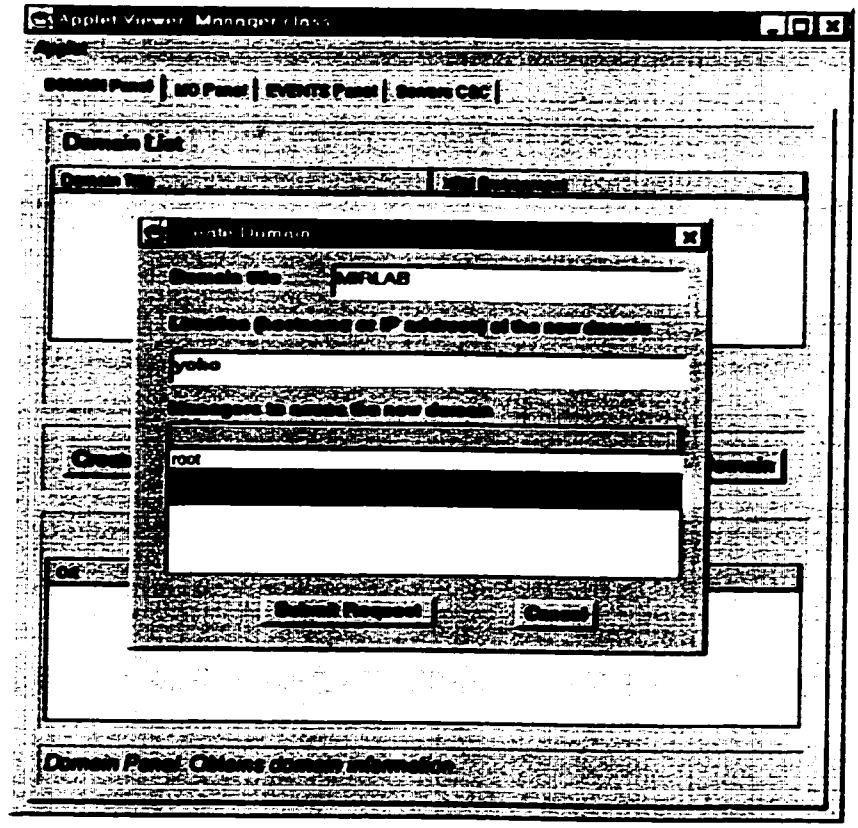

Figure 5.8: Binding the Domain Name to Attributes

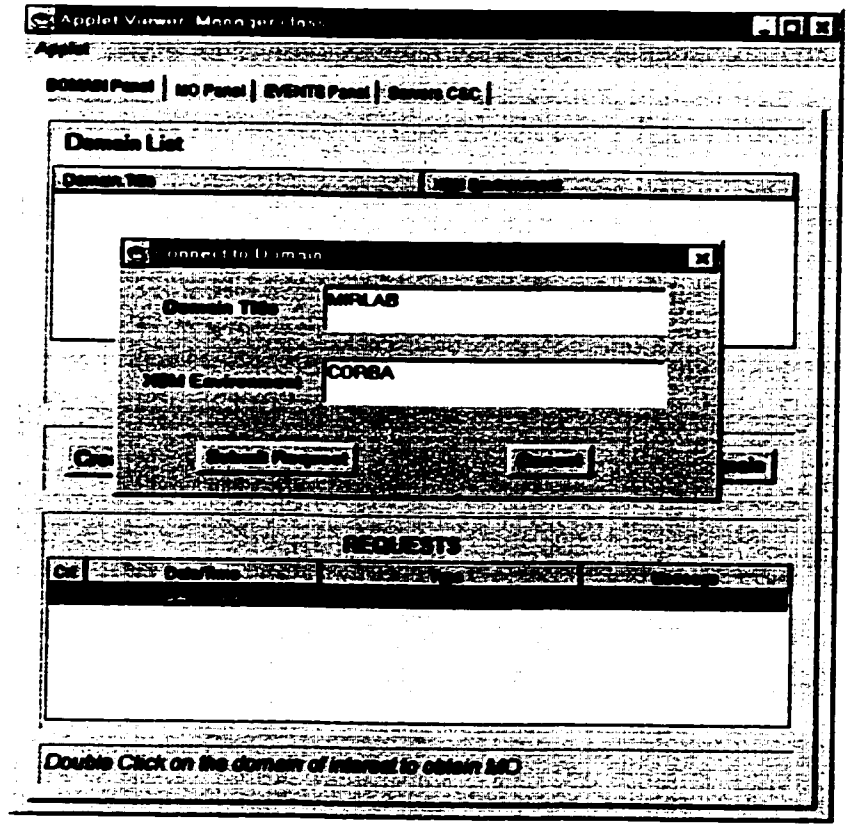

Figure 5.9: Resolving a Domain Name

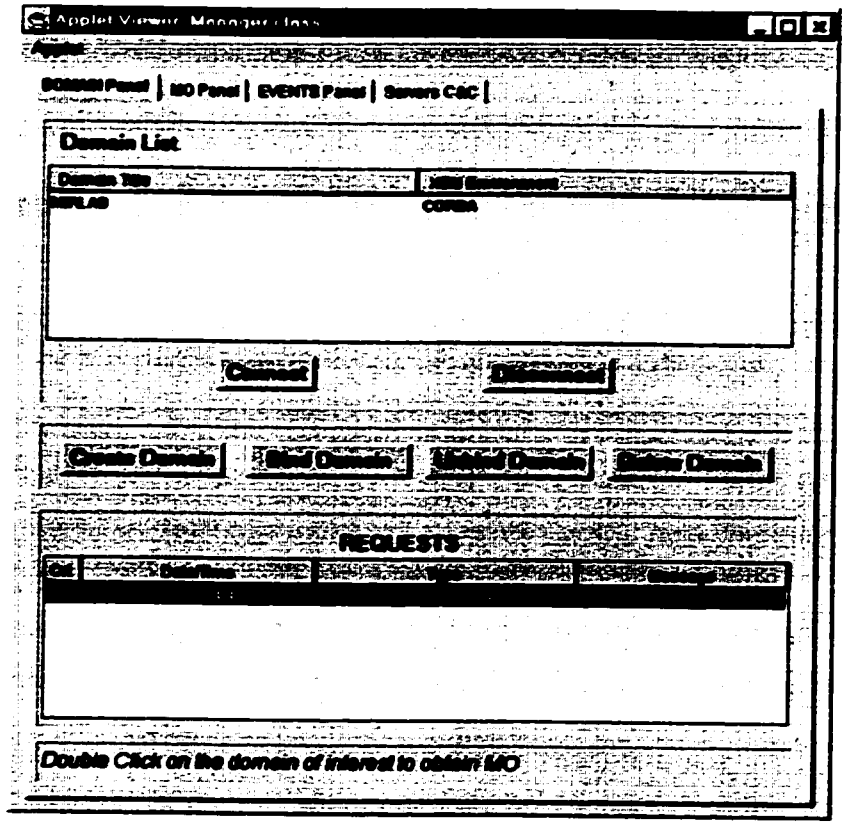

Figure 5.10: Result of the Name Resolution

Chapter 6

Performance Analysis

The locality of the annotated Delaunay algorithm is examined in terms of the number of touched nodes and number of generated instructions to be executed by the Configurator. This is done in the context of a scalability experiment that constructs a federation containing a large number of Resolver nodes.

The next two figures illustrate the locality property stating that the number of the federation links that have to be udated when a new node is inserted does not increase with the number of the federation nodes. The complexity of the insertion operation is independent of the size of the federation.

The Configurator server generates the commands necessary to update the federation links when new Resolver servers are added to the federation. In order to accomplish this functionality, the Configurator uses the Delaunay graph model of the whole federation. It performs locally the insertion in the Delaunay model and generates the sequence of commands that are necessary to be executed to update the federation links. All the nodes that are influenced by the addition are included in the *touched nodes* set. The set of commands generated by the Configurator consists of adding the new node, adding and deleting the links and the corresponding name bindings affected by the insertion. The

commands stored in the command pool are counted. The command execution triggers the invocation of the corresponding remote Resolver methods for link updates.

Initially, the federation contains one Resolver, Res_1. Another Resolver, Res_2 is added and consequently, the annotated Delaunay node insertion algorithm generates two commands: one for adding the new Resolver, Res_2, and one for linking the two Resolvers. The touched nodes set will contain Res_1 and its cardinality is equal to one. The insertion process is repeated for the federation containing N nodes and both the number of touched nodes and the number of generated commands are recorded.

In Figure 6.1 the number of touched nodes influenced by the insertion of a new node is presented as a function of the number of federation nodes, N. This is an important measure since all the local names in the touched nodes set have to be reinjected in the federation. This process is described in the Resolver insertion algorithm. In Figure 6.2 the number of generated commands is depicted as a function of the number of federation nodes.

As it can be observed in this example, there are no more than maximum 9 touched nodes and 35 generated commands for any number of nodes in the federation such that the insertion in the Delaunay federation is always executed in constant time.

The previous results were obtain by constructing a large federation comprising 128 mobile Resolver server nodes. It is depicted in Figure 6.3 as it appears in the graphical federation viewer window.

Starting with an empty federation, the user clicks the left mouse button in the viewer window to insert new Resolvers at the desired locations. The network address where the Resolvers are to be deployed, as well as their corresponding mapped Euclidean point coordinates are entered in the Interpreter window. At any time the user can create new name bindings or invoke name resolution operations.

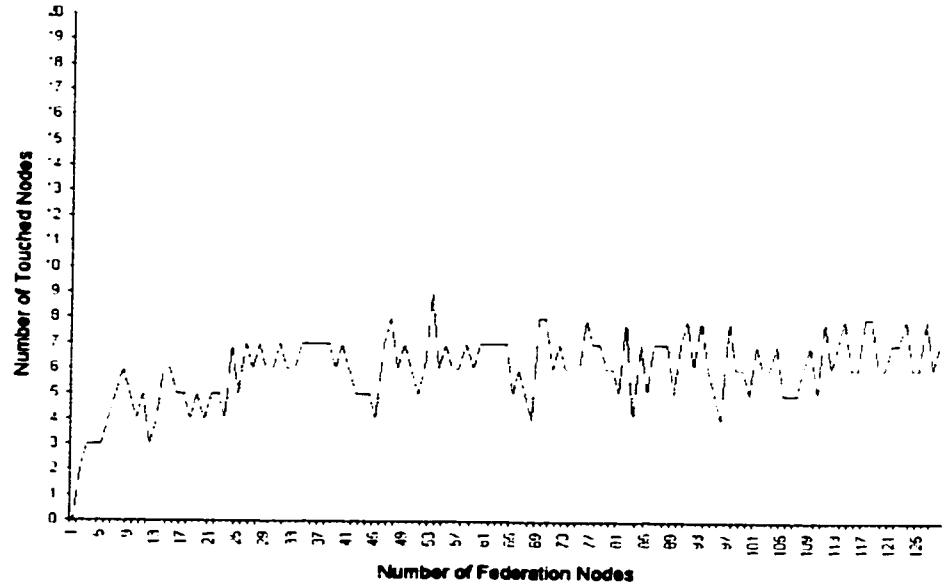

Figure 6.1: The Number of Touched Nodes Function of the Federation Nodes Number

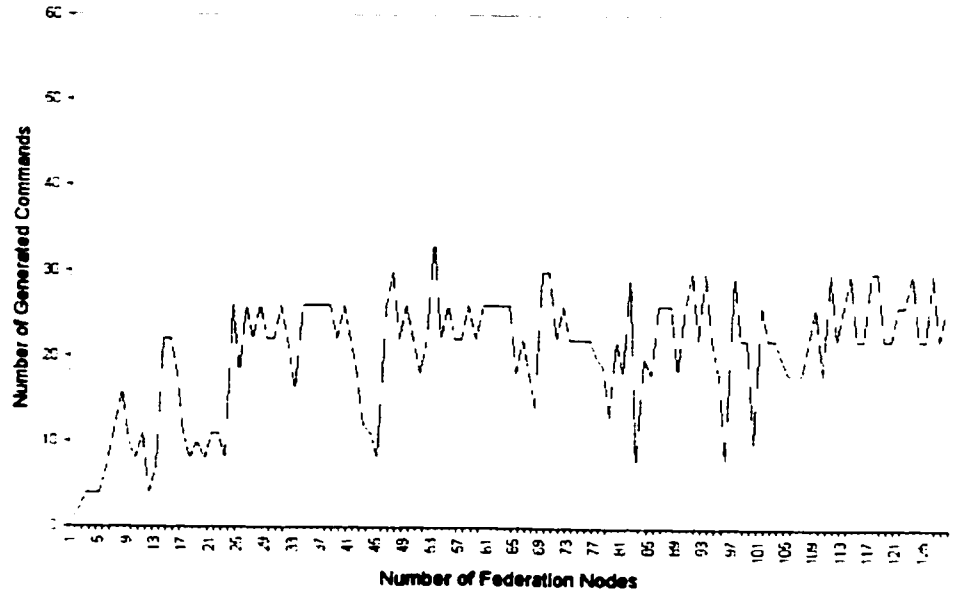

Figure 6.2: The Number of Generated Commands Function of the Federation Nodes

In the left part of the viewer window the refinement Property is illustrated by increasing the Resolver's density for example where the namespace is crowded.

The MIG laboratory network where this experiments were performed is depicted in Figure ??. Both underlying implementations, based on Voyager and Orbix, are available for the Solaris and Windows platforms.

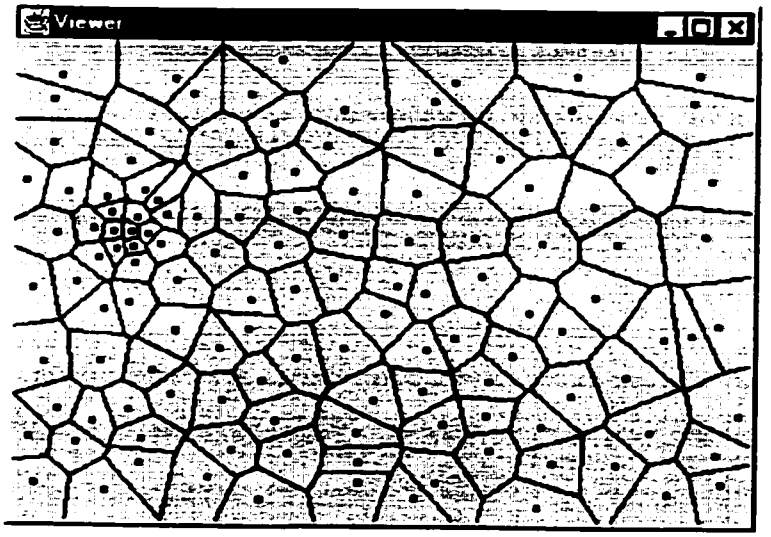

Figure 6.3: The 128 Node Federation

In Figure 6.4 the processing time for the two paradigms is presented: mobile and classical client/server. In this test a federation containing only one Resolver server was used. The get method of the Resolver object is tested both remotely, being invoked by a client from a different location and locally called by a mobile agent that moves to the Resolver location. The get operation returns the name bindings that are stored locally in the Resolver. It is used in the name reinjection process that occurs when a new Resolver is inserted in the federation, as described 3.5.2.

First up to 1000 remote method invocations from the client that resides on a remote virtual machine (VM) are performed, timed and the elapsed time is printed out. The

Voyager's time services are being used. After the remote invocations are completed, a mobile agent is constructed and instantiated at the remote location, it moves to the Resolver server's virtual machine and the same sequence of get operations, now local invocations, is called.

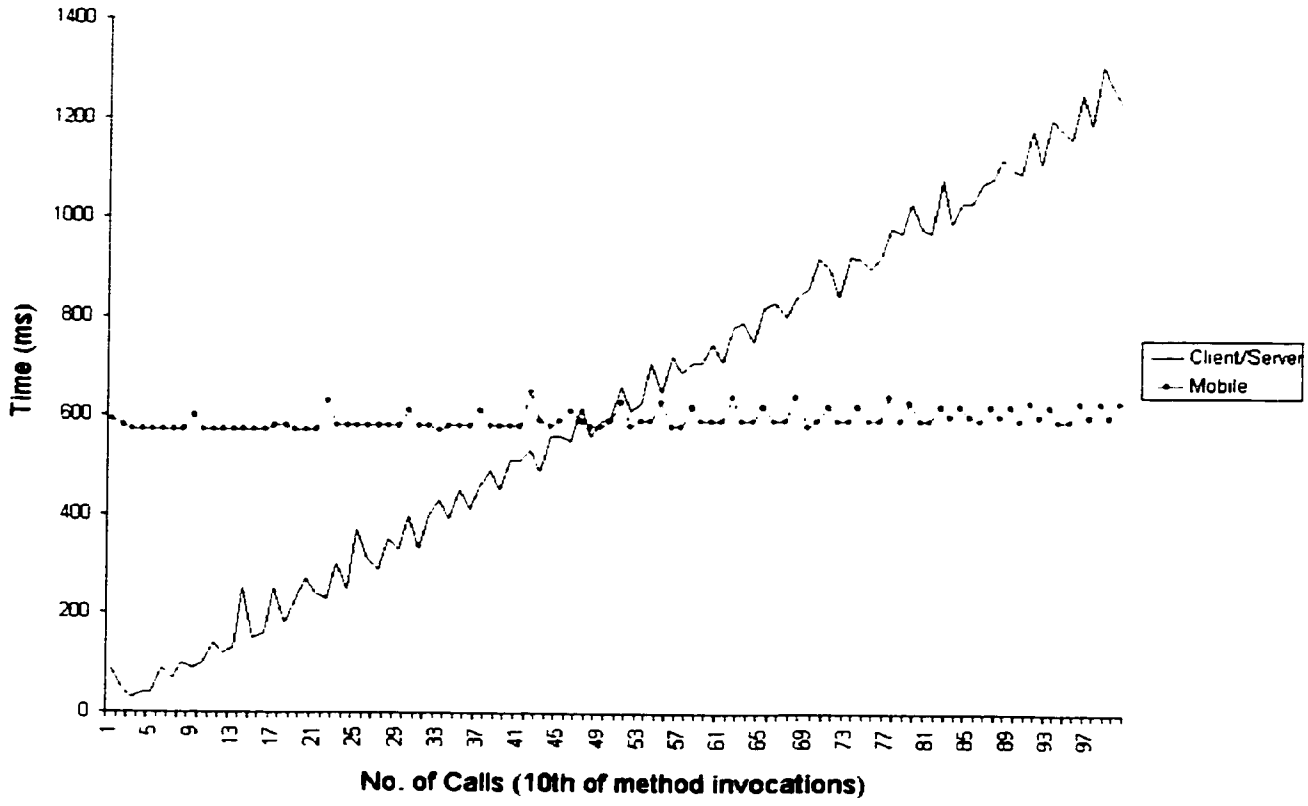

Figure 6.4: Processing Time in Mobile and Client/Server Paradigms

The processing times printed after both rounds of invocations illustrate the difference between performing actions on a remote object and moving to the same location as the object and performing the actions there. The difference in slopes and times is obvious. Although both VMs are in the same physical machine, the processing time after the agent moves to the remote location is noticeably shorter. In the mobile paradigm there is a relatively constant penalty of approximately 571ms due to the mobile agent class

migration. This value is function of the class length measured in bytecodes. It can be observed that the trend slope in the mobile case is almost horizontal while in the client/server case it has an observable greater value. For values greater than 500 method invocations it is more efficient to use the mobile agent approach. This is the case during the name reinjection process when the number of invocations for the method get is equal to the number of the locals names stored in the Resolver. An alternative solution is implementing a bulk mode insertion method.

Chapter 7

Conclusion and Future Directions

While the computer industry changes rapidly, established components of the network, such as the DNS, have remained stable for a relatively long time. As a basic network service, this federated naming service is also meant to be viable for many years and to provide a scalable path both for future needs and for transition to a different system if the need arises.

The federated name resolution system described here is sufficient for providing a flat naming services for the DIATEMs network management application and can serve also as a solution for the future implementations of URN namespaces. Its structure and resolution mechanism promotes longevity through a single, distributed, global registry. No central authority has absolute control over all the name bindings.

The dual structures Delaunay-Voronoi for the organization of the federation links and names association to nodes offers a balanced, decentralized solution that is believed to be superior with respect to other existing implementations involving B-trees or other hierarchical implementations. Apart from avoiding the root node bottleneck, an improvement over the alternative B-Tree implementation is that a Resolver can hold an arbitrary number of name bindings. In the alternative architecture the minimum

amount of data a member must hold is fixed [56]. However, what this amount is cannot be determined without experience and may even change over time. Instead of splitting a server's capacity into fixed chunks, the Resolvers are able to store a variable size namespace partition. If the physical space is consumed entirely, new resolvers on new machines can rebalance the federation.

The contributions of this thesis are as follows:

- The design of a flat name resolution server federation based on the Delaunay mesh and a corresponding node location algorithm.
- The use of mobile agents for the federation server's deployment.

There are multiple future directions to improve this federated name resolution system. Errors and exception conditions need to be accurately reported so that the initiating Resolver or Configurator can detect and deal with them. They are easy to be detected and back out of for the single name binding insert but more difficult to not just report, but also isolate in a bulk insert because of the associated transaction mechanism issues.

More tests need to be performed since the federated database has been deployed first on a small scale with all Resolvers residing on a single machine, then in a LAN environment but not on a large network comprising hundreds of computers.

More refinements are necessary in the context of multi-client use. A transactional mechanism that freezes all pending name resolutions during federation link updates is necessary. To speed-up the resolution, multiple firings from different seed nodes or special labeled "speed" links can be used.

The federated name resolution system does not have any provisions for access control and security, features that are delegated to other software layers in the DIATEMS

framework. Security is an open issue and its role in the federation needs further study.

Finally, a deployable database will need more refined administrative tools than the rudimentary ones described in the last chapter.

Related to the possible use of the system as a solution for the URN problem there is still work to be done. Only little amount of Information is known on the number of URNs, users, and resolution requests that would result from a successful URN system is known. Existing data comes from the DNS and the Web, although neither can be measured well due to their distributed nature. Work on URNs is ongoing in the Internet Engineering Task Force URN Working Group. While this thesis proposes herein a method of resolving URNs intended to be scalable and long-lasting, there are other ideas about what URNs are and how they should be managed. Only time will tell if the community adopts URNs at large, especially by companies that write client/server software. Meanwhile, if a flat name resolution system is needed, especially in the LAN environment, that does not imply any hierarchy for achieving scalability in the namespace, the Delaunay federation can definitively serve this purpose.

The geometrical nature of the Delaunay structure would also be a perfect choice for the topology selection for multihop packet radio networks. The possibility of using the Delaunay network to organize the CORBA trading service links has to be examined also.

In summary, the future extensions include:

- Refinements of the implementations in terms of: exceptional condition reporting, the locking mechanism necessary for multi-client access, access control and security.

- The alternative implementation of federation discovery and link update using mobile code instead of distributed method invocations.

- The use of the Voronoi diagram in the context of wireless networks.

Bibliography

[1] S. Albayrak, F. J. Garjo *Intelligent Agents for Telecommunication Application. IATA '98.* Lecture Notes in Artificial Intelligence.

[2] S. Baker. *CORBA Distributed Objects, using Orbix.* ACM Press, Addison-Wesley, 1997

[3] M. Baldi, S. Gai, G. P. Picco. *Exploiting code mobility in decentralized and flexible network management.* Proceedings of the First International Workshop on Mobile Agents, Berlin, Germany, April 1997.

[4] J. Baumann, C. Tschudin, J. Vitek, editors. *Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems.* Linz, Austria, July 1996.

[5] J. Baumann, F. Hohl, N. Radouniklis, K. Rothermel, and M. Straser. *Communication concepts for mobile agent systems.* Proceedings of the First International Workshop on Mobile Agents, Berlin, Germany, April 1997.

[6] R. Ben-Natan. *CORBA on the Web.* McGraw-Hill. 1998.

[7] I. Z. Ben-Shaul, G. E. Kaiser. *Federating process-centered environments: the Oz experience.* Automated Software Engineering. v 5 n 1 Jan 1998. p 97-132. URL: http://www.cs.columbia.edu/library/1997.html.

[8] T. Berners-Lee, L. Masinter, M. McCahill. *Uniform Resource Locators (URL).* RFC 1738, December 1994. URL: ftp://ds.internic.net/rfc/rfc1738.txt.

[9] T. Berners-Lee, et al. *The World-Wide Web.* Communications of the ACM, 37 (8):76-82, August, 1994.

[10] K. Bharat and L. Cardelli. *Migratory Applications.* Springer-Verlag, p 131-149. 1997.

[11] A.D. Birrell, et.al. *Grapevine: An exercise in distributed computing.* Comm. ACM 25, 4 (Apr 1982), 260-274.

[12] A. D. Birrel et. al. *A global authentication service without global trust.* Proceedings of the IEEE Symposium on Security and Privacy, 1986, p223-230.

[13] U. Black. *Network Management Standards: SNMP, CMIP, TMN, MIBs, and Object Libraries, 2nd Edition.* McGraw-Hill, 1995.

[14] H. Blum. *A transformation for extracting new descriptors of shape.* In Proceedings of the Symposium on Models for the Perception of Speech and Visual Form, pages 362-380, 1967.

[15] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. *Harvest: A scalable, customizable discovery and access system.* Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado, Boulder, August 1994. URL: ftp://ftp.cs.colorado.edu/pub/cs/techreports/schwartz/Harvest.Jour.ps.Z.

[16] J. Bradshaw (ed.). *Software Agents.* Menlo Park, California: AAAI Press / The MIT Press, 1996.

[17] M. Campione, K. Walrath. *The Java Tutorial. Second Edition Object-Oriented Programming for the Internet.* Addison-Wesley Longman, 1998.

[18] A. Carzaniga, G. P. Picco, and G. Vigna. *Designing distributed applications with a mobile code paradigm.* Proceedings of the 19th International Conference on Software Engineering, Boston, Ma., May 1997.

[19] A. Castillo, M. Kawaguchi, N. Paciorek, D. Wong *ConcordiaTM as Enabling Technology for Cooperative Information Gathering.* Japanese Society for Artificial Intelligence Conference" in Tokyo, Japan on June 17-18, 1998.

[20] S.J. Caughey and S.K. Shrivastava *Architectural Support for Mobile Objects in Large Scale Distributed Systems.* Proceedings, 4th Int. Workshop on Object Orientation in Operating Systems August 1995.

[21] B. Delaunay. *Sur la sphere vide.* Bulletin of Academy of Sciences of the USSR, pages 793-800, 1934.

[22] G. L. Dirichlet. *Uber die reduction der positiven quadratischen formen mit drei unbestimmten ganzen zalen.* J. Reine u. Angew. Math., 40:209-227, 1850.

[23] D. Howe. *Free On-Line Dictionary of Computing.*

[24] M. D. Gallager, R. A. Snyder. *Mobile Telecommunication Networking with IS-41.* McGraw-Hill, 1997

[25] E. Gamma, R. Helm. *Design Patterns.* Addison-Wesley Longman, 1995.

[26] C. Ghezzi and G. Vigna. *Mobile code paradigms and technologies: A case study.* Proceedings of the First International Workshop on Mobile Agents, Berlin, Germany, April 1997.

[27] J. Gosling, B. Joy and G. Steele. *The Java Language Specification.* Addison-Wesley Longman, August 1996. URL: http://java.sun.com/docs/books/jls/html/index.html.

[28] D. Harkey and R. Orfali. *Client/Server Programming With Java and CORBA.* John Wiley and Sons, 1997.

[29] D. Heimbigner and D. McLeod. *A Federated Architecture for Information Management.* ACM Transactions on Office, Information Systems 3, 3 (July 1985), pp. 253-278.

[30] R. Hoque, T. Sharma. *Programming Web Components.* McGraw-Hill, 1998.

[31] D. Ionescu, et al. *A Distributed Network Management Environment* Proceedings of CONTI'98, The International Conference on Technical Informatics, Timisoara, Romania, 1998.

[32] OMG. *Interaction Translation.* Final Submission to OMGs CORBA/TMN Interworking RFP, Edition : 4.2, OMG Document Number: telecom/98-08-14

[33] R. Kramer. *Databases on the Web: Technologies for Federation Architectures and Case Studies.* SIGMOD Record - Quarterly Publication of the Special Interest Group on Management Data, 1997, v.26, n.2, p.503, 4p.

[34] B. Lampson. *Designing a Global Name Service.* Proc. 4th ACM Symposium on Principals of Distributed Computing, Minaki, Ontario, 1986.

[35] D. Lange, M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets.* Addison-Wesley-Longman, 1998.

[36] R. C. Malveau and T. J. Mowbray, *CORBA Design Patterns*. John Wiley and Sons, 1997.

[37] T. Midtbo. *Spatial Modelling by Delaunay Networks of Two and Three Dimensions. Dr.Ing. thesis.* Norwegian Institute of Technology, University of Trondheim. February 1993.

[38] P. Mockapetris. *Domain Names - Concepts And Facilities.* Network Working Group RFC 1034, November 1987. URL: ftp://ds.internic.net/rfc/1034.txt.

[39] T. J. Mowbray, R. Zahavi. *The Essential CORBA - Systems Integration Using Distributed Objects.* John Wiley and Sons, 1995.

[40] E. P. Mucke, I. Saias, B. Zhu *Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations* Proceedings of the Annual Symposium on Computational Geometry 1996. ACM, New York, NY, USA., p 274-283.

[41] ObjectSpace. *Voyager and Agent Platform Comparison* ObjectSpace Press, 1998. URL: www.objectspace.com/voyager.

[42] ObjectSpace. *Voyager examples, Version 2.0 Beta 2.* ObjectSpace Press, 1998. URL: www.objectspace.com/voyager.

[43] ObjectSpace. *Quick Start Guide, Version 2.0 Beta 2.* ObjectSpace Press, 1998. URL: www.objectspace.com/voyager.

[44] ObjectSpace. *Core Technology User Guide, Version 2.0 Beta 1.* ObjectSpace Press, 1998. URL: www.objectspace.com/voyager.

[45] R. Orfali. *Instant CORBA.* John Wiley and Sons, 1997.

[46] R. Otte et al. *Understanding CORBA*. Prentice-Hall, 1996.

[47] E. Pitoura, G. Samaras. *Data Management for Mobile Computing*. Kluwer, 1998.

[48] A. Pope. *The CORBA Reference Guide : Understanding the Common Object Request Broker Architecture*. Addison-Wesley, 1997.

[49] F. Preparata and M. Shamos. *Computational Geometry*. Springer-Verlag, 1988.

[50] M. Ranganathan, A. Acharya, S. Sharma and J. Saltz. *Network-aware mobile programs*. Proceedings of the USENIX 1997 Annual Technical Conference, Anaheim, Cal., January 1997.

[51] K. Rothermel, R. Popescu-Zeletin. *Mobile Agents, First International Workshop, MA '97*. Berlin, Aplil 1997, Lecture Notes in Computer Science.

[52] W. A. Ruh and T. J. Mowbray. *Inside CORBA: Distributed Object Standards and Applications*. Addison-Wesley, 1997.

[53] M.D Schroeder et. al. *Experience with Grapevine*. ACM Trans. Computer Sys. 2, 1 (Feb. 1984), 3-23.

[54] J. Siegel et. al. *CORBA Fundamentals and Programming*. John Wiley and Sons, 1996.

[55] M. Singhal, N. Shivarartri. *Advanced Concepts in Operating Systems: Distributed, Database and Multiprocessor Operating Systems*. McGraw-Hill, 1994.

[56] E. C. Slottow. *Engineering a Global Resolution Service*. M.Eng Thesis, MIT, June 1997.

[57] R. Soley (Ed.). *The Common Object Request Broker: Architechture and Specification.* Object Management Group, Inc., Revision 2.0, July 1995.

[58] K. R. Sollins. *Supporting Longevity in an Information Infrastructure Architecture.* Proceedings of the 1996 SIGOPS European Workshop, Connemara, Ireland. September 1996.

[59] K. R. Sollins. *Functional Requirements for Uniform Resource Names.* Network Working Group RFC 1737, February 1995.

[60] K. R. Sollins. *Requirements and a Framework for URN Resolution Systems.* Internet Draft, March 1997.

[61] K. R. Sollins and J. R. Van Dyke. *Linking in a Global Information Architecture.* World Wide Web Journal, 1(1), November 1995, pp 493-508.

[62] K. R. Sollins. *Architectural Principles of Uniform Resource Name Resolution.* Network Working Group RFC 2276, 1998.

[63] F. Song. *Neighbouring graphs as alternative organizations for information retrieval.* Proceedings of the ACM International Conference on Digital Libraries 1997. ACM, New York, NY, USA, p 264.

[64] A. T. Sundsted. *An Introduction to Agents* JavaWorld, June 1998.

[65] A. T. Sundsted. *Agents on the Move.* JavaWorld, July 1998.

[66] S. G. Tatu. *An Agent-based Link Maintenance System.* M.Sc. Thesis, Simon Fraser, December 1997.

[67] D. E. Taylor. *Multiplatform Network Management.* McGraw-Hill, 1997.

[68] A.J. Thiessen and J. C. Alter. *Precipitation averages for large areas.* Monthly Weather Review, 39:1082-1084, 1911.

[69] C. Thompson. *Composition and Federation Patterns in Componentware Software Architectures.* OBJS, Technical Report, URL: http://www.objs.com/staging/federation.html.

[70] J. Vitek and C. Tschudin, editors. *Mobile Oject Systems: Towards the Programmable Internet.* Springer-Verlag, April 1997. Lecture Notes in Computer Science No. 1222.

[71] A. Vogel and K. Duddy. *Java Programming With CORBA.* John Wiley and Sons, 1997.

[72] A. Vogel, H. Wittig. *Discovering Mobile Code.* Position paper, Joint W3C/OMG Workshop on Distributed Objects and Mobile Code June 24-25, 1996, Boston, Massachusetts.

[73] M. G. Voronoi. *Nouvelles applications des parametres continus a la theorie des formes quadratiques.* J. Reine u. Angew. Math., 134:198-287, 1908.

[74] C. Weider and P. Deutsch. *A Vision of an Integrated Information Service.* RFC1727, December 1994. URL: ftp://ds.internic.net/rfc/rfc1727.txt.

[75] J. E. White. *Telescript technology: Mobile agents.* Software Agents, Menlo Park, California: AAAI Press / The MIT Press, 1996.

[76] E. Wigner and F. Seitz. . *On the construction of metallic sodium.* Phys. Review, 43:804-810, 1933.

[77] Xerox Corporation. *Clearinghouse Protocol.* XSIS 078404. Stamford, Conn., 1984.