Distributed Name Server Consistency

Supervised by:
Professor M. R. Lyu

FINAL REPORT

Tang Cheung Yin
S97082230
CSC 7260
10 August, 1999

Abstract

The increasing availability of cheaper, more powerful workstations and new networking capabilities has seen a corresponding increase in the use of distributed systems in many areas. Many organizations are by their very nature distributed, with individuals working in different locations, but requiring the ability to exchange information easily. A basic component in distributed systems is the Name Service. Clients can query the name server to find out the machine/port pair for the service in which they are interested. I have reviewed basic name service and its sophisticated "colleague", trading service. With multiple instances of the name service in different nodes, we can increase the availability and performance of the service. However, this will introduce problems in maintaining the consistency of data within the distributed system. We have designed a name server based on an existing implementation for OmniOrb2 CORBA implementation. We have extended the name server with group communication support, so that the extended name server can run an instance on every node of the network concurrently; accept name service query from any nodes; replicate naming data to each group member consistently. Any failure of some of the nodes would not affect the availability of name service to other running nodes.

Acknowledgement

I wish to express my gratitude to my supervisor, Prof. M. R. Lyu, for his continuous support during the whole project that I have spent time under his supervision. His confidence in me, his sense of clarity and precision, his patience and open-mindedness have been essential factors to the completion of this project report.

I wish to thank my employer, my family, and my wife Lai for their faithful and loving support.

Table of Contents

1 Introduction

For the reasons of computing speed, system reliability and cost effectiveness, the interest in distributed computing systems has grown rapidly in the last decode. It has been claimed that, the distributed computing paradigm can improve collaboration through connectivity and inter-working; give better performance through parallel processing; increase reliability and availability through replication; provide scalability and portability though modularity; enhance extensibility through dynamic configuration and reconfiguration; and be more cost effective with resource sharing and open systems.

Distributed systems have the potential to be more extensible than centralized systems. For example, if an increase in the range of services is required (e.g., a dedicated processor could be added to a network which could be used by many people on the network) then it can be provided by adding a service and registering its presence so users can gain access to it. Although some centralized systems can also be extended in this way, distributed systems have the potential for much greater scaling. Distributed processing also presents solutions to some of the problems associated with centralized systems when constructing reliable applications. Most notably, in a centralized system to fail is high, leading to the situation where no services are available. Whereas in a distributed system it is possible to make use of the availability of resources on components with independent modes of failure to tolerate such failures.

One of the promising approaches in building Distributed System is using object-oriented (OO) technology [1]. This approach focuses on the development of reusable components that interact with one another through well-defined interfaces. Name service is one of the key components in a distributed system. It resolves defined names and provides object reference information to requesting object. This allows the requesting object to communicate/access the remote service/resource using high level names. Another similar but more sophisticated component is Trading Service. It provides remote objects to be selected without supply of their names, but based on the desired characteristics of the services. Name Service and Trading Service simplify distributed system administration and promote more flexibility and transparency throughout a network by automating distributed object selection.

Because components are distributed and the failure of one component does not automatically result in the failure of another component, this poses new problems not normally encountered in centralized systems. Failures such as message loss and corruption, processor crashes, and network partitioning, can create difficulties in maintaining the consistency of information stored over a number of components. Further, because of the increased number of components that make up a distributed system (machines, communication links, etc.) there is a corresponding increase in the probability that one or more of the components can be faulty. Although these failures need not cause the entire system to fail, they do cause problems in maintaining the consistency of data, leading to the necessity to create applications, which are dependable despite faults in the distributed system.

Name resolution services are key-component to the success of a distributed system. This project is to design and implement a name server, which keep track of all the running objects on a network. The name servers should run on all nodes. Application object can register with any node. The information should propagate to other node in a consistent way. We have chosen the CORBA (Common Object Request Broker Architecture) technology to be used for implementation. As CORBA is designed using object-oriented software development principle. It also provides common interface for development on different platform, so it well suites our needs for developing Name Service component to be run on all nodes in a network.

2 Review

2.1 Name Service

In daily life, we collect names and information about things, list and make them publicly available so that people can reference this information. For example, the names of people and organizations are compiled in telephone directories that provide such information as telephone numbers and street addresses. In a geographically dispersed society, telephone directories provide an efficient, easy-to-use method of locating people. Telephone directories also allow people to be found after they move. When people move within a city, their new address and telephone number are listed with their name in an updated edition of the telephone directory.

Computer networks likewise require names and directories to describe and record the characteristics of the diverse services and information they provide. For example, an electronic mail system must be able to locate users' mailboxes in order to deliver their mail. The mail delivery application contacts another application, called a directory or name service, to look up the users' names and indicate the location of their mail boxes.

In a distributed computing environment, anything that can be accessed individually and given a name is called an object. Examples of objects are network services, electronic mailboxes, and computers. Each object has a corresponding listing in the directory service, called an entry, which contains information, or attributes that describe the objects. Name entries can be collected in lists of entries called directories. For example, in a telephone directory, the listings are the entries, and the location information, such as the telephone number or street address, represents attributes of these entries.

Attributes can be any type of information that describes an object, such as location, color, or size. For instance, regular telephone directories contain only location-specific attributes, such as a street address. Business telephone directories include additional attributes such as the hours of operation and types of credit cards that the business accepts.

Name Service is central to the distributed computing environment because objects are defined by their names. Name Service allows clients to find objects based on names. It supports a name-to-object association called name binding. A name binding is always defined relative to a naming context, which refers to an object bound by a set of unique names [4]. However, there is no requirement that all objects in a distributed system must be bound to a name.

Applications and services gain access to an object first by accessing its name entry and retrieving its attributes. Decoupling the location or access characteristics of an object from the object itself is called location independence. It allows the applications and services to access an object even when the object moves or changes other vital characteristics such as language. Name services can be figured as the white page service [5] in a telephone network. An object client supplies a name to look for information of other object. It is just like a person using a name to look for details of other person or company, provided the target person or company has already registered within the telephone network. Similarly, an object has to register with a name server before it is available for look up in the Name Service.

Name service maintains one or more mappings from some form of name (e.g. normally symbolic) to some form of value (e.g. normally a network address) [9]. Name services can operate in a very narrow, focused way--for example, the Domain Name Service of the TCP/IP protocol suite maps short service names, in ASCII, to IP addresses and port numbers, requiring exact matches. At the other extreme, one can talk about general naming services, which are used for many sorts of data, allow complex pattern matching on the name, and may return other types of data in addition to, or instead of, an address. One can even go beyond this to talk about secure naming services, which can be trusted to give out only validated addresses for services, and dynamic naming services, which deal with applications such as mobile computing systems in which hosts have addresses that change constantly.

In current computer systems, three naming services are widely supported and used. As previously mentioned, the Domain Name Service (DNS) is the least functional but most widely used. It responds to requests on a standard network port address, and for the domain in which it is running can map short (eight-character) strings to Internet port numbers. DNS is normally used for static services, which are always running when the system is operational and do not change port numbers at all--for example, the e-mail protocol uses DNS to find the remote mail daemon capable of accepting incoming e-mail to a user on a remote system.

The Network Information Service (NIS), previously called Yellow Pages (YP), is considerably more elaborate. NIS maintains a collection of maps, each of which has a symbolic name (e.g., hosts, services, etc.) and maps ASCII keywords to an ASCII value string. NIS is used on UNIX systems to map host names to Internet addresses, service names to port numbers, and so forth. Although NIS does not support pattern matching, there are ways for an application to fetch the entire NIS database, one line at a time, and it is common to include multiple entries in an NIS database for a single host that is known by a set of aliases. NIS is a distributed service that supports replication: The same data are normally available from any set of servers, and a protocol is used to update the full set of servers if an entry changes.

X.500 is an international standard that many expect will eventually replace NIS. This service, which is designed for use by applications running the ISO standard remote procedure call interface and ISDN.1 data encoding, operates much like an NIS server. No provision has been made in the standard for replication or high-performance update, but the interface does support some limited degree of pattern matching. As might be expected from a standard of this sort, X.500 addresses a wide variety of issues, including security and recommended interfaces.

Name service is the computational equivalent of the controller of a naming domain [2]. It offers a name service interface for its clients with operations for binding, unbinding, resolving, comparing and communicating names. When using the interface, the client is always attached to an implicit local naming context that determines how names are interpreted and where name resolution starts. The client does not identify its local naming context in any way. Special operations, externalize and internalize, can be used for name communication between two or more users. The externalize operation converts a name into a generic name-string that can be sent to anyone through the infrastructure. The receiver can use the internalize operation to convert the generic name-string back to a name that can be used in the receiver's private naming context. Naming service is made responsible for creating and interpreting any internal structures that may be present in the name. Typically, compound names are created by separating name components with a separator character, but this choice is implementation dependent. The client simply uses both compound and simple names as if they were character strings.

Name communication does not require the existence of a universal name representation or a naming tree with a global root. The reason is simple: if two infrastructure implementations can send and receive data, they also have the means to mutually agree on any conversations between different name representations. In practice, when two infrastructures implementations are linked together, they also agree on how names externalized in one environment are internalized in the other environment.

Some applications even allow the user to specify combinations of names from different systems. For example, the Unix command rcp which accepts names such as ‘myhost:/usr/local/bin/readme’. This name has two components; the first, ‘myhost’ is the hostname and the second, ‘/usr/local/bin/readme’ is the pathname. The name therefore spans multiple naming systems (in this case where multiple = 2), and in naming circles is known as a composite name (with two components).

The problem is that each application defines its own rules for composing, parsing and looking up (resolving) these composite names. The user is required to remember which applications permit composite names, and which do not (e.g. the Unix command cp). A further problem is that applications which support composite names need to be changed every time a new type of naming system is added.

Consider these problems in terms of a heterogeneous distributed environment where the applications themselves are potentially spread across a global network. Every system in the environment is capable of defining many different sets of naming rules leaving users and programmers with the impossible task of trying to incorporate and coordinate them all.

One solution to these problems is proposed in the X/Open Federated Naming specification (XFN) [7] [8]. This specification defines a general model for naming systems that fits a large class of naming problems. It also provides a model for joining arbitrary naming systems together to form what is known as a federated naming system. Such a system allows composite names to be resolved across multiple autonomous naming systems and provides two major advantages over the traditional approach to naming:

1. A single uniform naming interface is provided to clients (distributed or otherwise.)

2. Adding new naming systems does not require changes to applications or other naming systems.

There are three kinds of names specified in XFN [8].

1. A composite name contains zero or more components composed according to rules defined by XFN, and can span multiple naming systems. Each component of a composite name is either a compound name or an atomic name.

2. A compound name is a name from a single naming system that contains one or more atomic names composed according to a naming system specific syntax or naming convention (e.g. the Unix pathname ‘usr/local/bin/readme’ is a compound name consisting of four atomic names, ‘usr’, ‘local’, ‘bin’ and

‘readme’, where the naming convention specifies that the order of composition is left to right, and the separator is ‘/’).

3. An atomic name is an indivisible component of a compound name such as ‘usr’ in the above example.

The basis of the XFN model is the mapping of a composite name to a value. In XFN terms the mapping is called a binding and the value bound to a composite name is called an object reference or (more usually) a reference. The naming system provides facilities to resolve a composite name to find its associated reference, and also for binding, unbinding, and renaming names etc. In the Unix file system example, the composite name ‘myhost:/usr/local/bin/readme’ is said to be bound to a reference to a file. The reference contains enough information to allow the client of the naming system (in this case, the file system itself) to locate the file on some storage device.

So, to design a Name Service, we should take the following into consideration: [4]

1. The design imparts no semantics or interpretation of the names themselves; this is up to higher-level software. The naming service provides only a structural convention for names, e.g. compound names.

2. The design should support heterogeneous implementation and administration of names. So, it should not have semantics limitation on the naming convention.

3. Names are structures, not just character strings. A struct is necessary to avoid encoding information syntactically in the name string (e.g., separating the human meaningful name and its type with a “.”, and the type and version with a “!”). which is a bad idea with respect to the generality, extensibility, and

internationalization of the name service. The structure define includes a human-chosen string plus a kind field.

4. Naming service clients need not be aware of the physical site of name servers in a distributed environment, or which server interprets what portion of a compound name, or of the way that servers are implemented.

5. Name service is fundamental. It should have no dependency on other interface.

6. Name contexts of arbitrary and unknown implementation may be utilized together as nested graphs of nodes that cooperate in resolving names for a client. No “universal” root is needed for a name hierarchy.

7. Name server should support – binding/unbinding objects, name resolution, creating/deleting name context, listing of name context.

2.2 Trading Service

Distributed systems span heterogeneous software platforms, hardware platforms, and heterogeneous network environments. In order to use services in such systems, service users must be aware of potential services and service providers. Furthermore, locations and versions of services change quite frequently in large distributed systems, which makes late binding between service users and service providers a useful feature. To support late binding, mechanisms must be provided to locate and access services dynamically. The concept of a trading service provides a mechanism for dynamically finding services [12]. The object that realizes the trading service is called a trader.

Trader allows clients to find objects based on given properties [3]. It facilitates the offering and discovery of object instance of particular type without knowing the actual naming of the object. It can be viewed as an advertiser, just like the yellow page service [5] of the telephone network. A person looks up and selects a company in yellow pages through service type. While a client looks for object/service with characteristics specification, the Trader may supply available objects that meeting the criteria.

Traders use name services for various reasons: for identifying offers, links, importer and exporter objects, and for example for type names, property names, protocol names, and policy names. Most of these names are used only within the trading domain and never passed to other domains. However, type names and property names are passed to other traders during federated imports [10].

A trader facilitates the exporting and importing of services. A service advertises its functionality to a trader (exporting). The trader then takes requests from clients and matches the desired functionality with an advertised service (importing). Figure 1 shows the interaction of an advertised service (exporter), a client (importer), and a trader. The distributed software architecture's goal is to create a generic trader that client applications use to obtain services. For example, when a client requests a query service from a trader, it receives references to any qualified query implementations. The user could then select from a variety of query services available.

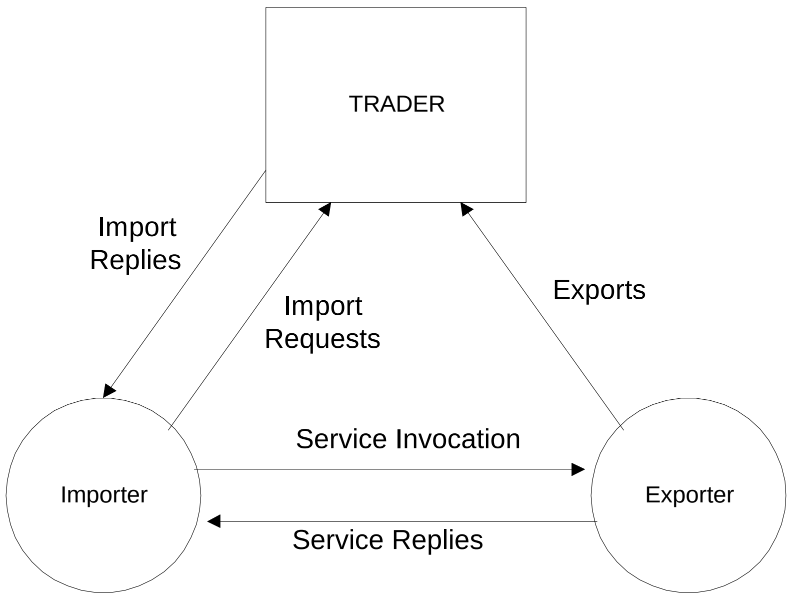

Figure 1 - Interaction among Trader, Advertised Service and Client

A trader accepts service offers from exporters of services when exporters wish to advertise service offers. A service offer contains the characteristics of a service that a service provider is willing to offer and the location of an interface at which the service is available. Service offers are stored by the trader in a centralized or a distributed database [11].

A trader accepts service requests from importers of services when importers require knowledge about appropriate service providers. A service request is an expression of service requirements made by an importer. A trader searches its service offer database to match the importer's service request. Moreover, if required, a trader can also select the most preferred service offer(s) (if one exists) that satisfies the importer's service request. The list of matched service offers, or the selected service offers, is returned to the importer.

After a successful match, the client, which requires a service, can interact with the service provider of a matched offer. The matching and selection of appropriate service at runtime by a trader allows client objects to be configured into an ODP system without prior knowledge of server objects that can satisfy their requirements.

Traders may be small or large, in terms of the number of offers they hold and the number of users they serve. Such a variety requires a range of Quality of Service options, e.g. response times measured in milliseconds or minutes. Thus, traders can be used for real-time trading, large scale trading [11].

There is also a requirement for some traders to be small in terms of the resources they consume. For example, in an embedded system, the trader holds a few offers for local services only and relies on interworking with traders on other nodes to give access to a larger set of offers. A trader may serve a large population of users or it may be private to a single importer/exporter pair. For example, one might want a trader on one's own node under one's private control. It could be completely isolated from other traders or it could interwork with one or more traders.

2.2.1 Service Offers

A service offer contains information about a service that is being traded. It is an assertion about a service that is offered for use by other objects at a computational interface. A service offer contains:

- the service type that identifies the kind of service on offer

- the interface to which a binding can be established to obtain the service
- zero or more property values for the service.

Service properties are expressed as name value pairs. A service offer must contain a valid value for each of the mandatory properties of the specified service type. However, optional properties need not have values specified. In addition, modifiable properties can contain, instead of a property value, an interface from which the actual value for the property can be retrieved at matching time. Such service properties are known as dynamic properties. The capability to support dynamic properties is optional.

Service offers provide information to a trader when it acts as a server to match service requests. A service offer identifier is assigned by a trader to each service offer stored in its offer space. This identifier uniquely identifies the offer within the trader information object. The set of all service offers stored in a trader forms part of the state of the trader information object.

2.2.2 Service Request

A service request contains the information provided by an importer to the trader so that the trader can match and select service offers from its service offer repository (or a subset of its repository) that will meet the importer's requirements. Importer policies in the enterprise viewpoint are represented by criteria in the information viewpoint. Criteria are requirement rules imposed by an importer for the matching of a service request against service offers, and are specified in a service request.

A service request contains:

- the required service type,
- the required service property values (matching criteria),
- for the trader to match its requirements.

A service request can, in addition, contain:

- some preference requirements for the trader to select and order the preferred service offers from the set of matched offers (preference criteria)
- some scoping requirements to restrict the extent of the search (scoping criteria)
- some capability requirements that scope the set of offers to be considered during matching (scoping criteria).
- a request for values of service properties in the matched offers.

A service request is represented in a Query operation in the computational specification, where the matching requirements are expressed as the type and constr parameters, the preference requirements as the pref parameter, scoping requirements as the policies parameter, and service property values as the desired_props parameter.

2.2.3 Trader Constraints

Each Trader object has its own characteristics and policies, for example: owner of a trader, search policy of a trader, capabilities supported by a trader, identifier of a type manager interface that a trader uses in order to access definitions of service types [11].

Trader policies in the Enterprise viewpoint are represented as trader constraints in the information viewpoint. Trader constraints are rules imposed by a trader that specify the requirements for the manipulation of trader information elements and structures. These include: Trader Matching constraint, Trader Preference constraint, and Trader Scoping constraint.

2.2.4 Applications of the Trading Service

1. Service Interaction Manager

Trading Service provides users with the ability to obtain information transparently about services available in a dynamic distributed environment. The Trading Service can be combined with other functions to provide other useful functions for users in a distributed environment, such as Service Interaction Manager [11] or SIM.

The SIM allows an importer of a service to:

- access all the import functionality of a trader;
- bind to an available selected service that the importer requires;
- initiate an available service that the importer requires. The initiate operation invokes an available service on an importer's behalf and is thereafter not involved
- in the interaction.

That is, an importer only needs to interact with the SIM to satisfy all the importer's required operations in a distributed environment.

2. Trader to provide ‘yellow page’ services

Trader can be used to trade services other than middleware services. The commercial activities of many organizations provide a useful analogy for traders. In essence, most organizations sell services. One of the many ways in which services are advertised is through the ‘yellow page’ (trader) maintained by some organization. Such a ‘yellow page’ directory typically has an index of general categories in which contact points are listed. When a service is to be imported, the importer first finds the category closest to its needs, and then finds the (hopefully small) set of organizations under that category.

The ‘yellow page’ directory service can be viewed as a trader that specializes in indexing other traders based on some categorization scheme. Many (overlapping) categorization schemes are possible, for example: categories based on existing Internet resources, e.g. research papers, library catalogues, Internet shopping, archive services; industry-based categories for further decomposition by an appropriate industry body, e.g. telecommunication industry, software industry, tourist industry; country-based categories, for further decomposition by categories within each country.

There is no reason why these categorizations cannot be used concurrently to give maximum flexibility and some redundancy (leading to reliability). Local trader policy could prioritize particular categories based on the needs of its local users. Users are also free to choose which ‘yellow page’ trader they wish to use both for advertising their own services and for importing other services.

Trading differs from repository services – like name servers, directory servers, and databases - in the assumptions made of the typical load profile. Traders are assumed to manage frequent updates and frequent queries [10]. The set of trader clients that are allowed to update the repository is large, unknown and constantly evolving. In addition, the structure of stored information evolves. The actions in which traders participate are not necessarily transactions and they are intended to be small in respect to required memory space and required processing time. However, keeping the agreed response time is critical, even more critical than being able to do a full search in the repository.

The functionality is independent of selected platform technology. The mechanism is specified on a logical level that can be implemented on a variety of platforms, programming languages, storage techniques and communication protocols. Due to this design choice, trading can be widened to a world-wide service: all trader realizations make their technical decisions independently, only considering the logical design and preparing to intercept technical differences.

In some technology-oriented aspects, general guidelines are required. However, such guidelines are highly flexible, as they offer multiple alternative implementation techniques. For example, traders may exchange information in object format or as lists of named values. Realizations of the trading functionality have to solve various problems. Trading makes information available to a large but still controllable set of users. The qualities of the available information should fulfil the user expectations: information consistency, freshness, and accuracy may be required, access times may be essential, and high probability for information availability may be crucial. However, the selected techniques may not affect the autonomy of the information producing systems nor the information user systems. Excess of network traffic must be avoided, as well as creating new security threats to the systems. Furthermore, the trading mechanism should be able to adopt to the evolution of the mediated information contents. Moreover, the topology of trader network that cooperates for the global trading service is constantly changing.

Lea Kutvonen has presented a list of Traders in his paper [10].

Simple trader

A simple trader can serve Lookup and Register interfaces. The register interface must support operations ‘export’, ‘withdraw’, ‘describe’, ‘modify’, ‘withdraw_using_constraint’, and ‘resolve’. A trader that accepts offers must specify what restrictions it has to the offer structure. The export operation takes an interface reference and a list of property values together with a service type name. It returns an object identifier, that can be further used as a key to the offer space in ‘withdraw’, and ‘modify’ operations (See Appendix I.) [10].

‘Describe’ is a subset of query operation at the Lookup interface. The motivation of including a simple query operation at the Register interface was related to conformance requirements. The simple query operation allows exporter objects to check whether their offers are correct and up-to-date and still be bound to the Register interface only. This is claimed to decrease the implementation cost of exporters because they are not required to include full query operation just for being able to have a type conformant interface with the trader they use.

‘Withdraw_using_constraint’ replicates the ‘withdraw’ operation. Instead of the offer identifier, the operation gets a matching constraint as an input parameter. The constraint language is capable of presenting a constraint that simply identifies an object. The operation ‘with-draw_using_constraint’ is result of the different design goals. On one hand, a single operation that is suitable for both end-user and administrative needs was expected. On the other hand, a simple operation was expected that could be required in all implementations. However, in the conformance requirements the whole interface is required. Thus, there is no acceptable way of supporting only the simple withdraw operation.

The ‘resolve’ operation translates a trader name to an interface reference, based on the naming system supported by the trading graph within a single administrative domain. If the trading graph exceeds the limits of an administrative domain, no trustworthy knowledge about the link names can be available.

Standalone trader

A stand-alone trader has the properties of a simple trader and includes in addition an Admin interface. The Admin interface includes operations for setting the various policies related attributes in the trader. It also has browsing operations for the offers and proxy offers.

Linked trader

A linked trader is similar to a stand-alone trader but has an additional Link interface for manipulating links. A linked trader must also be conformant to a Lookup interface in an importer role.

Proxy trader

A proxy trader is similar to a stand-alone trader but has in addition a Proxy interface for manipulating proxy offers. A proxy trader must also be conformant to a Lookup interface in an importer role.

The proxy offer contains service type and properties of a service offer. It is matched in the same way as normal offers. However, the proxy offer refers to a Lookup interface instead of the actual offered service. Therefore, the request is modified using the 'ConstraintRecipe' rules and forwarded to the Lookup interface. The original type parameter, preference parameters, and desired properties are passed on unchanged, but constraint parameters, policies and cardinalities are modified to suit the trader needs. With the parameter 'if_match_all' the proxy offer exporter can notify the trader that the proxy offer is a valid match for all imports of the specified service type irrespective of the importer criteria on property values. This approach would make the proxy offer similar to a type specific link, without any control mechanism for import propagation.

The proxy offer works like a simplified link. The type system in the object referred to and in the calling trader must be equal, but the query interface signature in the referred object can differ. Therefore, proxy offers can be used to encapsulate other objects than traders to the trading system.

The same functionality could be expressed by using interceptors together with links.

Full-service trader

A full-service trader includes Lookup, Register, Admin, Link and Proxy interfaces. It also has to be conformant with Lookup and Register interfaces in importer and exporter roles correspondingly.

2.3 Fault Tolerance

A centralized system can have a controlled physical and operational environment. Since a high proportion of system failures are the result of operational and environmental factors, careful management of this single environment can produce good availability. However, when something does go wrong the whole system goes down at once, stopping all users from getting work done.

In a networked system, the various computers fail independently. However, it is often the case that several computers must be in operation simultaneously before a user can get work done, so the probability of the system failing is greater than the probability of one component failing. This increased probability of not working, compared to a centralized system, is the result of ignoring independent failure.

On the other hand, independent failure in a distributed system can be exploited to increase availability and reliability. When independent failure is properly harnessed by replicating functions on independent components, multiple component failures are required before system availability and reliability suffer. The probability of the system failing thus can be less than the probability in a centralized system. Dealing with independent failure to avoid making availability worse, or even to make it better, is a major task for designers of distributed systems.

A distributed system also must cope with communication failures. Unreliable communication not only contributes to unavailability, it can lead to incorrect functioning. A computer cannot reliably distinguish a down neighbor from a disconnected neighbor and therefore can never be sure an unresponsive neighbor has actually stopped.

Global names -- the same names work everywhere. Machines, users, files, distribution lists, access control groups, and services have full names that mean the same thing regardless of where in the system the names are used. For instance, one's user name might be something like /home/user/student1 throughout the system. He/She will operate under that name when using any computer.

Name service is the key component for providing global names, although most of the work involved in implementing global names is making all the other components of the distributed system, e.g. existing operating systems, use the name service in a consistent way. Names -- provides access to a replicated, distributed database of global names and associated values for machines, users, files, distribution lists, access control groups, and services.

For global availability, the name services must work even after some failures. To achieve high availability of the service there must be multiple servers for that service. If these servers are structured to fail independently, then any desired degree of availability can be achieved by adjusting the degree of replication. As long as the failures do not exceed the redundancy provided, the service will go on working. For instance, we might decide to duplicate the name servers on each node in our network, but get by with one database replica per floor.

The most practical scheme for replication of the services in is primary/backup, in which a client uses one server at a time and servers are arranged (as far as possible) to fail in a good way, say by stopping. The alternative method, called active replication, has the client perform each operation at several servers. Active replication uses more resources than primary/backup but has no failover delays and can tolerate arbitrary failure behavior by servers.

2.4 Replication for Performance

The technique of data replication in distributed systems was initially used for fault tolerance issues, i.e., availability and reliability of the data in the presence of processor failures and network partitions. For example, if multiple copies of the same logical data are stored on different processors, the data can still be accessed if some of the processors are down. Nevertheless, it can also be used for saving communication overhead by reducing the number of messages between an object’s home node and the nodes that request remote accesses. [12]

Thus, data replication is useful in Name Service operation for two reasons. It can improve the performance and increase the availability and reliability of service. By accessing the copy in the nearest site, expensive remote access can be avoided. By storing critical data at multiple locations, the data may still be available even if some machines are down.

However, availability and consistency are competing goals in the management of replicated data. It is desirable to have high data availability while the database is still consistent in users' view. On the other hand, correct schemes that provide high availability may suffer performance penalties.

Henri E. Bal et al investigated the replication technique mainly for performance purpose [14]. It is first to identify between read operations and write operations on replicated data: a read operation does not modify the data, while a write operation (potentially) does. This matches with our application of Name Services, the read operation is an operation that does not change the internal data of the object it is applied to.

The primary goal of replicating shared data-object is to apply read operations to a local copy of the object, if available, without doing any interprocess communication. On a write operation, all copies of the object except the one just modified must be invalidated or updated [14]. To deal with this problem, communication will be needed, so write operations involve communication.

The second goal of replication is to increase parallelism. If an object is stored on only one processor, each operation must be executed by that processor. This processor may easily become a sequential bottleneck. With replicated objects, on the other hand, all processors can simultaneously read their own copies. Since a read operation does not change its object, it can be executed concurrently with other read operations consistently.

The effectiveness of replication depends on many factors. One important factor is the ratio of read/write operations on objects, which is determined by the user application. Another factor is the overhead in execution time for reading or writing objects. These costs are determined by the implementation of model. Henri E. Bal et al [14] stated the dependency as:

- The action undertaken after each write. If each write operation invalidates all copies, a subsequent read operation will need to do communication. If, on the other hand, all copies are updated, this disadvantage disappears, but write operations will become more expensive

- The protocol used for invalidating or updating copies. Many protocols exist (e.g., owner protocols, two-phase update protocols), each with their own advantages and disadvantages.

- The replication strategy. If an object is replicated everywhere, each read operation can be applied to a local copy, which is much cheaper than doing the operation remotely. On the other hand, writing an object that has many copies will be more expensive than writing a non-replicated object.

2.4.1 Invalidation versus Updating of Copies

If a write operation is applied to a replicated object, its copies will no longer be up-to-date. There are two different approaches for dealing with this problem. The first scheme is to invalidate all-but-one copies of the object. The second scheme is to update all copies in a consistent way.

With invalidation (or write-once), each object is initially stored on only one processor, say P. If another processor wants to do a read operation on the object, it fetches a copy of the object from P. In this way, the object automatically gets replicated. On a write operation, all-but-one copies are thrown away.

The alternative scheme is to update (or write-through) all copies of an object after each write operation. A problem here is how to update all copies in a consistent way. Updating of all copies should appear as one indivisible action. On systems supporting only point-to-point communication, a 2-phase protocol is needed. If reliable indivisible multicast messages are available, updates become much simpler.

There are several important differences between invalidation and update schemes. For one thing, keeping copies up-to-date is more complicated than invalidating copies, so the update scheme may require more messages to implement a write operation. Also, update messages will be larger than invalidation messages. An invalidation message merely needs to specify the object to be invalidated. An updated message will either contain the new value of the object or the parameters of the write operation, whichever is more efficient.

On the other hand, the update scheme also has several advantages. If an object is read after it has been written, the invalidation scheme will have to fetch the current value of the object from a remote processor. With the update scheme, this value will still be stored locally, so no messages need be sent at all.

In conclusion, which of the two schemes is most efficient depends on:

- The costs of the update protocol.
- The size of the object.
- The size of the parameters of the write operation.
- Whether the write operation is followed by a read operation or by another write operation.

Partial replication scheme [12] [15] are also found in some systems, in which each object is replicated on some of the nodes, based on compile time information, run time information, or a combination of both, is preferred to a no-or full-replication scheme. In static fixed replication scheme, programmer is required to give the compiler indirect knowledge of data layout by specifying how likely an application accesses to those data structures. An advanced scheme based on partial replication is to let the run time system decide dynamically where to replicate each object. Therefore, when a node wants to invoke a method on a remote object, it checks if it has a valid copy. If it has one, it creates a thread locally; otherwise, it sends a request to the home node to create a thread remotely. The home node counts read and write accesses from each node, to decide where to replicate. It replicates in nodes where read operations are frequent. It also has to see if many updates on this object are going on because it is better to keep a small number of replicas when the object is updated frequently. This replication strategy is especially useful if communication is slow, so the overhead of maintaining statistics is worthwhile, thanks to the reduced number of messages.

For an example of partial replication, Yixiu Huang and Ouri Wolfson have presented the CDDR (Competitive Dynamic Data Replication) algorithm [16]. They defined an object to be a unit of data to be replicated, and the replication scheme of the object to be the set of sites which hold an object replica. A data site is a site that belongs to the replication scheme. A non data site is a site that does not belong to the replication scheme. They supposed the read-write model is the following. At any point in time, one of the data sites is designated as the primary site (denoted it by p). Initially the replication scheme contains p alone. Whenever a data site issues a read request for the object, it is serviced locally. When a non-data site issues a read, the request is forwarded to, and serviced by p. A site s writes the object by sending it to p, and in turn p propagates the write to all the available data sites. In other words, the update policy is "read one write all available".

Every site in the network has a status. Status 1 indicates that this site is a data site, status 0 means that this site is a non data site. Every site knows its own status and where the current primary site is. The primary site knows every site's status. They defined the r-write (remote write) of a site s to be the write request issued from any other site (i.e. not from s) in the network.

The replication scheme changes as follows. If the primary site receives more read requests from a non data site j, then it will tell j to enter the replication scheme; whereas if the primary site receives more write requests from sites other than j, then it will keep j out of the replication scheme. The data sites decide by themselves whether or not to exit from the replication scheme based on the counters they have. If the data site i issues more read requests, then it will stay in the replication scheme, whereas if i receives more r-write requests from the primary site, then it will exit from the replication scheme. When the primary site needs to exit from the replication scheme, it has to choose a new site to inherit the primary role.

For further improvement, Rivka Ladin et. al. [17] presented the approach of Lazy Replication. This method can preserve consistency which providing better performance, It is applicable to application that has a weaker causal operation order.

Noha Anly also uses a non-traditional approach for achievement in performance [18]. Traditional approaches for managing replicated data are synchronous; that is, they require that read and write operations be synchronized in order to ensure that replicas are mutually consistent. Protocols requiring synchronization among a large number of replicas are difficult to implement across internetworks. They suffer from high latency and low throughput since links tend to be slow and unreliable and a large number of replicas generate considerable traffic over the network. Further, they lock or restrain access to resources during protocol execution and reduce the system availability when one or more nodes fail, or when the network is partitioned.

In contrast, weak consistency protocols allow updates and queries to occur asynchronously at any replica. They operate under the optimistic assumption that concurrent updates will rarely conflict and therefore synchronization at each step is unnecessary. Updates commit at the local replica, then they are propagated to other replicas and eventually, all replicas observe the updates. During the propagation, the data is in transient inconsistency and the value returned by a read request depends on whether that replica has observed the update or not yet. A weak consistency approach should provide a propagation mechanism which ensures that updates are efficiently and reliably propagated to all replicas even if the communication network does not provide such a guarantee. When link or node failures result in network partitions, replicas are allowed to diverge and continue providing service. When the partition heals, replicas merge their state and converge to a consistent state. Therefore, asynchronous approaches provide higher availability and better response time than synchronous approaches. However, this approach is based on the assumption that the applications can tolerate some inconsistency and reconciliation methods should be available to resolve conflicts. Typical applications that have used weak consistency are naming systems, information services, air traffic control and stock exchanges.

3 Extended Name Service

3.1 System Requirement

We will design and implement the extended name server which:

- may keep track of all the running objects on a network;
- may run on all nodes to support fault tolerant feature;
- allow object registration with any node;
- replicate information to other nodes in a consistent way.

3.2 System Design

We have selected the omniORB2 as the implementation of CORBA to work on. It is CORBA 2.1 compliant. It provides a basic name service with source code which supports all operations (bind, unbind, resolve, create, delete and listing) specified in the OMG specification [4]. This allows us to extend the name services with fault tolerant and replication features.

OmniORB2 is fully multithreaded. This will be very useful in doing group multicast in supporting asynchronous concurrent communication for monitoring, replication, etc.

We will design the name service base on group communication [22]. The group communication paradigm allows the provision of high availability through replication in a straightforward way (by gathering a set of replicas into a single group).

From specification, we can divide operations of name service into two types:

- No state change – operation that only query for information and does not change content of the name service database. As name service is going to run on each node and data will be replicated to all nodes consistently. All that query for information can be completed by local node's name service. These operations are **resolve** and **list** which will be unchanged with the original name service implementation.

- State change – operations that add, change or delete information from the name service database. For consistent replication of the change to all nodes, we will extend the name service with Group multicast and Group membership support. These operations are **bind**, **bind_context**, **unbind**, **new_context**, and **destroy** (also maybe **rebind**, **rebind_context** and **bind_new_context**) which will be overrided in the extended name service with Group multicast and Group membership features. (*bind* becomes an at least two steps operation involving group multicast; while *resolve* needs only one step operation)

For keeping the extension transparent to client, we add a group manager object (see Figure 2) to the name service and redefine the operations that will introduce state change. The original **bind**, **bind_context**, **unbind**, etc., will be re-wrapped as act_bind(), act_bind_context(), act_unbind(), etc, for actual handling of the individual naming data base.

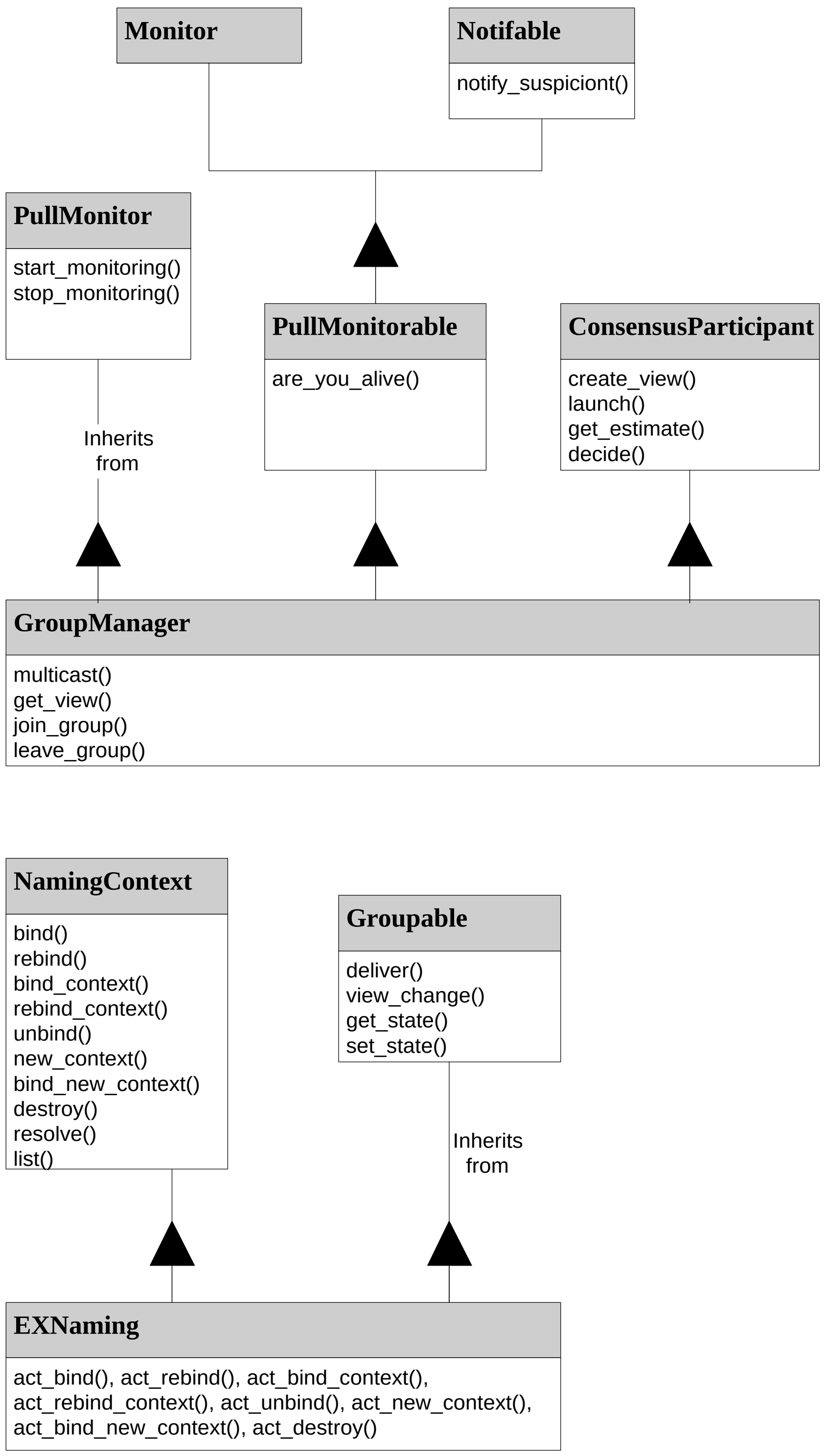

Figure 2 - Class Diagram for GroupManager and EXNaming Services

The group manager service is the core of our design. It is the service that actually provides object group support, and which has to be deal with by the extended name service. Together with classes it inherited from, it mainly provides:

- Reliable message service (not defined through IDL) which concerns reliable point-to-point and multicast communication.
- Monitoring service which uses reliable message communication for detecting object failure.
- Consensus service which uses failure detection and reliable message communication mechanisms to solve the distributed consensus problem.
- Group membership which uses failure detection mechanisms to monitor group members and a consensus service to agree on new composition of current active members.
- Group Multicast which uses reliable communication, consensus and group membership for atomic message delivery to all members of a group

3.2.1 Reliable Multicast

Reliable multicast is an essential mechanism for developing replicated and parallel applications. CORBA has not yet provided the service for non-blocking reliable multicast. Our solution to this is to use multi-threading. Each time a name service request that requires multicast to be issued, the group manager forks a new thread for each group member, and uses a standard CORBA synchronous invocation for each member invocation. As OmniOrb2 is fully multithreaded from the ground up design, we can have enough support for implementing this feature.

3.2.2 Monitor Service

For the group membership service, we need a detection mechanism that can provide information about component failures. This allows Group Manager to remove members from a group in case of a crash, such that further operation will not be affected or seriously hampered by the failure member. The failure detection is based on timeout mechanism. The choice of timeout values is crucial for the failure detector’s ability to respect accuracy and completeness. Short timeouts allow a process to detect failures quickly but increase the number of false suspicions, with a risk of violating accuracy. Hence, there is a trade-off between latency (short timeouts) and accuracy (long timeouts). As soon as a system involves more than one Local Area Network, the optimal timeout value between two services depends on both their respective locations in the system and on the characteristics of the underlying network.

We have defined the following objects (Figure 4) for failure detection:

- Monitor object will collect information about components failures.
- Monitorables are objects that may be monitored, i.e. to be detected for failure
- Notifiables register with monitor so that they will be notified about component failure.

We will use only the pull style model. Basically, the monitor periodically sends liveness requests to each member Name Service. If the Name Service replies, it means that it is alive. This model is easier to use for application development since the monitored service are passive, and do not need to have any time knowledge (i.e., they do not have to know the frequency at which the failure detector expects to receive message).

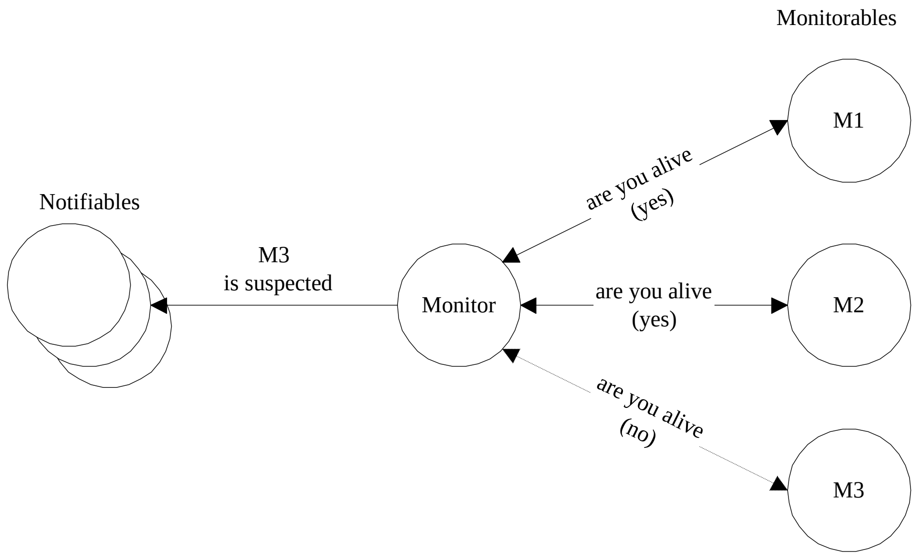

Figure 3 - The Pull Model for Object Monitoring. M1, M2 do response while M3 doesn't. M3 is considered suspected and status sends to clients (Notifiables)

A straight forward to implement a failure detector using the pull model is to have the services to support the specific operation, are_you_there(), which is invoked periodically. If the monitored service is alive and there is no communication failure, the invocation succeeds. If the invocation fails, the service is being suspected.

Among the group of Name Service, there will be 2 or 3 being assigned to start the monitoring for all members of the group. If Monitor is being suspected to be failure, other Monitor can be invoked to start_monitoring().

```
module GroupMonitor {
    enum Status { SUSPECTED, ALIVE, DONTKNOW };
    interface Monitorable {
    };
    interface Notifiable {
        void notify_suspicion(in Monitorable mon,
            in boolean suspected);
    };
    interface PullMonitor {
        void start_monitoring(in Monitorable mon, in boolean suspected);
        void stop_monitoring(in Monitorable mon, in Notifiable not);
    };
    interface PullMonitorable : Monitorable, Notifiable {
        oneway void are_you_alive();
    };
};
```

Figure 4 – IDL for Group Monitor

3.2.3 Consensus Service

As group members may receive request in different order in a distributed environment. This may cause inconsistency in the Name Service database. We need a consensus service to allow all members to reach a common decision, according to their initial values.

The design of the group consensus service is based on the algorithm of Chandra and Toueg [24]. It solves the consensus problem in an asynchronous system augmented by an unreliable failure detection mechanism under condition that a majority of the participating services do not fail.

Every group member participating the consensus will act as the consensus manager in turn. In each consensus, there will be four sequential phases:

1. Every member service sends its current estimate to the current manager. All these contribute to the initial estimate.

2. The current manager waits for a majority of estimates and selects one with the highest timestamp. That estimate is sent to all member services.

3. Every member service waits for the new estimate proposed by the current manager. Once a process receives the new estimate, it sends back an acknowledgment (ACK) message to the coordinator. However, if the member service does not receive the estimate and the coordinator is suspected by the failure detector, a negative acknowledgement (NACK) is returned to the coordinator.

4. The coordinator waits for the acknowledgements from a majority of processes. If none of those acknowledgements is a NACK, the coordinator decides its current estimate and reliably broadcasts the decision to all member services.

We use an implicit approach that does not need all consensus participants to launch the consensus. Each name service is attached to a group manager service. When the name service received a request message that requires consensus resolution, it invokes launch() on the ConsensusParticipant (i.e. the group manager, Figure 2), which starts the consensus algorithm [24]; each manager gets initial value from its attached name service, execute the consensus algorithm, and finally delivers the decision to the name services.

```
module GroupConsensus {
    interface ConsensusParticipant;
    // A consensus identifier
    typedef long ConsensusId;
    // A sequence of consensus participants
    typedef sequence<ConsensusParticipant> ConsensusParticipantSeq;
    // Object that represents a set of consensus managers
    interface ConsensusView {
        // Destroy a consensus view
        void destroy();
    };
    // Object that can lanuch and execute a consensus
    interface ConsensusParticipant {
        // Create a new consensus view
        ConsensusView create_view(in ConsensusParticipantSeq cms);
        // Lanuch a new consensus and return
        oneway void launch(in ConsensusId cid, in ConsensusView view);
        // Destroy the consensus Participant
        void destroy();
        // Return the participant
        any get_estimate(in ConsensusId cid);
        // Give the decision to the particpant
        void decide (in ConsensusId cid, in any decision);
    };
};
```

Figure 5 - IDL for Group Consensus

The interfaces for consensus are presented in Figure 5. We use untyped **any** variables for estimate and decision values, so that a consensus can decide on variable size name services request and reply data. A consensus is invoked by the launch() operation of the manager. This operation expects a unique identifier and a consensus view, i.e. the list of participating manager as parameters. Once a consensus has been launched, each manager gets the estimate of its associated participant by invoking the get_estimate() operation. Then, the consensus algorithm is executed between all participating managers. When the decision is reached, the managers give it to the participants through the decide() operation.

3.2.4 Group Membership and Group Multicast

The group multicast and group membership services interact closely. In particular, multicast operations are defined on object groups, thus involving group membership. Therefore, both features are contained in the set of interfaces forming the GroupManager (Figure 6).

- Group multicast – which provides support for sending multicast invocations to all the members of a group. It provides the primitives for sending invocations to groups to support the seemless view of client as sending invocation to a singleton object.

- Group membership – which manages the life cycle of objects groups. It maintains the updated list of all correct group members. It provides support for joining and leaving groups, view change notification, and state transfer. A group membership service is generally associated with failure detection mechanism for detecting group member failures.

Group membership is handled by the Group Manager service is a distributed system on behalf of the basic name services that create it. The composition of group can change over time. New members can join in, leave it explicitly or may be removed from group implicitly because of a failure. Name Service wants to join a group do so by contacting Group Manager service of any existing member. The Group Manager will update the list of group members. Once admitted to the group, the Name Service may interact with other group members. Finally, if the Name Service fails or leaves the group, the Group Manager will again update the list of group members. Dynamic group membership needs two operations:

1. Change of View – A view is the sequence of group members. Each time the composition of a group changes. The view_change() is invoked on each member. It ensures that every correct member of the group receives a view change notification that indicating the new composition of the group as a list of group members with mutually consistent rankings.

2. State Transfer – It is for transferring data from existing member to the new member. It is an atomic operation that happens during view change, when new member joins an existing group. It consists in obtaining (get_state()) database from current group member, and giving (set_state()) it to the new member.

We defines the interfaces and operations for dynamic group management and for state transfer (). The group management operations include the ability of group members to join a group and to leave a group using join_group() and leave_group() operations of GroupManager. It can also notify all group members of a view change using view_change() operation. The GroupView structure represents a stable view of a group at a given time. Group views have version numbers that are incremented every time the composition of the group changes. A group view is passed as parameter upon view change.

Also, we have two operations for group members: get_state() and set_state(). The state is transferred using a value type of any, which any kind of application-specific data. For Name Services, the state is the content of its database, which may grow quite large upon usage.

For data consistency, we need the consensus server for implementing group multicast and group membership. The role of the consensus is to agree on the respective ordering of the events received by the group members. These events are messages and view changes. Hence, each consensus instance decides on:

- An ordered set of messages to deliver.
- A set of suspected members to remove from the current view (i.e., the composition of the new view). This set can be empty, in which case there is no view change.

The consensus algorithm ensures that all correct participants eventually decide on the same value, and thus that every group member delivers the same set of messages and view changes in the same order. Messages are reliably multicast in the group before being ordered by the consensus algorithm.

The algorithm for total order and view membership is presented in Appendix II [23]. Each Consensus Participant maintains a set of unordered messages updated each time a message is received. The list of the suspected members from the current view is updated each time the group monitor (failure detector) notifies the participant about a suspicion or an “unsuspicion” (i.e., a suspected participant is trusted again). An ID variable is used as consensus identifier to synchronize the consensus instances run by all participants. A consensus is launched when there are messages to order, or there are members to remove from the current view. Each participant delivers the set of messages contained in the decision in a deterministic order that was agreed in the consensus by all participants.

If the set of suspected members contained in the decision is not empty, all members install a new view that does not include the suspected members. The process of joining or leaving an existing group is slightly different from that of removing a failed member. It is implemented through a totally ordered request issued by the member joining or leaving the group. This request is processed by every group member, which updates its view accordingly. Joining members receive service-specific information as part of the state transfer protocol (e.g., the composition of the group, the identifier of the next consensus to execute, etc.)

```
// IDL
module GroupMember {
    struct GroupView {// Composition of a group at a specific time
        // Group composistion.
        sequence<Groupable> composition_;
        // View identifier
        unsigned long version_;
    };
    interface Groupable {
        // Invoked upon message delivery
        any deliver (in any msg);
        void view_change(in GroupView view);
        // invoked upon state transfer on a current member
        any get_state();
        // Invoked upon state transfer on a new member
        void set_state(in any state);
    };
    interface GroupManager{
        // Get the latest group view
        GroupView get_view()
            raises(GroupError);
            // Join the Group
        void join_group(in Groupable member, , in GroupManager manager)
            raises(GroupError, AlreadyMember);
        // Leave the group
        void leave_group(in Groupable member)
            raises(GroupError, NotMember);
    };
};
```

Figure 6 – IDL for GroupManager

3.3 Work Flow

We use the omniORB2 version 2.7.1 as the working platform of CORBA. So, installation and environment setting is all refered to the "omniORB2 version 2.7.1 User' Guide"

3.3.1 Initialization

For the name service to be able to respond to a query from client, we can either:

1. supply the client with the stringified IOR of the name service;

2. or we can add to the configuration file omniORB.cfg. For example,

```
ORBInitialHost    sparc16

ORBInitialPort    12345
```

ORBInitialHost is the host name and ORBInitialPort is the TCP/IP port number. The parameter ORBInitialPort is optional. If it is not specified, port number 900 will be used. The second approach is much easier to specify than a stringified IOR. Another advantage is that it is completely compatiable with JavaIDL. This makes it possible for a client written in JavaIDL to share with a omniORB2 server the same Naming Service.

The extended name service consists mainly two objects (EXNaming, GroupManager). Upon initialization, EXNaming instantiates (1,2) the GroupManager through the Factory object [Figure 7]. With the returned reference to the GroupManager, it invokes the join_group() function (3) of the GroupManager with self-reference as the parameter.

The GroupManager optionally starts the monitor function by calling start_monitoring() with reference of EXNaming as parameter (4). Finally it invoke the view_change() function of EXNaming to finish the initialization (5) such that the group then contains one member of Naming Service.

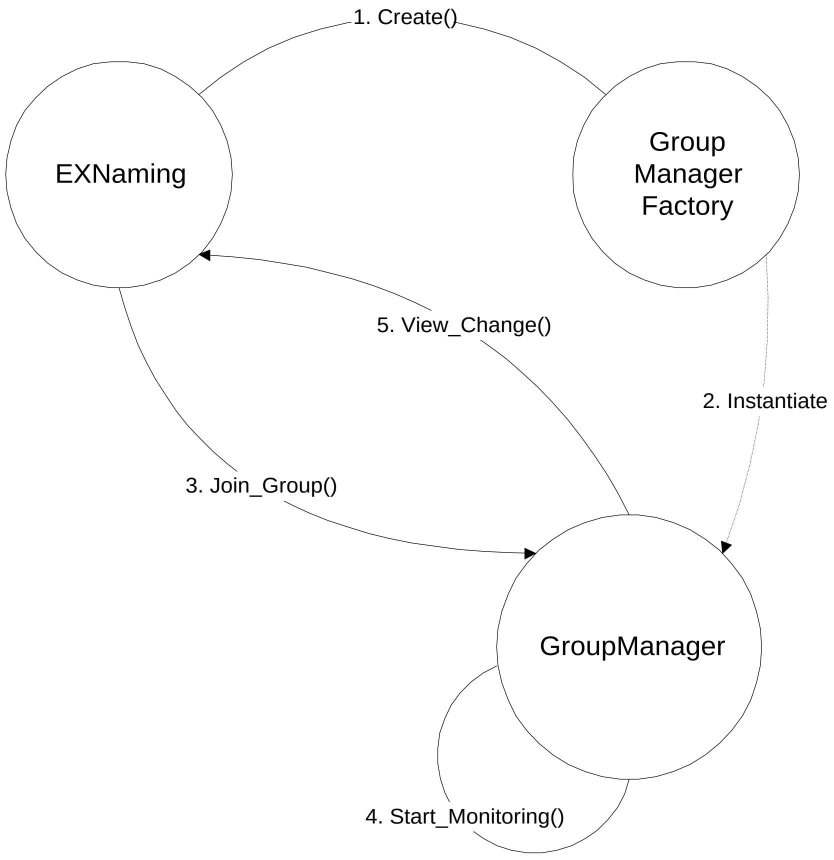

Figure 7 - Initialization of the Extended Naming Service

3.3.2 Running on all nodes

To support fault-tolerance, the extended naming service has to run on multiple or all nodes of the network. Each one can join in the group and leave the group dynamically at run time.

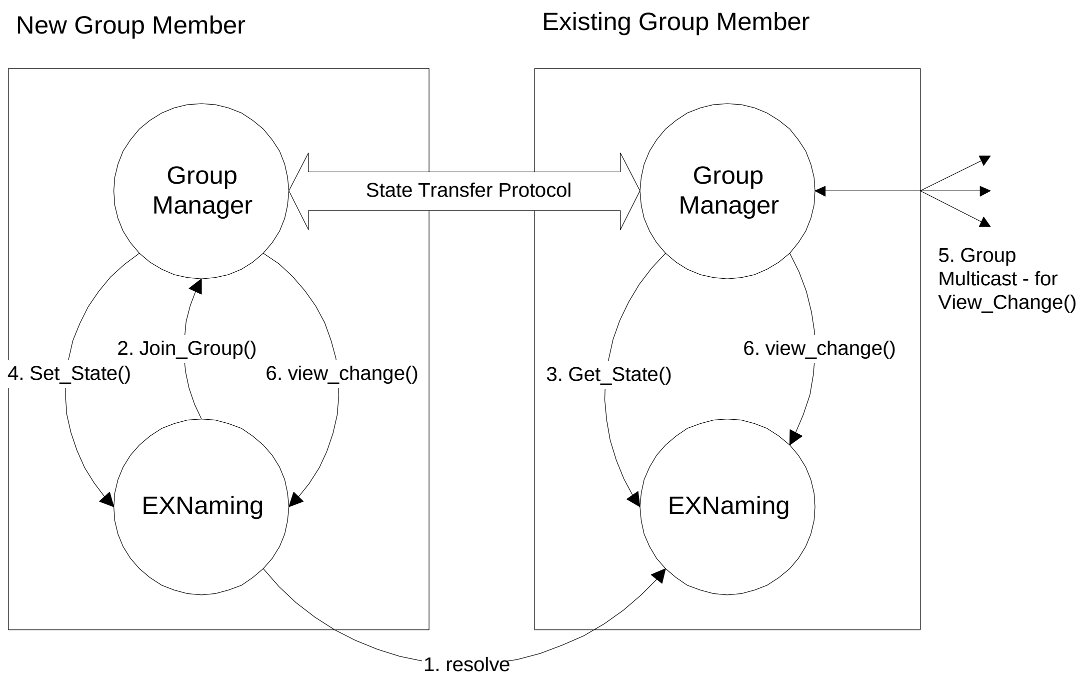

Figure 8 – Joining an existing Extended Naming Service group

To join in an existing group, new member naming service has to get reference to one of the existing group member (Figure 8). It can be done by either method suggested in Initialization section. After resolving the existing member reference (1) and creating self Group Manager, new member EXNaming invokes the join_group() from GroupManager with self reference and reference of an existing group manager (2). This allows communication with the target manager to transfer the current group information: the composition of the group, the identifier of the next consensus to execute and the naming database of the group (3, 4). Steps 3 and 4 may be called repeatedly for transferring name data to new member. If load balance is being considered, get_state() may be invoked on different members of the group. After state transfer, the existing group manager will multicast the view_change() message (5). Using the consensus service described in section 3.2.3 previously, every member including the newly joined one should receive the same ordered messages in decision. All members will then invoke the view_change() message consistently (6).

To leave a group, it is much simpler (Figure 9). The ready-to-leave member invokes leave_group() on its Group Manager (1) which in turn multicast the message to all other members (2). After consensus decision, each member invokes view_change() to update the new view of group membership (3). The ex-group member can then close, shutdown, or else relocate.

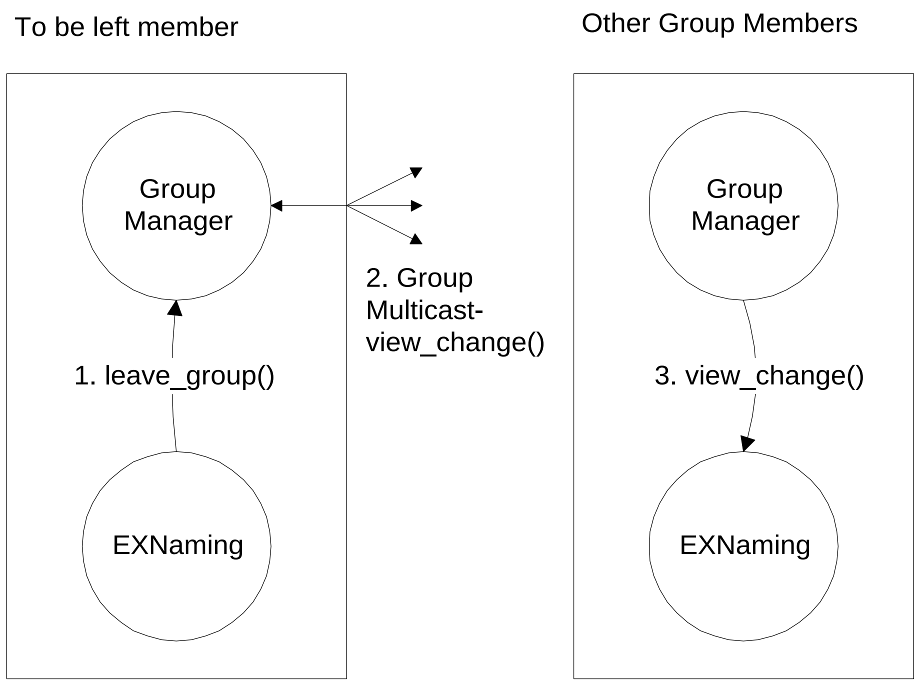

Figure 9 - To exit from a group

3.3.3 Replication

As the Extended Naming Service is running on multiple nodes and “grouped” together with the assistant of Group Manager object, client objects may register with the Naming Service on any node. The registration information will automatically be replicated to other nodes for future reference.

We take the operation “bind” as an example (Figure 10). A client object registers with the Extended Naming Service on one of the nodes. It supplies a name and object reference for binding (1). The EXNaming object invokes multicast() of Group Manager to this binding operation with parameters to all other members (2). Go through the consensus process (3) to make sure every member have a consistently ordered message list, all member Group Manager invoke act_bind() to actually register the object naming and reference into database (4). "bind" operation is here as an example only. Other operations, such as "unbind", "newContext" work the same flow where only the request message and actual working function call (act_bind(), act_newContext()).

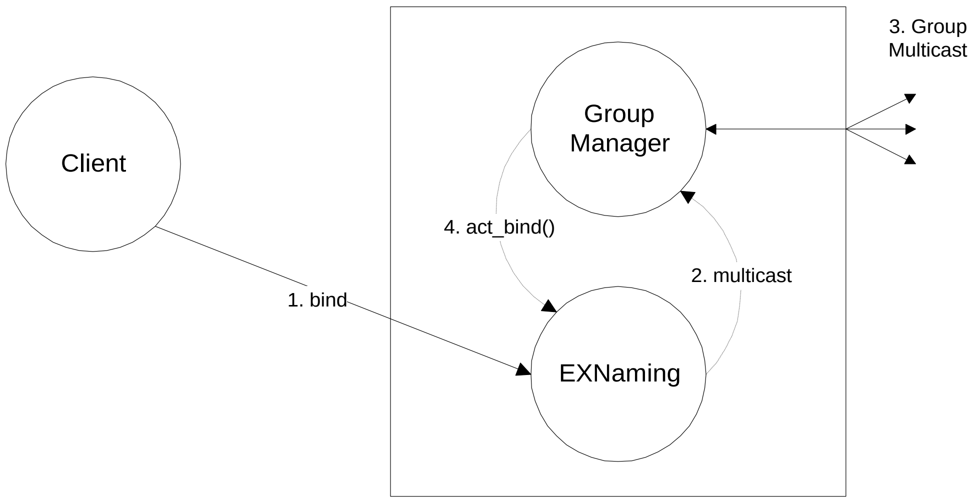

Figure 10 - Register at one node, replicate to all nodes

Since data is replicated to each group member on-the-fly, every instance of the extended naming service within the group possesses a complete copy of the naming database. To query for any naming information only need to invoke resolve() (or list()) operation (1) on the EXNaming object (Figure 11). No need to wait for consensus of other group members.

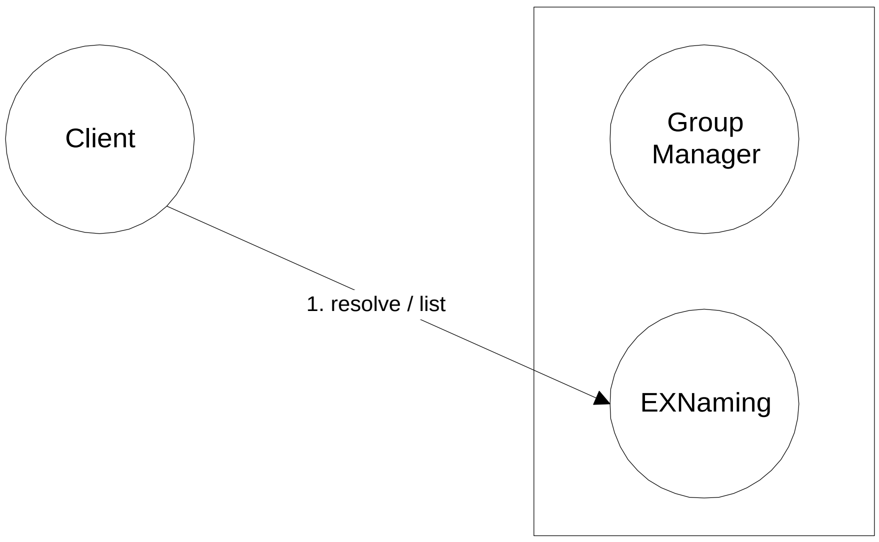

Figure 11 - Query object information with the extended name service

3.3.4 Fault Node handling

Before using the group manager service, the extended name service must create a GroupManager object. Upon creation, each group manager is associated with an extended name service object. We assume that the two components will not fail independently.

Some of the group managers may start their monitoring service (set by initial configuration) for monitoring all members of the group. Monitor pulls response from member periodically. Non-responsive member will be marked as suspected and be made notify to the rest of the member group (Figure 3). Suspected member may be trusted again if it does response to the next pulling query and before the next consensus starts.

When a consensus starts, for the reason of membership view change or message event that requires agreement on ordering, the consensus algorithm [23] not only makes agreement on the ordering message. It also decides on the set of suspected group members among all the proposed set. When final decision is made, every group members will handle the ordered message events and update its group member view by removing suspected member according to the decided set.

Removed group members can only re-join the group by invoking join_group() with one of the existing member.

4 Discussions

- CORBA does not differentiate between local and remote invocations issued to IDL specified operations. Stubs and skeletons hide the distribution from the application programmer, and allow both local and remote implementations. This programming model bears many advantages, but the developer has to deal with link failures and independent component crashes; these events have effects on the behavior of the system depending on whether the components are remote or local. In the design, we make assumptions about the locality of some service specific object. Group Manager, Monitor, Consensus and EXNaming which are associated with each group member are located in the same process. Local invocations are reliable, and we assume that independent failure of co-located objects do not occur.

- A desirable property of the group membership is to behave like a singleton object, i.e., to act as an identifiable, encapsulated entity that may invoked by a client. The key design issue to provide this abstraction consists in hiding the major differences between object groups and singleton objects. We do this by encapsulating plurality and behavior. So client accesses the naming service by a single invocation and receives a single reply of success or fail without other concern.

- Current design of the extended naming service has provided a level of client transparency, i.e. an advantage, such that current client application does not need to make any modification in order work with the new naming service. Because interface of the original Naming Service has been kept unchanged. The payoff is the client cannot keep on working without changing its target node deliberately if its target node service is down. If we want to have the transparency that client can automatically switch to other available Naming Service, we have to modify the client by adding a layer to interact with the service group.

- Current implementation of Name Service will write each change to its transaction log and commit the change periodically. In order to minimize the traffic for state transfer and increase the availability of service, we can pre-transfer the committed database from the target member to the new member before joining the group. Then, the existing member just needs to transfer non-committed data from its transaction log to the new member for state transfer.

- The problem of scalability is a major concern for a monitoring service that has to deal with large systems. A traditional approach to failure detection is to augment each entity participating in a distributed protocol with a local failure detector that provides it with suspicion information. However, this architecture raises efficiency and scalability problems with complex distributed applications, in which a large number of participants are involved. In fact, if each participant monitors the others using point-to-point communication, the complexity of the number of messages is $O(n^2)$ for n participants. Therefor, it is very important to reduce the amount of data exchanged across distant hosts. The simple interfaces of our pull-style monitoring service make it easy to configure the monitoring system in a hierarchy. The hierarchical configuration permits a better adaptation of failure detector parameters (such as timeouts) to the topology of the network or to the distance of monitored objects, and reduces the number of messages exchanged in the system between distant hosts.

- Although the implementation of the consensus service allows us to run several consensus instances in parallel, we have to serialize consensus executions. This is required by the total order and view membership algorithm (Appendix II). This restriction however does not completely slow down the system since (1) a consensus may decide on the ordering of several messages at once, and (2) it does not prevent non-totally ordered messages (e.g. resolve naming request) from being delivered.

- When a group of replicated objects invokes another group of objects (or a singleton object), request filtering must be performed so that target objects do not receive duplicate invocations. For example, several group members want to launch the Consensus targeting the same coordinator at the same time. Request filtering can be performed upon request reception (by the invokee). Performing request filtering upon request reception limits filtering to invokee that has knowledge about groups and filter. This is just fulfilled by our member group.

- Our solution to reliable non-blocking communication by using multi-threading with a separate thread for each invocation is conceptual simple and straight forward. But the use of thread sometimes may increases the application’s complexity and the probability of programming errors, by adding resource and synchronization problems. It will be best if OMG’s messaging service for this purpose can be finalize and available soon.

- As we will use multi-threading to asynchronously wait for incoming events and perform background task like failure detection. Multi-threading is supported by all major operating systems but, as of version 2.1 of the CORBA specification, thread support and management are not specified, and thus are not portable. Although different platforms provide the same object-oriented wrappers for threads, locks, and condition variables that allow platform independence, but these wrapper classes are not compatible with each other.

5 Conclusion

Distributed computing is one of the major trends in the computer industry. As systems become more distributed, they also become more complex and have to deal with new kinds of problems, such as partial crashes and link failures. We are interested in the naming service because it provides the key services for identifying objects in this ever growing distributed environment. We have reviewed about this service, its close relatives and some of the awaiting features (fault tolerance, replication, etc.). We have made a designed using group communication to extend the basic name service. We have also provided the complete IDL with class diagram and shown the workflow of the design. By replication on multiple nodes, the extended name service can provide fault tolerance and increased availability.

6 Appendix

6.1 (I) IDL for Extended Name Service.

```
// Extended Naming Service
#ifndef __EXNAMING_IDL__
#define __EXNAMING_IDL__
#pragma prefix "omg.org"
#include <ir.idl>
#include "naming.idl"

module GroupMonitor {
    enum Status { SUSPECTED, ALIVE, DONTKNOW };
    interface Monitorable {
    };
    interface Notifiable {
        void notify_suspicion(in Monitorable mon,
                        in boolean suspected);
    };
    interface PullMonitor {
        void start_monitoring(in Monitorable mon, in boolean suspected);
        void stop_monitoring(in Monitorable mon, in Notifiable not);
    };
    interface PullMonitorable : Monitorable, Notifiable {
        oneway void are_you_alive();
    };
};

module GroupConsensus {
    // Forward declaration
    interface ConsensusParticipant;
    // A consensus identifier
    typedef long ConsensusId;
    // A sequence of consensus participants
```

```
        typedef sequence<ConsensusParticipant > ConsensusParticipantSeq;
        // Object that represents a set of consensus managers
        interface ConsensusView {
                // Destroy a consensus view
                void destroy();
        };

        // Object that can lanuch and execute a consensus
        interface ConsensusParticipant {
                // Create a new consensus view
                ConsensusView create_view(in ConsensusParticipantSeq cms);
                // Lanuch a new consensus and return
                oneway void launch(in ConsensusId cid, in ConsensusView view);
                // Return the participant
                any get_estimate(in ConsensusId cid);
                // Give the decision to the particpant
                void decide (in ConsensusId cid, in any decision);
        };
};

module GroupMember {
        // The object is not member of the group
        exception NotMember {};
        // The object is already member of the group
        exception AlreadyMember {};
        // Error while performing an operation on the group
        exception GroupError { string description_; };
        //Error while performing an operation on a non-existent group
        exception NoGroup {};
        // Invalid group name (e.g. containing invalid characters)
        exception InvalidGroupName {};
        // Forward reference
        interface Groupable;
        // Composition of a group at a specific time
        struct GroupView {
```

```
        // Group composistion.
        sequence<Groupable> composition_;
        // View identifier
        unsigned long version_;
};
typedef sequence<any> AnySeq;

// Service's view of group members
interface Groupable {
        // Invoked upon view change
        void view_change(in GroupView view);
        // invoked upon state transfer on a current member
        any get_state();
        // Invoked upon state transfer on a new member
        void set_state(in any state);
};

interface GroupManager :
        GroupMonitor::PullMonitor, GroupMonitor::PullMonitorable,
        GroupConsensus::ConsensusParticipant {
        // Issue a multicast to the group
        AnySeq multicast (in any msg)
                raises(GroupError);
        // Get the latest group view
        GroupView get_view()
                raises(GroupError);
                // Join the Group
        void join_group(in Groupable member, , in GroupManager manager)
                raises(GroupError, AlreadyMember);
        // Leave the group
        void leave_group(in Groupable member)
                raises(GroupError, NotMember);
};
// Factory for creating group administrators
interface GroupManagerFactory {
```

```
            // Create a group GroupManager{
            GroupManager create(in string group_name)
                raises(GroupError, InvalidGroupName);
        };
    };

    module EXNaming  {
        interface EXNamingContext: CosNaming::NamingContext,
                GroupMember::Groupable {
            void act_bind (in Name n, in Object obj)
                    raises (NotFound, CannotProceed,  InvalidName, AlreadyBound);
            void act_rebind (in Name n, in Object obj)
                    raises (NotFound, CannotProceed, InvalidName);
            void act_bind_context (in Name n, in NamingContext nc)
                    raises (NotFound, CannotProceed, InvalidName, AlreadyBound);
            void act_rebind_context (in Name n, in NamingContext nc)
                    raises (NotFound, CannotProceed, InvalidName);
            void act_unbind (in Name n)
                    raises (NotFound, CannotProceed, InvalidName);
            NamingContext act_new_context ();
            NamingContext act_bind_new_context (in Name n)
                   raises (NotFound, CannotProceed, InvalidName, AlreadyBound);
            void act_destroy () raises (NotEmpty);
        };
    };
    #endif // __EXNAMING_IDL__
```

6.2 (II) Total Order and View Membership Algorithm [23]

```
1: {Protocol of the client}
2: procedure TO-multicast $(m, dsts)$                                   {Issue total order multicast}
3:     R-multicast $(m, dsts)$ to $dsts$                 {dsts is a set of processes (or a group)}

4: {Protocol of the server (code of process p)}
5: Initialization:
6:     $delivered_p \leftarrow \emptyset$                                {Messages already TO-delivered}
7:     $unordered_p \leftarrow \emptyset$                                     {Messages not yet ordered}
8:     $suspected_p \leftarrow \emptyset$                       {Suspected members from the current view}
9:     $cid_p \leftarrow 0$                                                  {Consensus identifier}

10: when R-deliver $(m)$                                               {Receive total order message}
11:     if $m \notin delivered_p$ then
12:         $unordered_p \leftarrow unordered_p \cup \{m\}$

13: when suspect $(q)$                                                     {Suspect a group member}
14:     $suspected_p \leftarrow suspected_p \cup \{q\}$

15: when unsuspect $(q)$                                               {Trust a group member again}
16:     $suspected_p \leftarrow suspected_p \backslash \{q\}$

17: when $unordered_p \neq \emptyset$ or $suspected_p \neq \emptyset$                  {Order messages and views}
18:     $k \leftarrow cid_p$
19:     propose $(k, \{unordered_p, suspected_p\})$                                   {Launch consensus}
20:     wait until decide $(k, \{unordered_k, suspected_k\})$
21:     atomically TO-deliver all messages in $unordered_k$ in some deterministic order
22:     $delivered_p \leftarrow delivered_p \cup unordered_k$
23:     if $suspected_k \neq \emptyset$ then
24:         install new view without the processes from $suspected_k$
25:     $unordered_p \leftarrow unordered_p \backslash unordered_k$
26:     $suspected_p \leftarrow suspected_p \backslash suspected_k$
27:     $cid_p \leftarrow cid_p + 1$
```

7 References

1. Douglas Schmidt and Steve Vinoski, “Modeling Distributed Object Applications”, SIGS, Vol 7. No. 2, February 1995.

2. George Colulouris, Jean Dollimore and Tim Kindberg “Distributed System – Concepts and Design”, 2nd Ed., 1995

3. Steve Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," IEEE Communications Magazine, Vol. 14, No. 2, February, 1997.

4. Object Management Group. CORBAservices: Common Object Service Specification. Object Management Group (OMG) March 1995. Updated July 1996

5. Dr. Michael R. Lyu “Distributed Systems – Topic 6: Naming and Trading”, Course notes for CSC7241, CUHK, 1998.

6. Kahkipuro, P., Kutvonen, L., Marttinen, L., “Federated naming in an ODP environment”, Joint International Conference on Open Distributed Processing (ICODP) and Distributed Platforms (ICDP), May 1997. Publ. Chapman and Hall, pp. 314-326.

7. Martin Chilvers, David Arnold, Andy Bond, Richard Taylor, “What's in a name? A Distributed, Federated Naming System in Python.”, CRC for Distributed Systems Technology, University of Queensland, Australia, http://www.dstc.edu.au/AU/staff/andy-bond/publications.html

8. "CAE Specification - Federated Naming: The XFN Specification" X/Open document number: C403 ISBN: 1-85912-052-0

9. Kenneth P. Birman, “Building Secure and Reliable Network Application”, Manning Publishing Company (Greenwich, CT) and Prentice Hall, 1997.

10. Lea Kutvonen, “Trading services in open distributed environments”, P.O. Box 26, FIN-00014 University of Helsinki, Finland, Lea.Kutvonen@cs.Helsinki.FI, PhD Thesis, Series of Publications A, Report A-1998-2, Helsinki, June 1998, viii + 231 + 6 pages, ISSN 1238-8645, ISBN 941-45-8223-3

11. Mirion Bearman, “Tutorial on ODP Trading Function”, DSTC, Faculty of Information Sciences & Engineering, University of Canberra, ACT, 2616 Australia, MirionB@ise.canberra.edu.au, Revised February 1997 to align with ISO/IEC IS 13235:1 | Draft Rec . X.950:1 (1997).

12. Mirion Bearman, Keith Duddy, Kerry Raymond, Andreas Vogel, “Trader Down Under: Upside Down and Inside Out”, TAPOS 3(1), pp. 15-29 (1997)

13. Suk Yong Lee, “Supporting Guarded and Nested Atomic Actions in Distributed Objects”, July 1998, University of California – Santa Barbara

14. Bal, H.E., Kaashoek, M.F., Tanenbaum, A.S., and Jansen, J., "Replication Techniques for Speeding up Parallel Applications on Distributed Systems", Concurrency Practice & Experience, Vol. 4, No. 5, pp. 337-355, August 1992.

15. Gagan Agrawal, “Availability of Coding Based Replication Schemes”, Symposium on Reliable Distributed Systems, 1992, pp. 103-110.

16. Yixiu Huang, Ouri Wolfson, “A Competitive Dynamic Data Replication Algorithm”, ICDE 1993, pp. 310-317.

17. Rivka Ladin, Barbara Liskov, Liuba Shrira, Sanjay Ghemawat, “Providing High Availability Using Lazy Replication”, TOCS 10(4), pp. 360-391, 1992

18. Noha Adly, “Management of Replicated Data in Large Scale Systems”, Corpus Christi College, University of Cambridge; A dissertation submitted for the degree of Doctor of Philosophy, August 1995.

19. C. Rigney, A. Rubens, W. Simpson, S. Willens, “Remote Authentication Dial In User Service (RADIUS)”, RFC2138, April 1997.

20. P. Chung, A. Baratloo, Y. Huang, S. Rangarajan, and S. Yajnik, “FilterFresh - Transparent hot replication of Java RMI servers”, in proceedings Conference on Object-Oriented Technologies (COOTS) April 1998

21. Sai-Lai Lo, David Riddoch, “The omniORB2 version 2.7.1 Users Guide”, AT&T Laboratories Cambridge, February 1999

22. P. Felber, Benoit Gaarbinatio, Rachid Guerraoui, “The Design of a CORBA Group Communication Service”, Ecole Polytechnic Federal de Lausanne, CH-1015 Lausanne

23. P. Felber, “The CORBA Object Group Service: A Service Approach to Object Groups in CORBA”, PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998

24. T.D. Chandra, S. Toueg., “Unreliable failure detectors for reliable distributed systems”, Journal of the ACM, 267, 1996