PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 9/45

A1

(11) International Publication Number: **WO 98/58313**

(43) International Publication Date: 23 December 1998 (23.12.98)

(21) International Application Number: PCT/AU98/00464

(22) International Filing Date: 17 June 1998 (17.06.98)

(30) Priority Data:

PO 7401	18 June 1997 (18.06.97)	AU
PO 9988	24 October 1997 (24.10.97)	AU

(81) Designated States: AU, CA, JP, US, European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.

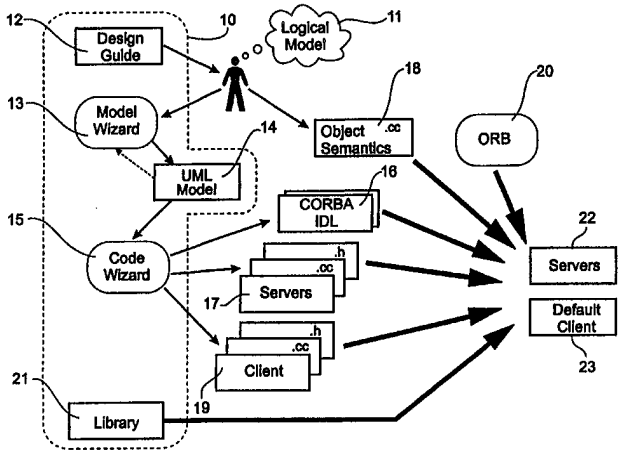
(71) Applicant *(for all designated States except US)*: CITR PTY. LTD. [AU/AU]; Level 2, South Tower, 339 Coronation Drive, Milton, QLD 4064 (AU).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: ZANDER, Andrew, Albert [AU/AU]; Level 2, South Tower, 339 Coronation Drive, Milton, QLD 4064 (AU). ROSE, Ian, Alexander [AU/AU]; 16 Pedley Street, Wavell Heights, QLD 4012 (AU).

(74) Agent: FISHER ADAMS KELLY; Level 13, AMP Place, 10 Eagle Street, Brisbane, QLD 4000 (AU).

(54) Title: SYSTEM DEVELOPMENT TOOL FOR DISTRIBUTED OBJECT ORIENTED COMPUTING



(57) Abstract

A development tool for building large scale distributed object oriented (LSDOO) computer systems, which systems typically include a plurality of clients (26), a plurality of servers (29), and a distributed object infrastructure (25) for communicating client requests for services to servers. The development tool includes a series of templates (12) providing predetermined object design patterns, a code generator (15) and, preferably, a set of basic distributed services (21). The code generator (15) is arranged to produce, from an object oriented system model (11) created by a user for defining desired server processes (33) to be requested by client processes (32) and incorporating selected ones of the design patterns, a client access layer (27) for each client process, isolating client application code from the distributed object infrastructure (25); a server access layer (28) for each server process, isolating server application code from the distributed object infrastructure (25); and a stub portion (29) of the server application code for implementing each service, including provision for the user to integrate an implementation of server semantics. The set of basic distributed services may include a file replication service (30) for replicating files within the system and a service finder service (31) for the discovery of the services available in the system.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

TITLE

SYSTEM DEVELOPMENT TOOL FOR DISTRIBUTED OBJECT ORIENTED COMPUTING

FIELD OF THE INVENTION

This invention relates to distributed object oriented computing systems, particularly large scale distributed object oriented systems (LSDOO). Such a system has objects associated with many machines, typically machines linked in a computer network, which cooperate to autonomously perform some business function. Guidelines on the size of system contemplated are typically those including some 1 million to 100 million objects, 100 to 1000 users, executing a total of 1000 to 50000 operations per second on approximately 100 to 1000 machines.

BACKGROUND TO THE INVENTION

Distributed, object oriented (OO) technologies offer developers of applications for installation on computer networks many potential advantages in deploying their applications. Object oriented techniques provide a controlled environment in which to manage complexity and change. Distributed computing allows applications to operate over a wide geographical area while providing a resilient environment in the event of a failure in part of the network.

In general terms, LSDOO systems have the potential to provide acceptable reliability for a minimal cost, allowing scaling by a factor of ten (with the upper end approaching global enterprise systems), support standardized interaction with other business-critical systems and efficiently support operations, not merely effect data storage.

An example of a known standard for building such systems developed by the Object Management Group, Inc. (OMG), a consortium of software vendors and end users, is the Common Object Request Broker Architecture (CORBA). The CORBA object request broker (ORB) is an application framework for providing interoperability between objects, which may be implemented in disparate languages and may execute on different machines in a non-homogeneous environment. CORBA is a very flexible architecture allowing the objects to transparently make requests and receive responses within the framework. Reference may be made to the *CORBA 2.0/IIOP Specification*, the *CORBA Services Specification* and other relevant specifications published by OMG.

Whilst CORBA and similar DOO architectures have a wide range of applications, their complexity results in high development costs for large systems. These costs flow first from the problem of assimilating the very lengthy specifications for these architectures, which provide a multitude of possible choices for addressing particular tasks, and secondly from the problem of identifying which combinations of these choices provide optimal solutions to broader design requirements.

Further, realizing the benefits of distributed, object oriented technologies is challenging, with many developer concerns to be addressed including:

- Scalability – how to build systems that scale to tens of machines and up to ten million CORBA objects.
- Performance – how to build systems with satisfactory performance.
- Fault tolerance – how to achieve availability for 24 hours of 7 days of every week.
- Persistence – how to efficiently and securely retain object state in a high performance distributed environment.
- Management – how to manage non-centralized computer systems.
- Implementation – how to contain development, testing effort and risk, while allowing high re-use of software.

European Patent Publication EP 727739 in the name of International Business Machines discloses a programming interface for converting network management application programs written in an object-oriented language into network communications protocols. International Patent Publication No. WO 97/22925 in the name of Object Dynamics Corp. discloses a system for designing and constructing software components and systems by assembling them from independent parts which is compatible with and extends existing object models. US Patent No. 5699310 in the name of Garloff et al. discloses a computer system wherein object oriented management techniques are used with a code generator for generating source code from user entered specifications. However these earlier disclosures do not describe a development tool and method for building LSDOO systems characterised by design patterns providing the power and flexibility of those set out below.

Glossary

Unless otherwise specified or apparent from context of usage, the following terms take the meanings attributed to them below:

"Attribute": an NState datum exposed by get and (usually) set operations.

"Broadcast Delegation": a delegation to all Confederates.

"Bulk Operation": an operation that accesses a lot (or all of) the NState of an object.

"Client": an application that requires access to one or more services;

"Cluster": the Peer-Cluster distribution model comprises machines organized into clusters. Clusters cooperate as peers and machines within clusters are specialized.

"Collection": an object that references a set of Members that possess some commonality; preferably an unordered set of objects;

"Confederates": the Collections that form a Federation.

"CORBA Object": an object for which a client may obtain a CORBA object reference.

"COSS": common object services.

"Delegation": one object involving another object to execute an operation.

"Design pattern": a design solution for addressing LSDOO issues that arise in system design.

"Directed delegation": delegation to an identified subset of Confederates.

"Distribution Model": describes how machines and data links are physically organized to implement a system.

"Explicit Group": a Group Operation, where the members are listed by identity.

"Factory": an object that creates other objects.

"Federation": a set of Collections cooperating to provide a faster, cheaper or more reliable service.

"Friend": a relationship wherein objects share IState, but appear to have independent NState.

"Gateway": a CORBA representation of a non-CORBA environment.

"Generic interface": an interface that identifies underlying problem domain commonality existing between specific interfaces.

"Graceful failure": a system responding to component failure by continuing to perform operations not directly affected by the failure.

"Group operation": an operation where one basic operation is applied to a number of objects.

"Identity": an NState datum that has a immutable one-to-one relationship with an object.

"Implicit group": a Group operation where the members are defined by a search predicate.

"IState": the state stored by an object's implementation.

"Life-cycle": the process of creating, copying, relocating and destroying an object.

"LSDOO": Large Scale Distributed OO, typically a system that has objects on many machines that cooperate to autonomously perform some business function.

"Member": an object that is referenced by a Collection.

"Natural collection": a Collection that is required to support system functionality.

"Normalized object": an object is normalized if the following are all true:

(i) the state of the object does not overlap with any other object;

(ii) the semantics of the object are defined and implemented by the object alone; and

(iii) its interface is just sufficient to access all state and change it in any way consistent with the object semantics.

"Normalized system": a system is normalized if all objects in the system are normalized.

"NState": Normalized State, the minimum amount of state needed for an object to exhibit the correct behaviour.

"Object": The OO concept of object, ie. an atom of state (NState) that has identity and is accessed through a defined interface. An object is the fundamental component of a LSDOO system, which may be a hardware device (such as a printer) or a software application (such as a print manager).

"Object reference": a pointer to an object that has an immutable many-to-one relationship to the object.

"OOPL": Object Oriented Programming Language.

"Partition": a physical grouping of objects, where each object is associated with exactly one partition. There is generally a one-one correspondence between a Partition and a set of computing hardware.

"Performance collection": a Collection that improves system performance by providing efficient implementation of Group Operations.

"Replica": One of the IState duplicates of a replicated Object.

"Replication": the duplication of an object's IState at several physical locations.

"RDBMS": relational database management system.

"Second class object": an object that cannot be referenced by a CORBA Object Reference.

"Service": an abstract provider of functionality, defined by CORBA IDL interfaces. (A service is typically provided by logically grouped objects co-operating with one another.)

"Tree descent": a form of recursive Delegation, where the delegates form a tree.

"Worm": a form of recursive Delegation, where the delegates from an arbitrary graph.

"Wrapper": an internal interface that protects system components from changes and defects in other components, common services and infrastructure.

OBJECT OF THE INVENTION

It is an object of the present invention to provide a development tool for large scale distributed object oriented computer systems which ameliorates or overcomes at least some of the problems associated with the prior art.

It is another object of the present invention to provide a method for developing large scale distributed object oriented computer systems which implements a small number of powerful CORBA usage models, allowing developers to focus on object modeling and implementing business logic.

It is yet another object of the present invention to provide a system development tool and/or method for large scale distributed object oriented computing that provides:

- guidelines on system modeling using CORBA;
- a modeling tool for translating abstract object models into deployable CORBA interface design language (IDL);
- a code generator, including code libraries, for generating infrastructure level code for producing CORBA clients, servers, factories and collections; and
- system management functions for distributed configuration, debugging, administration and performance measurement.

Further objects will be evident from the following description.

DISCLOSURE OF THE INVENTION

In one form, although it need not be the only or indeed the broadest form, the invention resides in a development tool for building a large scale distributed object oriented computer system, which system includes a plurality of clients, a plurality of servers, and a distributed object infrastructure for communicating client requests for services to servers, said development tool comprising:

(a) a series of templates providing predetermined object design patterns, including -

(i) an object identity pattern, facilitating unique identification of each object,

(ii) a collection pattern, facilitating the logical grouping of objects having some commonality,

(iii) a group operation pattern, facilitating operations targeted at a set of objects,

(iv) a friend pattern, facilitating association of one object with another object independently of clients, and

(v) a partition pattern, facilitating physical grouping of objects for performance

purposes;

(b) a code generator arranged to produce, from an object oriented system model created by a user for defining desired server processes to be requested by client processes and incorporating selected ones of the object design patterns, the following -

(i) a client access layer for each client process, isolating client application code from the distributed object infrastructure,

(ii) a server access layer for each server process, isolating server application code from the distributed object infrastructure, and

(iii) a stub portion of server application code for implementing each service, including provision for the user to integrate an implementation of server semantics.

Suitably the development tool further comprises a set of basic distributed services including a service finder service for the discovery of the services available in the system.

If required, the series of templates may further include one or more of the following object design patterns:

(vi) a federation pattern, a set of collections cooperating to provide an improved service;

(vii) a unified service pattern, facilitating the optimal choice of a collection from the set within a federation; and/or

(viii) a bulk operation pattern, facilitating multiple operations on a particularly identified object.

Preferably the object identity is an attribute of an object, is represented using a structured name and allows for object replication.

In preference, objects grouped into a collection are known as members and knowledge of a collection's members may be kept explicitly, such as in the form of a list, or implicitly by the application of a rule.

Preferably the set of objects to which a group operation applies is known as the scope of the group operation, which scope may be explicitly or implicitly defined.

An explicitly defined group operation is based on an underlying operation supported by objects comprising the service and a parameter to the operation, suitably a list of object identifiers, defines the scope.

An implicitly defined group operation is not based on an underlying operation and a client specified filter is applied to the objects to define the scope of the operation.

Suitably two objects are friends if they do not appear associated to clients via the distributed object infrastructure, but appear associated to one another.

Preferably a partition is a physical grouping of objects, wherein each object in the system is associated with only one partition, which partition corresponds to a set of computer hardware.

In preference, the collections in a federation are able to delegate operations to

each other in order to provide a faster, more extensive or more reliable service.

Suitably, a unified service is a federated collection wherein a predetermined sub-set of collections is transparent to clients.

The client access layer preferably includes agent classes to access the objects that implement a service and other classes to represent data structures manipulated by the agent classes.

In preference, the agent classes separate interface code for the distributed object infrastructure from the client application code and encapsulates an object's identity, interface and the means to address its implementation(s).

The server access layer may include service managers for managing objects with respect to any partitions and allows for the creation and deletion of objects.

The server access layer preferably includes adapter classes for providing access to objects that implement a service.

Suitably the set of basic distributed services further includes a file replication service for replicating files within the system.

The set of basic distributed services are preferably provided by code libraries.

If required, the system utilizes the CORBA standard, wherein:

(a) the distributed object infrastructure comprises an object request broker (ORB);

(b) the object oriented system model is modeled using CORBA concepts; and

(c) the server interface is generated in accordance with CORBA interface design language (IDL).

In further form, the invention resides in a method for the development of a large scale distributed object oriented computer system, which system includes a plurality of clients, a plurality of servers, and a distributed object infrastructure for communicating client requests for services to servers, said development method including the steps of:

(a) selecting one or more templates, from a series of templates for predetermined object design patterns, which include -

(i) an object identity pattern, facilitating unique identification of each object,

(ii) a collection pattern, facilitating the logical grouping of objects having some commonality,

(iii) a group operation pattern, facilitating operations targeted at a set of objects,

(iv) a friend pattern, facilitating association of one object with another object independently of clients, and

(v) a partition pattern, facilitating physical grouping of objects for performance purposes;

(b) creating an object oriented system model for defining desired server processes to be requested by client processes, which model incorporates selected object design patterns; and

(c) generating, from the object oriented system model, code modules for the following -

(i) a client access layer for each client process, isolating client application code

from the distributed object infrastructure,

(ii) a server access layer for each server process, isolating server application code from the distributed object infrastructure, and

(iii) a stub portion of the server application code for implementing each service, including provision for the user to integrate an implementation of server semantics.

Preferably the method includes the further step of providing a set of basic distributed services including a service finder service for the discovery of the services available in the system.

Preferably the series of templates available for selection in step (c) may further include one or more of the following object design patterns:

(vi) a federation pattern, being a set of collections cooperating to provide an improved service,

(vii) a unified service pattern, facilitating the optimal choice of a collection from the set within a federation, and/or

(viii) a bulk operation pattern, facilitating multiple operations on a particularly identified object;

If the object identity pattern is selected, representing the identity attribute of an object by using a structured name, which attribute may also allow for object replication.

If the collection pattern is selected, referring to objects grouped into a collection as members and keeping knowledge of a collection's members either explicitly, such as in the form of a list, or implicitly by the application of a rule.

If the group operation pattern is selected, referring to the set of objects to which a group operation applies as the scope of the group operation, which scope may be explicitly or implicitly defined.

If the friend pattern is selected, arranging friend objects such that they do not appear associated to clients via the distributed object infrastructure, but they appear associated to one another.

If the partition pattern is selected, assigning a physical grouping of objects to the partition wherein each object in the system is associated with only one such partition, which partition corresponds to a set of computer hardware.

If the federation pattern is selected, allowing the collections to delegate operations to each other in order to provide a faster, more extensive or more reliable service.

If the unified service pattern is selected, arranging a predetermined sub-set of collections within a federated collection to be transparent to clients requesting unified service.

The step of generating a client access layer preferably includes the further step of generating agent classes to access the objects which implement a service and other classes to represent data structures manipulated by the agent classes.

The step of generating agent classes preferably includes separating interface

code for the distributed object infrastructure from the client application code and encapsulating an object's identity, interface and providing means to address its implementation(s).

The step of generating a server access layer may include the provision of service managers for managing objects with respect to any partitions and facilitates the creation and deletion of objects.

The step of generating a server access layer preferably allows for adapter classes to provide access to objects that implement a service.

The step of providing a set of basic distributed services may further include the step of providing a file replication service for replicating files within the system.

In another form the invention resides in a large scale object oriented system built using the development tool or development method set out in any of the preceding statements, wherein the object oriented system includes a common administration interface.

Suitably the common administration interface facilitates remote management of all unified services in the system, including the provision of test, enable, disable, backup and restart functions.

Most suitably the administration interface also supports a set of attributes for which each unified service may be queried, including one or more of version number, copyright information, status, host machine, process identity or like attributes.

BRIEF DETAILS OF THE DRAWINGS

To assist in understanding the invention preferred embodiments will now be described with reference to the following drawing figures in which:

FIG. 1 is a diagram of a computer network over which objects may be distributed in a large scale OO system.

FIG. 2 is a diagram of usage model for the development tool of a first embodiment;

FIG. 3 is an overview of the architecture of the first embodiment;

FIG. 4 is a diagram illustrating the graph of a Collection design pattern;

FIG. 5 is a diagram illustrating a Collection implemented using a Pull model;

FIG. 6 is a diagram illustrating a Collection implemented using a Push model;

FIG. 7 is a diagram showing a Group Operation delegating to individual operations;

FIG. 8 is a diagram showing a Group Operation delegating to another Group Operation;

FIG. 9 depicts an example of the Friend design pattern;

FIG. 10 depicts a Peer-Cluster Distribution Model wherein Clusters relate to the Partition design pattern;

FIG. 11 shows a Federated Collection;

FIG. 12 illustrates an example of a Transparently Federated Factory;

FIG. 13 shows an arrangement of Federation interfaces existing between two

Confederate objects;

FIG. 14 illustrates an example of a Unified Service pattern;

FIG. 15 illustrates a typical client/server scenario involving Unified Federated Servers;

FIG. 16 depicts the operation of an object identifier (OID);

FIG. 17 shows the relationship between the Adaptor and Impl classes in a server process of the embodiment; and

FIG. 18 shows the interaction between the ServiceLocator service and a ServiceManager.

DETAILED DESCRIPTION OF THE DRAWINGS

Overview

FIG. 1 shows a computer network on which objects of an OO system may be distributed. The network may include computing machines, such as computer terminals or PCs 1 and file or print servers 2 interconnected to a network backbone 3 or other communications infrastructure for sharing instructions and data with one another. The computing machines may be located at widely spaced locations 4, 5 and 6, whereby branches of the network backbone may be interconnected by a switching device, such as router 7. In this context, an example of an object might be a particular printer 8 and an example of a service might be a file storage function.

Referring to FIG. 2, integers of the system development tool 10, as it might be implemented in the present embodiment, to produce a system compliant with the CORBA standard is shown. A working knowledge of CORBA is assumed in the following description, otherwise reference should be made to the CORBA specifications mentioned above. The developer first conceives a logical object model 11 using a common object modeling technique with the assistance of a design guide 12. In the embodiment the development tool further includes a model wizard 13 which assists the developer to specify a CORBA system model 14 expressed in a universal modeling language (UML). In other embodiments the system model may be produced by some other tool, such as another object oriented computer aided software engineering (CASE) tool, or the model wizard might be initialized with a pre-existing UML model or other OO model.

When the developer is satisfied that the CORBA system model is correct, the code generator in the form of a code wizard 15 takes the system model and produces:

- the CORBA interface design language (IDL) module 16;
- an implementation for the services 17, including hooks for the developer specified object semantics 18; and
- a simplified client interface 19 to the services.

Suitably the developer 10 codes object semantics 18 in an appropriate language, such as C++, which is conveniently the language used by the code wizard for the simplified client interface 19 in the embodiment. Other embodiments might use Java, SmallTalk or like

languages suited to OO software. A set of basic distributed services is also provided by the development tool library 21, which includes other common functions.

A CORBA compliant object request broker (ORB) 20, the environment within which the system operates, is also to be supplied. Commercially available products such as Visigenic's "Visibroker" or Iona Technologies' "Orbix" are suitable for this purpose. It is envisaged that appropriate target operating systems will be Sun Microsystems' "Solaris", Hewlett Packard's "HPUX" or Microsoft Corporation's "NT" or any other suitable multi-tasking OS.

The executables for the servers 22 and default client 23 are then produced by linking the generated modules 16, 17, 19 and developer hand coded modules 18 together with the externally sourced ORB components 20. The developer can extend or modify any of the code wizard output files in order to modify their initial choices or to use CORBA concepts that are more complex than those supported by the development tool. For convenience, the system development tool 10 of the embodiment may also be referred to hereinafter as "ORBMaster".

Architecture

FIG. 3 shows a typical client/server view of a generated service generated by the embodiment. Key aspects of the ORBMaster architecture shown in this diagram are the Client Application Code 26, the Client Access Layer 27, the Server Access Layer 28, the Server Application Code 29, and the basic services, which include the File Replication Service 30 and the Service Finder Service 31. It will be appreciated that the client process 32, the server process 33 and basic services communicate via the distributed object infrastructure, in the form of the ORB 25.

The ORBMaster architecture allows service developers to concentrate development effort in the areas of application code (Client Application Code and Server Application Code). It does this by providing some useful distributed services (file replication and service finding), code of some useful design patterns (group operations, unified service, etc., as discussed below). The architecture also provides client and server access layers (CAL and SAL) which separate ORB dependent code from application specific code, and code which implements distribution aspects of the service from code that implements the other service semantics.

The most fundamental component in the OO architecture is the object. In the client/server paradigm, groups of objects cooperate to provide services, and services are accessed by client applications. The ORBMaster architecture, in contrast to CORBA, relies on the concept of objects that have identity, support interfaces and have implementations. Identity is represented using structured names that are the attributes of the objects; interfaces are defined using IDL and implementations are addressed using CORBA object references. Accordingly, ORBMaster objects are just first class CORBA objects with the addition of identity. Identity is implicit in the design patterns described below.

There is a many-to-one relationship between naming attributes and objects. Naming attributes are read-only attributes of objects. One of the naming attributes of an object is designated as its ObjectIdentifier (OID). There is a one-to-one relationship between OIDs and objects. OIDs are used both as the database key and as the object_key (within the CORBA Interoperable Object Reference (IOR)) for the object. Object references either address the current implementation of an object or address nothing. That is, an object reference for one object can never subsequently be used to address a different object. Object references may change when an object moves, they therefore cannot represent object identity. As a consequence object references should not be persistified by clients (OIDs should be used instead). The architecture allows objects to have more than one implementation, that is, for objects to be replicated.

A service is functionality, logically grouped to meet some distinguished business need. A service is implemented by a group of related service provider objects. Services are identified by OID (i.e. name). A set of ServiceManagers that all support the same unified service are effectively a single replicated object (with service name as their OID). There are generally two kinds of service provider objects, as follows:

ServiceManagers - provide the operations of the service which deal with distribution management, group operations on collections of component objects, life-cycle management of component objects; and

component objects - the smallest separately identifiable components of the state managed by the service. Both ServiceManagers and component objects are implemented within server processes. The implementation of ServiceManagers and component objects are separated into the Server Access Layer (SAL) and the Server Application Code as shown in FIG. 2. The SAL is described in more detail in the section entitled 'Server Access Layer' below.

One aim of the system development tool, at least with respect to client applications, is to provide full access to the services without the need to understand the CORBA architecture. The ORBMaster Client Access Layer (CAL) seeks to achieve this aim. The CAL separates application code from code needed to access the objects that implement the services. The interface between the CAL and the application code does not expose any of the C++ classes generated from the Interface Definition Language (IDL) for the service. The CAL is discussed in more detail in the section entitled 'Client Access Layer' below.

Interface

This section contrasts the role of an interface in LSDOO with OOPL interfaces, such as C++ header files. The role of an interface is to offer a definition of a service. The role of an implementation is to implement the service by acting on the object's state to perform the defined operations. A developer needs a concept of object state to use the service. Admittedly, this may not be the actual state used in a real implementation, rather, it

is the concept of "NState" the Normalized state that a reference object would have. Without the concept of NState, the object's operations appear disconnected. For example, consider a basic name server:

```
interface NameServer{
        boolean add(in string name, in Object named);
        boolean delete(in string name);
        Object find(in string name);
};
```

This IDL precisely specifies the syntax of the interface. The token names imply the operation semantics. Assuming, for example, that the add() operation is associating a 'name' with a 'named'. If that association is valid (according to some set of rules), the return result will be true. The single return from find() indicates that name must be unique. Comments can increase this understanding. It implies inter-operation semantics. It could be assumed that the find() operation will return objects that match name and have suffered more successful add() operations than delete() operations. This understanding can be augmented through comments that describe NState. It says nothing about quality of service. That is, how fast, how many, or how reliable?

Failing to expose an object's NState makes the interface unusable as the operations have no interconnectedness. As an overreaction to RDBMS, OO designers have traditionally gone to great lengths to avoid discussing object state regarding this as merely an implementation issue. This reticence validly applies to implementation state (IState), however NState must be discussed. Moreover NState is not an amorphous blob, it has structure. One useful NState concept is that of Attribute. An Attribute is an NState datum which has an operation to get and usually set its value. Getting an Attribute does not change the NState. Setting it will either change its value to that proposed or, if it would violate the object semantics, fail. Setting one Attribute changes no other parts of the NState. A special type of Attribute is the *relation* this is an Attribute whose type is Object reference.

Another special type of Attribute is a *name* this is an Attribute that has a many-to-one relationship with an object. A special type of name is a *key* this is an immutable name. A special type of key is *identity* this is the key that has a one-to-one relationship with an object. NState that is not exposed as an Attribute will be exposed through operations *constant* operations are designed to leave NState unchanged.

The concepts of NState, Attribute, relation, key, name, constant operations and quality are too important to the functionality of applications to avoid in interfaces. However, none of these concepts are intrinsically expressible in CORBA's IDL grammar. The IDL attribute is close, but is sub-functional as it cannot support user defined exceptions. Therefore, conventions must be developed, published, and consistently applied to encode these concepts in IDL. CORBA does specify an Interface Repository (IR) service. However, this service has a very small role because it is solely a *syntax repository* it has no way of expressing any of the other complexities that comprise an interface. The IR is probably

solely useful for Gateways and rudimentary Browsers, that is, things which have no concept of a service beyond that of syntax.

Design Patterns

The system development tool of the embodiment provides an architecture for building distributed applications based on CORBA. This section describes a number of ORBMaster design patterns, which are templates for designing systems. Each pattern addresses one or more LSDOO issues. For further details of design pattern concepts see Gamma, E. et al.; *Designs Patterns*, Addison Wesley, New York, 1995 and Mowbray, T. J. et al., *CORBA Design Patterns*, Wiley, 1997. The patterns discussed below are particularly useful and find important roles in servers constructed with the assistance of the system development tool . The developer can selectively apply the design patterns to system design as indicated by the design guide 12 and the logical model 11, whilst focussing on the patterns that address the system's most pressing business priorities.

- Collection Pattern

A Collection is an object that references a set of objects possessing some commonality that addresses issues of performance and system modeling. This pattern works with the Friend and Federation patterns described below. A Collection is an object that references a set of Members. The Members are objects that have some form of commonality and it is this that the Collection manages. The following are examples of Collections:

- a Name server represents a set of objects, each of which has a name, thus the Collection supports search by name;
- a Topology server represents a set of objects, each of which has relations, thus the Collection supports search-by-relation;
- a CORBA-IP gateway consists of objects that represent interface protocol (IP) concepts and a Collection that represents the set of IP objects, thus the Collection supports IP-type operations, such as find the object corresponding to a given IP address; and
- an Error log is a Collection of error objects, thus the Collection supports the life-cycle of error objects, their retrieval, and related statistics.

A Collection always knows its Members. It may keep this knowledge explicitly, for example, as a list of the Members object references or implicitly via some rule, for example any object with an IP address matching 123.22.*.*. A Member can be a part of multiple Collections for example, a Printer object may be part of a name server Collection, an inventory Collection and a IP object gateway Collection. The knowledge a Member has about its Collections can vary. Some Members have a tight relationship with their Collection, they know its identity and are designed to interwork with it, for example, gateway members. Other objects may not be cognisant of who, if anyone, is collecting them. Such objects may support functions that easily allow them to be collected, such as life-cycle and state-change notifications.

Collection is the most pervasive LSDOO pattern and is highly likely to be

used in the design process. In an RDBMS system all the data is available in tables for you to access via a simple query language. In contrast a typical OO system starts with an initial object, this will reveal other objects, and those objects still others, until all the objects in which you are interested have been discovered. Any objects that are disconnected from the relation graph are unobtainable. With reference to FIG. 4, the root of the graph is the initial object reference 35 you obtain from the ORB using a statement such as CORBA::resolve_initial_references(). The Collections are the non-leaf objects 35, 36, 37 and 38 in the graph illustrated in FIG 4.

A major function of tradition OO modeling is identifying the Collections. Such identification and the techniques for doing it are similar for LSDOO. The following are common situations in which reveal Collections in an object model:

- attribute searching, for example, find the printers which are out of paper;
- group operations, eg. find printers which are off-line and set them to on-line;
- naming, eg. find the object with the IP address 12.34.56.78;
- containment relationships, eg. return a list of the printed circuit boards in the equipment; or
- connectivity, eg. find the least-cost path connecting two end-point objects.

Collections may return pointers to some of their members typically modeling searching, naming or containment, return some NState of their members analogous to table look-up, perform operations on their members (active Collections), manage the life-cycle of their members such as cascade delete. You can apply the Collection pattern in combination with other OO concepts. Many Collections have non-Collection aspects, for example, a printer object may be a Collection of its component objects as well as implementing the printer function. More examples of combining Collection with other OO concepts are:

- the Federation pattern, which allows Collections to cooperate in answering more wide-ranging queries;
- the Friend pattern, as discussed below;
- the Factory, wherein the combination of Collection and factory has an efficient implementation see further below; and
- Gateway, wherein a gateway to a non-CORBA environment usually has a Collection that represents the foreign environment as a whole, and individual objects that represent the foreign concept of object.

The interface to a Collection has operations which provide the Collection's functionality, for example, find members by name; has some operation to add and remove members (gateway/Collections are a possible exception); usually supports Federation; and supports cancellation and the incremental return of results, if results are large or responses slow. There are a number of approaches for adding and removing members. Some useful ones are a Factory if a Collection is also a Factory, every object created by the Factory is automatically a Member; offer a Collection supports an add member and remove member

operation; Gateway if a Collection is a Gateway into a foreign domain, membership of the Collection is usually expressed in that domain. For example, installing an IP router will cause membership of the IP Gateway Collection with the corresponding net mask.

Building efficient and flexible Collections is a challenge, scale is the nemesis of the Collection. The problem is how the Collection is able to maintain enough of its members' state to efficiently execute its operations. For example, a Printer Collection that support return a list of printers that are faulty could execute that operation either by polling each printer (which is slow if there are many printers) or by searching a local cache of printer state (which requires the cache to be kept in synchronism with the printer state). The Friend pattern provides one solution to Collective NState a Collection that is a Friend of its members can quickly access their IState. For example, finding the printers whose manufacturer=HP is easy if the printers IState is stored in an RDBMS. The Friend relationship has some disadvantages that are discussed in the friend pattern.

Non-Friend Collections must maintain a list of Members. This is enough to support the pull and push data sharing models, as shown in FIGS. 5 and 6:

- the pull model 40 requires the Members 42 to have a State interface 43 that allows the Collection 41 to access their NState as required. (The pull model can be used to maintain a cache, delegate on demand, or execute collective operations.)
- the push model 45 requires a Collection to have an Offer interface 47 that can be used to update the collective NState. (The push can come from either the Member 48 or a third party).

The push model is easy to implement from the Collection's perspective, at the expense of client complexity. The pull model is more Member-friendly, at the expense of the Collection. Delegation is usually slow and caches are complex to maintain using the pull model. Collections can be difficult to design and build and they tend to pervade design, therefore reuse of standard Collections should be considered. Whilst custom interfaces may be designed, they should be implemented using standard coded implementations.

Candidate standard Collections are as follows:

- Name Server for any Collection which has one or more globally unique keys and returns the corresponding object, for example, an IP gateway which converts an IP address to an object reference;
- OMG compatible Trader for any Collection which selects one object from many, based on attributes that rarely change, for example finding the best printer;
- OV Telecom Topology for any Collection where one object establishes some form of relationship with another object, for example the relationship between owners and owned objects. There are few good implementations of standard services at the moment, which somewhat limits this approach. Moreover, a standard solution will always be slower than a custom built one.

• Bulk Operation (or Natural Collection) Pattern

A derivative of the Collection pattern, which can be referred to as a Natural Collection, implements Bulk Operation. Multiple operations on a particularly identified object are considered to be Bulk Operations. A Natural Collection is a Collection required for a Normalized system model. Natural Collections are Collections that are required to implement the system's functionality, they are part of a Normalized system.

When designing an LSDOO system, in common with any OO system, you start with an abstract object model. This is a model that captures the logic of the system, without being polluted by implementation issues. You then map the abstract object model onto the particular implementation technology, in this case CORBA. Natural collections are a mapping of the abstract concept of collection. It is likely to be difficult to recognize the Natural Collections for a system, as they result from traditional OO modeling. Considered in object oriented programming language (OOPL) terms, there are some Natural Collections that may not recognize in an LSDOO system; these are:

- Many OOPLs have the concept of class data, for example, C++ or Java static member functions and data. DOO does not have this concept. You should explicitly model class data using Natural Collection objects.
- The OOPL concepts of construction, for example, the C++ and Java new() operators, are global operations not performed on any particular object. In DOO the new operation must be executed on some specific object typically a combined Factory/Collection object. When you perform event traces or CRC to test your object model, be very careful to examine how you found each object and what you used to create it. You must be able to trace each object back to the initial Object Reference you get from the ORB, that is, using the *CORBA::resolve_initial_references()* statement.

Natural Collection interfaces expand on the interface structure described for Collections. The following themes often arise in Natural Collection functionality:

- lookup by name for details of what constitutes a name, see the discussion of the role of *Interface*, above;
- search for objects conforming to some predicate formed from the object s Attributes;
- object life-cycle operations, particularly cascade delete; and
- collective themes specific to your project possible examples are path-selection, propagation, and best-choice. A common interface expression should be developed for these common themes. Here are some examples and caveats:
- Name your Collection interfaces predictably for example, a collection of Xs is called XCollection.
- There should be consistent support for Bulk Operations, notifications, and the like.
- Look for combining Performance Collection operations with your Natural Collection.
- Use exceptions carefully. For example "Object not found" is rarely an exception condition, it is an expected outcome of searching and name lookup. A recurrent theme of Natural Collection interfaces is the operation that returns a list of objects typically search

operations. There are several options for representing the returned objects:

- Return the object reference if there are not too many and your typical client will not immediately make a 'get attributes' call on each of the returned references.
- Return object Identity plus Attributes, if that is what your typical client needs and the Attribute size is not too big.
- Return object Identity if there are many objects.

If you return Identity, you must provide a Group Operation for converting Identity to Object Reference. If you do not, you are implementing non-CORBA compliant or Second class objects. In implementation, Collection interfaces may show a great deal of similarity (list return, cancel, incremental result, Federation and the like). This is all 'house-keeping' code that you should wrap on the client and server sides.

- Group Operation (or Performance Collection) Pattern

A further derivative of the Collection pattern, which might be referred to as a Performance Collection, implements Group Operations. Operations that target many objects rather than just one are considered to be Group Operations. The effect of a Group Operation is the same as repeatedly executing a single operation. There are two forms of group operation, Explicit and Implicit:

- an Explicit group is where the client lists the target objects for example, return the status attribute of the objects referenced by this list of objects; and
- an Implicit group is where the client specifies a membership condition and the group consists of all objects matching that criteria for example, execute self test on each object that has status faulty.

Performance Collections do not arise from the object modeling process. They arise from a diligent search of a system's dynamic behaviour, using techniques such as event. There are two conditions required for a Group Operation to be worthwhile first clients must be interested in the planned groupings and secondly the group operation must be significantly faster than the corresponding single operations. The factors that make an worthwhile explicit group include:

- the clients must hold a list of references you need to consider how that can happen; (The obvious way is that a client is given a list of object pointers by some operation, such as a search operation. A more subtle way is a client progressively accumulating individual pointers).
- the clients often perform the exact same operation on each object in the list; (For example, a client wishes to perform the test operation on many of printers to which it holds pointers).
- the group operation must be faster than the single operations. (This depends on issues discussed in Interface. A rule-of-thumb is that single operations returning less than 2kb of data are candidates for group operation).

The factors that make an interesting implicit group are as follows:

- some Natural Collection should offer an operation that takes a membership criteria and returns a list of object pointers typically some form of search operation;
- the clients often perform the exact same operation on each returned object; and
- the group operation must be faster than the combined search and single operations. All implicit group operations meet this criterion, except for extreme cases.

An important decision on explicit group interface design is how to point to the target objects? A CORBA Object Reference is a poor choice because it is slow to marshal, and hard to efficiently delegate (see below for discussion of the Federation Pattern). Some key attribute is usually a good choice. A Group interface should correspond to its underlying single operation interface. To aid the developers and maintainers of a system, a predictable correspondence is desirable. For example, the single operation:

Result X::M(in_args, out_args); would correspond to the explicit Group Operation:

sequence<Result>XCollection::M(sequence<Xid>,in_args, sequence<out_args>).

A mapping between the exceptions raised by the Single and the Group Operations should be defined. Generally, the Group Operation should not raise user exceptions as it will be unclear exactly to what an exception corresponds.

Implicit Groups can suffer population explosion if the Collection has N list returning operations and its Members have M Single operations, there are N*M implicit Group Operations on the Collection. This can be reduced to N+M by getting all the list-returning operations to return object Identity; and providing Group Operations on a sequence of object Identities.

A Performance Collection is far more effective when its Members are friends of the Collection. However, there are certain caveats of the Friend pattern. Friend will allow you to make these speed improvements:

- for group size N, you will save N-1 ORB round trips approximately 5(N-1) ms;
- if the Group Operations use object Identity rather than object reference, significant time will be saved in marshalling references, approximately N ms; and
- if the IState is held in an RDBMS, you will get substantially faster operation using bulk database operations. It is likely to be advantageous to Federate any Performance Collections. The general approaches to federated delegation are discussed below in relation to the Federation pattern. However there is a special issue in delegating group operations. Group operations should not delegate to a set of single operations as shown in FIG. 7. inefficiency. Rather group operations should delegate to group operations as depicted in FIG. 8, which is more efficient. The initial collection breaks up the group into a set of groups, each targeted to exactly one implementing collection. This should ensure that group operation performance is maintained.

• Friend Pattern

One fundamental property of an object is *encapsulation*, that is, the object state is available through a published interface only. Unfortunately, accessing encapsulated objects through a CORBA interface for current CORBA implementations is a relatively slow operation, typically allowing only a few hundred operations per second. The friend pattern 50 relaxes the strict encapsulation model as depicted in FIG. 9. Two objects 51 and 52 are friends if they appear to clients to be encapsulated 53, but do not appear encapsulated to each other 54. Friend is a useful pattern because the interface through which the friends communicate is designed to be faster or richer than a published CORBA interface. Friend objects share IState for performance, but do not share NState. They address performance issues and work with the Collection and Factory patterns.

An example of Friend behaviour is a Printer Collection that implements 'return printers that are off-line' by accessing the database in which the Printer objects store their IState. Friend behaviour need not be symmetrical. Using the example above, the Printer objects may never access the Collection IState. An object can be a friend of many objects of many types. If objects are implemented by the same process, they can share IState stored in memory. If the objects share the same disk, they can share IState stored in a database. Formally, Gateways and their members are inherently Friends the friend interface is the foreign domain. This discussion avoids this rather obvious case and instead focuses on where Friend is used for performance reasons.

Friend is an important pattern in extracting reasonable performance from LSDOO. This is because a friend interface can operate hundreds of times faster than a CORBA interface, while the overall system retains its essential OO character. Because Friend conflicts with pure OO concepts, its application should be limited to areas where it adds a significant performance benefit. Search operations on Natural Collections using the pull state model, and all operations on Performance Collections will be significantly faster if the collection and members are friends. Without Friend, these operations have either simple, inefficient implementations or complex, efficient implementations. With Friend, simple, efficient implementations are possible.

If one person is the solely responsible for implementation of the objects in a given system, then the friend concept will not be a limitation at all and the system will look OO and pure CORBA from without. If other developers are expected to implement some of the objects in the system, then those points of integration must also be pure CORBA, that is, not rely on Friend relationships. However it is rarely necessary that every object in your system be re-implementable independently of every other.

A group of objects that must be implemented together is the concept of extensibility boundary. Outside the boundary objects can be replaced at will, inside the boundary there are restrictions. By analogy, consider the hardware of a printer-PC system. The printer is outside of the extensibility boundary of the PC, therefore there is a published interface and any conforming printer implementation is acceptable. Consider now the toner

cartridge in the printer even though this is an encapsulated unit, it does not have a published interface and cannot be replaced with a different implementation. The toner cartridge is inside the extensibility boundary of the printer. Presumably the printer manufacturer could have negotiated and conformed to a toner cartridge industry standard, however the value of extensibility at that level of granularity was not worth the costs. In LSDOO terms, the printer and toner cartridge are friends, the printer and PC are not. It is as unrealistic for every object in an LSDOO system to be independently re-implementable, than it is to expect every component in a printer to be so. Friend is a candidate implementation for objects within an extensibility boundary.

The external interface to objects that are friends should not be influenced by that fact. That is, from without, the friend objects should appear as pure CORBA objects. The interface between the friends is discussed in relation to implementation below. Even though the objects appear to be pure CORBA objects, there are a three further levels of independence:

- Dependent, an object is Dependent if it has NState that can not be set to all valid values via the published interface. The term Dependent implies that there is some non-CORBA access to the object, this may be from some foreign domain or from some other object via a friend interface;
- Weak Independence, two objects are Weakly Independent if every operation could, in principle be done though a published interface. In practice, that implementation may be impracticably slow; and
- Strong Independence, two objects are Strongly Independent if one may be replaced with a new implementation without changing the system's function or performance. Typically, objects within an extensibility boundary are Dependent, objects outside the extensible boundary are Strongly Independent.

There are many techniques for implementing the Friend interface. Some example approaches are:

- One object can store its state in an RDBMS, and the other object can access the database. This approach is attractive for implementing search operations in Natural Collections, or Group operations in Performance Collections.
- The objects may share in-memory state, either in the same process, or shared memory between processes.
- The objects can use a CORBA interface, and achieve high speed by linking the client and server into the same process. Note that this is still a friend interface, it does not support Strong Independence.

Weak Independence is achieved through interface design. Strong Independence requires interface design and implementation design. Two approaches to Strong Independence are:

- Add, the Collection supports an add() operation which allows the addition of non-

friends. The Collection is therefore a mix of Friends and non-Friends. Assuming you want to provide egalitarian quality of service, an efficient non-Friend implementation has to be built, thus you can drop the Friend pattern altogether.

- Federate, the Collection supports a Federation interface which allows a foreign Collection to participate in implementing your Collection's operations. This provides Strong Independence and retains your efficient and simple Friend implementation. However, the foreign developer now has the burden of implementing part of the Collection as well as Members. In summary, although there appears to be two approaches, the "Add" approach is considered pointless.

An analogous situation is where IP routers externally expose IP ports, however, internally and between themselves they need not use IP protocols. As discussed above, the Friend interface gives two objects a privileged relationship. This privilege is not available to other developer's objects, therefore the extensibility of the system is reduced.

• Partition Pattern (or Distribution Model)

A Distribution Model describes how objects map onto machines, addressing the issues of performance, reliability, and manageability within a system. CORBA allows clients to be unaware of the location of objects the ORB guarantees that the system will work where ever objects are located. However, the location substantially affects the system s performance and reliability. A Distribution Model describes the following:

- what machines exist in the system, that is, their type, number, and purpose;
- which objects reside on which machines, that is, how Collections, Members, and Replicas are distributed over the machines; and
- what data flows exist, that is, how much data is transferred between each pair of machines. A Distribution Model exists as either a Meta-model or a Deployment model. The Deployed Distribution Model is that implemented by a particular customer. The Meta Distribution Model is the complete set of Deployment models allowed for by the system designer.

Both system and component designers need a Meta Distribution Model. System designers need a detailed model one that accommodates the needs of target users. Component designers only need a general model but with enough detail to demonstrate that the component is usefully deployable. System developers have an obligation to end-users to ensure that they actually design their Deployment Distribution Model users without a large system background will not expect to do this themselves. Success without a Distribution Model is as likely as randomly connecting cables, hubs, and routers would be in producing a LAN.

Component developers should expose CORBA interfaces to allow the Meta-model to be instantiated as a Deployment model. System developers can either use CORBA or foreign interfaces, such as command line or configuration files. The following implementation issues should be considered when designing a Distribution Model:

- the smallest and largest system size including a post-deployment growth path;
- data flows in particular identifying and exploiting cohesion between clients and servers;
- a system administration strategy including backup, software upgrade, and machine maintenance; and
- the effects of machine and network failures. Both Meta and Deployment models need to consider these things the difference is in generality. Meta-model design must interact with the system object model and interface design to exploit Collection, Federation, Replication and Friend patterns. Deployment model design must select machines, network bandwidth, machine location, system configuration, and end-usage. It is not necessary to be too ambitious for many reasons, it is unlikely that the largest deployed system will be more than ten times larger than the smallest.

The abstract goal of the Distribution Model is to reduce the cost of ownership. Apart from the obvious cost of ownership issues such as hardware, software, and the like, a large component will be administration. Rules of thumb for administration costs are, in a first order approximation the cost is proportional to the number of database machines, whilst in a second order approximation the cost is proportional to the number of things that have to be configured. The Distribution Model for many current systems is one server machine with several UI machines. It is reasonable to expect CORBA systems to deploy on tens, to possibly hundreds, of server machines. Beyond this level surprising difficulties may well be encountered.

FIG. 10 illustrates an example of one useful Distribution Model, the Peer-Cluster Distribution Model 55 that has the following salient features:

- Machines within each Cluster have special functions, eg. a database machine or an event handler.
- Clusters 56, 57 and 58 are equally functional peers.
- Entry level would have one Cluster comprising one machine, deployment can scale-up by adding either machines to Clusters or Clusters to the system.
- For many systems, this model will scale to about ten Clusters each of about ten machines.
- Administrative costs are reduced because:
  (i) the number of database machines is usually equal to the number of Clusters; and
  (ii) each Cluster is configured with only information about itself and the identity of its peers.
- Clusters are located near to external systems 59, 60 and 61 with which they inter-work.
- Users 62, 63 and 64 primarily inter-work with a nearby Cluster, ie. Clusters 56, 57 and 58 respectively. However, nothing precludes access to data located anywhere in the

system.

Clusters establish the concept of Partition. Partition is a design pattern embodying physical grouping for performance purposes. This is unrelated to the logical concept of domain, which is a logical grouping of objects for various administrative purposes. Many real world systems exhibit a peer cluster model, for example telephony switches are peers which have internal structure growth occurs by either internally expanding the existing switches or by adding new ones. The peer cluster model rarely scales beyond 100 machines.

- Federation Pattern

Federation allows Collections to cooperate to provide a better service, thereby addressing the issues of system performance and reliability. Many objects delegate by using some other object when executing their operations. Collections often use a specific form of delegation called Federation. FIG. 11 depicts a Federated Collection 65. Federation occurs when groups of Collection objects (known as Confederates 66 and 67) are designed to cooperate to provide a better service than they could individually; ie. a faster, more extensible and more reliable

Name servers are a good example of Federated Collections. Each name server holds a fraction of the name space and has pointers to other name servers. If a server cannot answer a request, it delegates to a server that can. The fact that a particular Collection federates is of interest to the client Federation is part of the Collection's NState model. For example, when using an IP name server, it is important to the client which fraction of the total IP name space is searched (the NState scope), it is not important how it is searched (IState model). Federation can also be:

- Transparent, in which case the client is not involved in achieving Federation. The Unified Service Pattern is a special case of Transparent Federation.
- Translucent, in which case the client has some involvement in achieving Federation.

Another special use of Federation applies to Collection/Factory objects. The COSS Life-cycle describes some models for an object Factory and a common theme is "where is the object located?" One approach is let the client decide the object is located on the same machine as the Factory. Another approach is to let the Federated Factories decide, thus the factories jointly decide location based on some algorithm unknown to the client. FIG. 12 depicts a Transparently Federated Factory 70 where, for example, each Confederate (72, 73) handles a particular IP address mask. Creation requests 71 are delegated by the Federation 70 to the correct Confederate. In the example from confederate 72 to confederate 73 which returns 74 a new object 75.

Federation is a key pattern for scaling up the number of objects managed by a system. When a system is scaled up and computers added to the system, more collection objects will need to be added. This is an inevitable consequence of physical implementation. However, clients should be insulated to the greatest extent from physical implementation

issues. Federation reconciles the conflict as it allows multiple physical objects while retaining the illusion of one logical collection. The following situations can be indicators that Federation is needed, they are illustrated using a PrinterCollection as follows:

- A Collection has Members on multiple machines. This indicates local Collections on each machine, with Federation of the Collections. The local Collections could exploit Friend implementations for improved performance. For example, a PrinterCollection exists on each machine that has Printer objects, the PrinterCollections delegate searches to each other.
- A client usually accesses specific subsets of Members but occasionally accesses more. This indicates Collections that exploit the localized data access pattern but Federate to access global data. Localized access, which is often associated with domains, is common in management systems. For example, PrinterCollection objects are organized on departmental lines. Most queries can be answered by the department's PrinterCollection, but it will delegate if needed.
- Graceful failure is required, ie. the system continues to perform operations not directly effected by a component failure. This indicates a Federated Collection that delegates operations to the working Confederates. For example, if a query requires a PrinterCollection that is not working, it will be avoided and a partial result returned.
- Extensibility by others developers is required, particularly where the other Collections have very different implementations. This indicates a Federated Collection, with different developers providing their respective implementations. For example, in a Federated Collection of IP and common management interface service (CMIS) managed printers, each Collection will issue requests in its respective protocol and interpret the responses into CORBA return values.

It is possible that Federation comes naturally from the Collection interface. For example, in a COSS Name Service the NameContext objects are Collections and the *resolve()* operation delegates to the other NameContext objects using their normal interface. Natural interfaces are usually associated with the Worm, Tree Descent and Directed Delegation implementations. Sometimes a special Federation interface is required, this is usually associated with Broadcast Delegation implementations.

The interfaces should support Transparent Federation (rather than Translucent), unless there is information available to the client that influences the operation of Federation and that information is difficult to give to the Collection. Federation is an algorithm that the Confederates jointly execute implying inter-object communications. Therefore, Federation and its attendant algorithm is part of the Collection's NState model it is not merely an IState issue. Admittedly, a transparently Federated service will have two types of clients Confederates and others the others will be uninterested in the Federation algorithm. Therefore, it is clearer to have a general and a Federation interface on the Confederates. FIG. 13 depicts the Federation interfaces 70 that exist between two Confederates 71 and 72, being distinct from the respective general or Client interfaces 73 and

74.

When considering implementation it is probably true that, for a multi-machine implementation, all Collections should be federated. Federation can give lots of advantages for little cost. However, the goal of the federation should be carefully considered (performance, reliability, or the like), to ensure the implementation does achieve the desired goal. The primary issue for implementing Federation is "how to achieve delegation?" Some example approaches to delegation include tree descent, worms, directed and broadcast delegation.

- Tree Descent requires Members to have a natural tree structure. The Collection delegates to other Confederates by descending the tree, for example fully distinguished name (FDN) resolution in a M.3100 network representation. Tree Descent has predictable and acceptable performance.
- Worms require the Members to be interconnected as a general graph. The Collection delegates by traversing the nodes in the graph, for example a 'shortest path algorithm' for network routing. Issues which affect Worms include cycle detection, goal seeking, and non-predictable worst case performance.
- Directed delegation is where the delegating Confederate can directly identify the delegated Confederate. This often arises in partitioned problems. For example, telephone number can be separated into a number of directories any number can always be delegated to the correct directory by examining the number. Directed Delegation has predictable and acceptable performance.
- Broadcast delegation is when the delegating Confederate cannot identify a particular delegate Confederate, therefore, the only solution is to delegate to all Confederates and accumulate the responses. For example, a Federated Alarm Log requires the query find the alarms that occurred in the last five minutes to be broadcast. Broadcast delegation has predictable but poor performance. All of these delegation approaches can exhibit graceful failure. Tree Descent and Directed delegations can be more authoritative in declaring they have totally executed the operation. Worms and Broadcast delegations are more affected by Members which are not part of the solution, therefore, these algorithms sometimes incorrectly report partial execution. To avoid a single point of failure, Directed and Broadcast Federations should offer the client a choice of Collection objects on which to execute the operations for details see the discussion of the Unified Service Pattern below.

Some conclusions flowing from the above discussion are as follows:

- Tree descent is a consistently good performer.
- Worms can perform particularly poorly, however, good performance is possible for some problems. For example, telecom networks are designed to have a short path (fewer than say five hops) between any two termination points a worm will perform well in discovering the shortest path in such a network.
- Directed Delegation is a good performer, provided the problem can be partitioned

such that every Confederate need not be visited. The larger the system gets, the more this will be required and the more likely it is to be true. For example, if there is three PrinterCollection objects, a particular query is likely to hit many of them if you have 300, it will still hit about three.

- Broadcast, though it responds in constant time, does not scale very well. After the question: "How to delegate?"; comes the question "To whom?". The Worm and Tree Descent approaches typically have the delegation target implied in the Member and Collection interfaces. For example, COSS Naming delegates based on context names that are NState concepts.

Broadcast delegation has a simple solution delegate to all the Confederates. All you need is a method for Confederates to tell each other of their existence a Replicated registration base is a good approach. Directed Delegation is more complex, it can be broken into three parts:

- Who are the Confederates? A registration base, as for Broadcast delegation, is a good approach.
- How do they partition the space? You need a partition rule, based on either a natural key (such as FDN) or a surrogate key (such as a partitioned sequence number). Good manageability requires each Confederate store its own partition rule. The others inquire of it when needed this avoids the need for a global configuration base.
- How to key into the space? Operations make the key explicit, for example passing object identity or some other key.

One subtle problem that affects broadcast delegation is looping the initial Confederate broadcasts to its peers that then must know not to re-broadcast. There are many solutions, some of which pollute the client interface. The best approach is to use a specific federation interface whose implementation does not re-broadcast. Efficient delegation is generally important for a distributed system. It is a design decision that is difficult to change it impacts design of object keys and identity. If done poorly, it will limit your ability to scale and federate. This discussion has identified some implementation approaches, however, a thorough analysis is required for particular systems.

Analogies to the Federation pattern may be drawn with Name services such as DNS or X.500, and with a help desk which can answer simple queries, whereas more complex ones are delegated to more experienced staff and defects are directed to development engineers.

It is tempting to have a private Federation interface this makes the Confederates distributed Friends. This breaks the Strong Independence characteristics of Friends. As such, it may be acceptable for a product, however, it is rarely acceptable for a reusable platform component. The system should preferably be designed to minimize use of Broadcast and contain the performance of Worms. It is tempting to design Federated Collections that can answer globally scoped queries. This is acceptable for a system needing

one to ten machines or Clusters. It fails thereafter, for details see the section above dealing with the Distribution Model Pattern. You could design a non-CORBA Federation scheme one approach is to publishing a database schema. This low-cost mechanism is a Friend relationship and, therefore, has the inherent limitations associated with violating object encapsulation.

- Unified Service Pattern

A Unified Service is a convenient to use and reliable Federated Collection that accordingly works with the Friend, Collection and Federation patterns. The desirable characteristics of a Federated Collection is one:

- that is transparently Federated;
- where there is a mechanism for a client to obtain a Confederate based on the name of the service alone;
- where all Confederates respond to any operation identically with the possible exception of speed;
- where the Confederate returned to the client shall be one that has acceptable (preferably good) performance; and
- where the client can obtain another Confederate if one fails. These characteristics give a Unified Service reliability and usability.

Any Federated Service implemented using Broadcast or Directed delegation 76 should be a Unified Service. Unified Service 75 has an interface 78 to select 76 and re-select 77 Confederates based on the name of the service. A Unified service is built as a front-end to a Federated Service, as depicted in FIG. 14. The front-end requires the ability to select the 'best' Confederate 79 from those available. It is tempting to implement this as a Trader operation. However, single point-of-failure can limit this approach. The implementation of 'best' should consider the following:

- Confederates that are not working are not 'best'.
- Confederates that are lightly loaded are better than those that are heavily loaded.
- Confederates which have a shorter network delay and higher bandwidth connectivity are better.

Ticket agencies are analogous to Unified Services. They have many Confederates (booking clerks), there is a Confederate selector (phone directory and call distributor), all Confederates can provide the same service (but some are slower), and failure of one Confederate has no effect on the service. It should be noted that Tree Descent and Worm delegated Federations execute operations on a Member and are generally not candidates for Unified Services.

FIG. 15 illustrates the operation of United Federated Servers in a client/sever scenario 80. There is provided an ORB 81 that is in communication with a client 82, a first server 83, a second server 84 and a service finder 85. The operation of Service Finder Services is described in more detail below. Entities A, B and C are available from first

server 83, whilst entities D and E are available from second server 84. An example of the operation of the first and second servers, which are United Federated Servers, is as follows:

1. client 82 requests the service finder 85 for a server (which supports a specified service) and the service finder returns the first server 83;
2. client 82 invokes operation on the first server 83;
3. first server 83 invokes operation on the second server 84, because the request cannot be totally serviced by the first server 83;
4. first server 83 returns a result (entity E) to the client 82, which client in turn invokes operation on entity E.

Functional View

This section provides a functional view of an ORBMaster generated service. The ORBMaster architecture recognizes that there are common functions performed by all services, which common functions are addressed by the architecture. In many cases they are completely implemented by generated code or library code. The common functions are grouped together into the following categories:

- distribution management;
- group operations;
- management of client/server interactions; and
- component object lifecycle. As described in the introduction to services above, these functions are performed by ServiceManager objects.

• Distribution Management

In the ORBMaster architecture all ServiceManagers are unified service providers. This means that the details of the distribution model adopted by a service are hidden from the clients of that service. The unified service design pattern allows clients to treat all component objects as belonging to a single collection regardless of their distribution. In order to be able to provide this single collection view, all ServiceManagers must support the following interfaces:

resolve, given an OID, return the object's reference or return the *ORBMasterIDL::NoSuchObject* exception.

In addition, ServiceManagers, which allow clients to create component objects must support:

create, given (at least) the read-only attributes (except OID if the service generates this), create a new object or return the *ORBMasterIDL::ObjectAlreadyExists* exception.

Note: These interfaces are encapsulated within the ORBMaster generated Access Layers; see the overview above in conjunction with FIG. 3.

The system development tool supports both the replication of component objects and the partitioning of component objects for distribution models. If component objects are replicated then their state is stored by more than one of the ServiceManagers for the given service. The ORBMaster architecture enables replication of objects by separating

the concepts of object identity from object implementation. The current embodiment of the system development tool does not provide direct support for replicating object state. However, the ORBMaster File Replication Service can be used by service developers for this purpose. Other embodiments of the system development tool will provide more support for replication. If component objects are partitioned then their state is stored by only one ServiceManager. For these services two related issues that must be addressed, namely OID resolution and component object location management.

OID resolution (mapping from OID to object reference) for partitioned component objects is logically a two step process. First find the authoritative ServiceManager, then ask the authoritative ServiceManager to resolve the OID or return the

ORBMAsterIDL::NoSuchObject exception

ServiceManagers are authoritative for OIDs if they can determine, without reference to other ServiceManagers, whether or not the OID represents a component object of the service. Authoritative service managers also create the component objects for which they are authoritative (if the service supports object creation).

When object partitioning is used object resolution is based on an object registration tree. An object registration tree is a tree where the nodes represent authority for the sub tree of which they are the root. The nodes have names bound to them and OIDs are structured names where components correspond to these names. The OID may have more components than those corresponding to nodes in the object registration tree 90, as illustrated in FIG. 16.

In the example the OIDs consist of three components, service name, chunk name and id, ie. OID = "printer/chunk2/123/24". Whilst object registration trees in the present embodiment have depth 2, other embodiments may support arbitrary OIDs and object registration trees of any depth. OIDs are absolute names defined relative to a common root 91. The intermediate nodes 92 and 93 effectively group the services by service name, for example "printer" node 92. The leaf nodes 94, 95, 96 and 97 in the object registration tree are ServiceManagers that support the resolve and (if the service supports it) create operations.

ORBMaster generated services provide support for their users (typically only those acting in system administrator roles) to specify where component objects will be located in two ways:

- by specifying which ServiceManager will create a given component object (this is expressed using a partitioning rule); noting that not all services allow clients to create component objects; and
- by moving component objects after they are created; wherein nodes in the object registration tree are objects which support the

CosLifeCycle::LifeCycleObject::move operation, thus allowing groups of component objects to be moved based on their OIDs. A partitioning rule defines a

mapping from values for a sub-set of the attributes given to an object when it is created to the name bound to a leaf node in the object registration tree.

If one of the attributes to the create operation is the OID it is an error if the operation is invoked with an OID which does not also resolve to the same node in the object registration tree to which the partitioning rule maps. Object creation (where supported) of partitioned component objects is logically a two step process:

- find the authoritative ServiceManager, to which the partitioning rule maps; and
- ask the authoritative ServiceManager to create the new object and return its object reference or return the *ORBMasterIDL::ObjectAlreadyExists* exception if the object already exists.

• Group Operations

Group operations are those operations that apply to more than one component object. The set of objects to which a particular invocation of a group operation applies is known as the scope of the operation. Group operations are supported by ServiceManagers, rather than component objects. Group operations are categorized according to the following criteria:

- scope definition;
- support for partial completion;
- delegation method; and
- transactional behaviour.

The scope of a group operation is defined either explicitly or implicitly. Explicitly defined group operations are based on an underlying operation supported by component objects of the service, are equivalent to invoking the underlying component object operation for every object in the scope, have their scope explicitly defined by a parameter to the operation (a list of OIDs), exist solely for reasons of implementation efficiency, and have their interface generated by the system development tool based on the developer specified underlying component object interface. Whilst implicitly defined group operations are not based on an underlying operation on an individual component object, have their scope determined by the service by it applying a client specified filter to all the component objects of the service, exist because they implement problem domain semantics identified by the developer (rather than for reasons of implementation efficiency), and have their interface partially generated by the system development tool based on the developer specified interface. Implicitly defined group operations are typically query interfaces.

The code generator generates the interface for an explicitly defined scope group operation from a developer specified component object interface. The component object interface which is used as the basis of a group operation preferably should:

- return void;
- have zero or more in parameters of any type;
- have an optional out parameters which can return any type;

- only raise user exceptions that contain an *ORBMasterIDL::ReturnCode* structure as their data contents.

Component operations should therefore be of the form:

```
interface <interface> {
    void <base operation> (
        [<in arg list>, ]
        [ out <result type> <out param> ])
    [ raises ( <exception list> ] ;
}
```

where:

<interface> is the name of the component object interface defined by the developer;

<base operation> is the developer defined underlying operation on a component object;

<in arg list> is the optional list of in parameters;

<result type> is the type of the out parameter;

<out param> is the optional out parameter; and

<exception list> is the optional list of user exceptions raised by the operation.

An example of the IDL generated by the system development tool is:

1. A structure (in the scope of the module that contains *<interface>*):

```
struct <interface><base operation>Result {
    boolean valid;
    [ <return type> <out param> ];
    ORBMasterIDL::OID oid;
    ORBMasterIDL::ReturnCode rc;
};
```

where:

valid indicates whether the other fields in the structure (except rc which is always valid) are valid;

<out param> is the value returned that is associated with the object identified by oid (only exists if <base operation> has an out parameter);

oid is the OID of the object for which the structure holds the result; and

rc represents the exception that would have been returned by the component object operation.

2. A typedef defining a sequence of *<interface><base operation>Results:*

```
typedef sequence
<
    <interface><base operation>Result
> <interface><base operation>ResultList;
```

3. An operation (defined on the ServiceManager interface):

```
void <interface><base operation>Group (
```

```
        in ORBMasterIDL::OIDList oids,
        [<in arg list>, ]
        out <interface><base operation>ResultList results
    );
```

where:

oids defines the scope of the operation

<in arg list> is the optional list of in parameters supported by *<base operation>*

results is the list of results obtained for the operation.

The following is an example of a component object operation getName for which the developer requires the system development tool to generate a group operation:

```
module Example {
    interface X {
        void getName ( out string name);
    };
};
```

For this example the system development tool generates the additional IDL shown below:

```
module Example {
    interface X {
        void getName ( out string name);
    };
    //
    // Start ORBMaster generated IDL
    //
    struct XgetNameResult {
        boolean valid;
        string name;
        ORBMasterIDL::OID oid;
        ORBMasterIDL::ReturnCode rc;
    };
    typedef sequence< XgetNameResult >
        XgetNameResultList;
    interface XManager {
        XgetNameGroup (
            in ORBMasterIDL::OIDList oids,
            out XgetNameResultList results);
        };
    //
    // END ORBMaster generated
    //
};
```

The C++ CAL and SAL interface classes which encapsulate the IDL interfaces defined above are described in the example provided in the section on 'Access Layers', below. The operations have an implicitly defined scope are defined by the developer and identified to the system development tool as implicitly scoped group operations.

Group operations may also support partial completion. By this is meant that

the service which implements the operation will make a best effort to apply the operation to all objects in the scope. Those that are available to the service will have the operation applied to them. It would be desirable for the clients to be informed when an object in the scope could not be contacted. In the case of implicitly scoped group operations, this may not be possible because the service may partition the component objects and it may not be able to determine which objects are actually in the scope (say if some partitions are unavailable when the operation is executed). For explicitly scoped group operations the system development tool requires that services return an *ObjectUnavailable* return code associated with each object that was not available. For implicitly scoped group operations the system development tool requires that services return a parameter of type *ORBMasterIDL::ResultAuthority* along with the result.

The delegation methods include those set out briefly below:

- none, which applies only to operations on services based on replicated component objects;
- directed, which applies only to operations on services based on partitioned component objects and to explicitly scoped group operations; and
- broadcast, which applies only to operations on services based on partitioned component objects and applies to both implicitly and explicitly scoped group operations. It is anticipated that transactional operations will be supported by other embodiments of ORBMaster.

• Management of Client/Server Interactions (Result Iterators)

Group operations that are queries typically return a list of individual "results" to their clients. If these queries are implemented using an IDL operation per query the following will result:

- the query client must allocate the memory required to receive all the "results" before any "result" becomes available; and
- the query client cannot start acting on some of the "results" until the whole operation is complete. Result Iterators are a standard design pattern which are used to solve these problems. This design pattern consists of a standard form for the IDL which defines these interfaces, a templated class, *ORBMaster::ResultIterator<class T>*, which forms part of the ORBMaster Support Library, and ORBMaster generated agent classes which use the templated class.

The system development tool ResultIterator design pattern uses three IDL operations to implement each query:

1. an operation which returns the first batch of results;
2. an operation which returns the next batch of results; and
3. a cancel operation which allows the client to notify his lack of interest in the query.

A batch of results is a set that contains no more than a client specified number of individual results. The server determines the actual number of results in the batch as described below:

- The operation which returns the first batch blocks until either:
  (i) at least one result is available; or
  (ii) the server determines that all available data has been searched;
  and then returns those results that are available (but no more than the client specified number). The server outputs a *resultId* that the client uses as the identifier for the query. It also outputs a boolean *isLast* which is set to true when all available data has been searched by the server.

The operation which returns the next batch behaves similarly except that it takes the *resultId* as input. Once a result has been returned to the client it is discarded by the server so that each result is only ever returned once. The cancel operation is essentially a notification to the server that the client has no more interest in the query. It has no semantics other than this. The server may use this notification to release resources that it has allocated to the query execution.

The following example shows a query, getAll, which returns a very long list of strings:

```
typedef sequence< string > ResultList;
        interface X {
        //
        //an operation which can return a very
        //large list of strings as its result:
        //
        void getAll(out ResultList veryLongList);
};
```

In order to make use of result iterators this query is implemented by the following operations:

```
module Example { interface X {
        void getAll(
                in short batchSize,
                out ResultList firstBatch,
                out string resultId,
                out boolean isLast);
        void getAll_next(
                in string resultId,
                in short batchSize,
                out ResultList nextBatch,
                out boolean isLast);
        void cancel(
                in string resultId);
        };
};
```

- Life-cycle management

Some services allow clients to create component objects and some do not. If clients can create component objects then, when the object is created, it must already have all

its read-only attributes defined. This means that the create operation must either take, as input, values for all those attributes. or the service must generate them. Typically the only read-only attribute that is generated by the service is the OID. Therefore the OID is either specified by the client as an input parameter to the create operation or it is generated by the service as part of the implementation of the create operation.

Access Layers

The Client Access Layer (CAL) separates application code from code needed to access the objects that implement the services. The interface between the CAL and the application code does not expose any of the C++ classes generated from the IDL for the service. Similarly, the ORBMaster Server Access Layer (SAL) separates the application code in the server from the IDL generated code used to access it. Further details of the CAL and SAL of the preferred embodiment are set out below.

Client Access Layer (CAL)

The classes that provide an interface between client application code and the CAL consist of:

- agent classes used to access the distributed objects which implement a service; and
- classes which represent the data structures manipulated by the agent classes. Agent classes serve two main purposes to separate IDL generated code from application code and to provide a complete encapsulation of the object, ie. one which encapsulates the identity of the object, its interface, and the means to address its implementation(s). In effect, the agent classes and the supporting data structure classes, are an alternative C++ mapping for IDL. The next sub-section explains the justification for introducing a non-standard alternate C++ IDL mapping.

• Justification for providing an alternate C++ binding for IDL

There are two main justifications for the provision of an alternate C++ IDL mapping:

1. Developers require a large amount of training before they are proficient developers of CORBA services and the client code that accesses them:

(a) The classes which define the interfaces to the CAL and SAL (unlike the standard C++ IDL mappings) use the standard template library of ANSI C++. This means that C++ proficient developers already understand the data structures that are a major part of the interface.

(b) The standard IDL mappings present more than one way of doing a task (eg. there is more than one model for memory management), by providing a simpler interface with less options, the SAL and CAL interface classes are consequently easier to use.

(c) The standard IDL mappings expose more complex concepts to users (eg. the standard object reference is a much more complex concept than the system development tool's agent classes).

2. Standard C++ mappings for IDL do not address object identity and object mobility.

(a) CORBA object references do not allow comparison for equality.

(b) CORBA object references may change (eg. if an object is moved from being implemented by one server to another).

(c) CORBA object references are unsuitable for use as database keys for the persistent storage of objects since they are too big.

- Alternate IDL bindings

The alternate client side mappings for the IDL are developed in accordance with a set of mapping rules. These rules are typically as follows:

1. For each IDL interface, a C++ agent class is provided. The name of the C++ agent class is: *<IDL module>_<IDL interface>Agent.*

2. Agent class inheritance (public virtual) mirrors the inheritance of the IDL interfaces, for example:

```
//IDL module
    module Example {
        interface X {
            void oppr_x();
        };
        interface Y : X {
            void oppr_y();
        };
    };
    //C++ agent Classes
    class Example_XAgent :
        public virtual ORBMaster_Agent {
        //details of class omitted
    };
    class Example_YAgent :
        public virtual Example_XAgent {
        //details of class omitted
    };
```

3. Agent classes contain a method for each IDL defined operation for the corresponding interface. All these operations return OVErrors and use STL data types rather than data types generated by the IDL compiler.

4. Agent classes provide assignment operators and copy constructors that have the same semantics as those for *ORBMaster_Agent.*

5. IDL defined structs are represented as C++ classes. The naming convention for these classes is (depending on the scope of the IDL struct):

<IDL module>_<IDL interface>_<Structure name>; or

<IDL module>_<Structure name>.

6. Classes that represent IDL constructs manage their own memory and have constructors that provide the initial values for all their data members. On construction they

take copies of all the provided data members. They support assignment operators and copy constructors that result in deep copies of the source objects.

- Agent Classes

Instances of these agent classes represent the distributed objects from the point of view of a client of those objects. Agent objects always contain the OID of the distributed object that they represent. For the case of agents for component objects this is the OID of the component object, for the case agents for ServiceManager objects this is the name of the service. When the agent is used to access the object, the CORBA object reference for the object must also be present. Agent classes obtain these object references when required. Instances of agent classes are bound to distributed objects by the following methods:

- binding them to strings which represent OIDs;
- assigning one agent to another, following assignment both agents now represent the same object;
- constructing one agent using another as source, following construction both agents now represent the same object;
- binding them to CORBA object references (only available to ORBMaster code within the CAL and SAL- not to application code).

If the CORBA object reference is not present when it is required, the agent resolves the OID to the appropriate CORBA object reference automatically. The agent classes also intercept system exceptions, and, when they occur, re-resolve OIDs to CORBA object references. This allows the same agent to be used to access different object implementations without intervention by the client of the object. This is useful in the following example cases:

- an object is moved by an administrator to balance load;
- one replicated copy of an object fails and is automatically substituted for by another.

An error is returned to the client only if a successful automatic resolution is not possible.

The purpose of the agent classes is to provide a clear separation between CORBA dependent code and application code. As a consequence of this separation, no CORBA header files or header files generated by the IDL compiler are included in the header files for agent classes. Implementations of agent classes (which obviously do depend on ORB code) are obscured by having each agent class include a private data member which is a pointer to an instance of a hidden access layer class (the access layer class header file is not provided for general use but is available only for internal use within the Client and Server Access Layers). The following class is provided as part of the ORBMaster Support Library of the embodiment to support the agent paradigm *ORBMaster_Agent*, which serves two purposes:

- a base class which is specialized by agent classes for specific objects implemented by ORBMaster servers, ie. it encapsulates the commonality of all agent classes; and

- can be used to represent any object implemented by an ORBMaster server.

This class is accessed by the developer and a man page is included for it, as follows:

```
#include <std/string>
#include <ORBMaster.hh>
#include <OVError.hh>

class ORBMaster_AgentAL;
class ORBMaster_Agent {
        friend class ORBMaster_AgentAL;
public:
ORBMaster_Agent();
ORBMaster_Agent(const ORBMaster_Agent& src);
virtual ~ORBMaster_Agent();
ORBMaster_Agent& operator=(
        const ORBMaster_Agent& rhs );
virtual OVError getOID(
ORBMaster_StructuredName & theOID)
        const; virtual OVError getOR(string& iorStr) const;
virtual void bind(const ORBMaster_StructuredName& oid);
virtual OVError exists(bool &result) const;
protected:
ORBMaster_AgentAL *impl;
ORBMaster_StructuredName oid; }
```

The system development tool also generates (from the IDL) the specific agent classes that inherit from *ORBMaster_Agent*. The system development tool generates these agent classes together with their complete implementation. Like the *ORBMaster_Agent* classes, the specific agent classes contain hidden access layer objects. These similarly contain CORBA object references. Also, like the *ORBMaster_Agent* class, these specific agent classes allow developers to bind the agents to specific CORBA objects using copy constructors, assignment operators and the bind method. Note: Because CORBA object references are not enough to identify the objects which support the system development tool services, agents cannot usually be bound to CORBA object references.

Server Access Layer (SAL)

There are two basic assumptions behind the implementation of the system development tool servers:

1. Component objects are always persistently stored by some object storage service. The object storage service may be a relational, or OO database or it may be an external application (as in the case of a CORBA gateway to a legacy system, or to network devices accessed via a network management protocol); and

2. ServiceManagers know the partitioning of component objects, that is, for a given

component object they know which ServiceManager is authoritative for it, but they delegate to the object storage service the knowledge as to whether a given object actually exists or not.

The object storage service provides an applications program interface (API) which, given an OID, either locates the state of the component object or indicates that the OID does not represent a valid component object.

If the service being implemented supports the concept of objects being created and deleted then the object storage service provides an API that does this lifecycle management.

Note: Although the term "object storage service" is used, the actual service may do much more than just store the objects persistently, for example it may be a complete application. The point is that it should preferably at least: manage lifecycle, answer object existence queries and store the objects persistently.

The above assumptions then allow the following:

- a distributed service to be built using a non distributed object storage service; allow the service to scale up to the scalability limit of the object storage service and not be limited by the ORB;
- the friend relationship which exists between ServiceManagers and the component objects for which they are authoritative to be exploited by allowing the internal state of the component objects to be accessed by the ServiceManager;
- avoid the need for ServiceManagers to separately keep track of which objects exist thereby removing the need for them to synchronize this information between themselves and the object storage service; and
- ServiceManagers to swap component object implementation objects into and out of memory at will because both the object's state and the knowledge of the object's existence is persistently stored by the object storage service.

The assumptions described above amount to requiring a very clean separation of responsibilities between the classes that encapsulate the SAL and the classes that encapsulate the service implementation. In fact this separation is so complete that the ORBMaster approach is essentially an application of the CORBA gateway server design pattern. This pattern is normally only applied to legacy system encapsulation, however the system development tool applies it to all servers because of the benefits it provides in decoupling ORB dependent code from application code. FIG. 17 shows the interactions between the server C++ objects in the server access layer (SAL) and the server application code.

The SAL includes a ServiceManager Adaptor 100 which manages access and memory for the ServiceManager Implementation 101 and contains the ORBMaster_ServerCache 102. The server cache controls the swapping of the Component Object Adaptors 103 listed in the LRU, which object adaptors in turn control access and

memory for the respective Component Object Implementations which are swapped in and out of the Object Storage Service 105. The Object Storage Service 105 may be an RDBMS, an OO DBMS or a legacy application, which service:

1. creates and deletes component objects;
2. answers queries regarding the existence of component objects; and
3. stores component objects.

- SAL / SAC Interface

Corresponding to every IDL defined interface are two classes:

- an *Impl* class which is provided by the system development tool only in the form of a stub; these exist within the Server Application Code as shown in FIG. 3;
- an *Adaptor* class which is derived from the IDL generated server stubs and is fully implemented by the system development tool; these exist within the Server Access Layer shown in FIG. 3.

Like the agent classes, the Impl classes support a method for each operation defined in the IDL interface. In fact, the name and signature of these methods are identical to those in the corresponding agent class. The suggested naming convention for Impl classes is:

<IDL module>_<IDL interface>Impl

Developers provide the actual implementation of these classes by building on the stub implementations generated by the system development tool. Developers can choose to implement these classes as stateless gateways to the object storage service or they may cache component state in them.

The infrastructure classes are derived from the IDL compiler generated server stubs. These classes provide the access path between the ORB and the developer provided implementations. The suggested naming convention for infrastructure classes is:

<IDL module>_<IDL interface>Adaptor

The server application code never accesses Impl class methods directly. All access to Impl objects is via the corresponding Adaptor object. This means that when an object implementation needs to access another object (even of the same class), it uses an agent object. Failure of developers to use agent classes in code that they supply may introduce problems relating to thread safety and memory addressing.

- Lifecycle and memory management

The memory management of each Impl object is handled by the infrastructure. That is, developers never directly construct or destruct a *<IDL module>_<IDL interface>Impl*, this is done by the corresponding *<IDL module>_<IDL interface>Adaptor*. Similarly, in the case of component objects, the infrastructure (ie. Adaptor classes) is responsible for initiating the swapping in and out of the component objects. The developer is only required to provide any service specific swap in or swap out implementation.

Clients create and delete component objects via operations on a ServiceManager. The create and delete operations used for component object lifecycle

management are not special to the system development tool. The implementation of these operations, like all other operations, involves an Adaptor object (in this case the Adaptor object for the ServiceManager) calling through to its corresponding ServiceManager Impl object. ServiceManager objects are created when the server process which instantiates them is started with the "install" command line option set. Similarly they are deleted when the server process which instantiates them is started with the "deinstall" command line option set. In other words, installation administrators explicitly mange the lifecycle of ServiceManager objects.

- Distribution issues

FIG. 17 shows how instances of the C++ classes in a server interact within a single process. The Adaptor classes for ServiceManagers have the responsibility for encapsulating distribution concepts. This means that the Impl classes for ServiceManagers are not concerned with issues of distribution, they simply implement the service on a single host. In other words all operations on Impl classes are implemented by only referring to the local object storage service. Adaptors provide the distributed view by delegation to the appropriate peer ServiceManagers (by using broadcast or directed delegation to the peer ServiceManagers as appropriate).

- C++ Classes

The following classes are provided as part of the Support Library to help implement ORBMaster servers:

- *ORBMaster_Component*
  an abstract base class for component object implementations;

```
class ORBMaster_Component
{
private:
        virtual OVError swap_in() = 0;
        virtual OVError swap_out(bool &refused) = 0;
};
```

- *ORBMaster_ServerCache*
  although part of the support library, developers do not access these objects directly.

The system development tool also generates (from the IDL) an Adaptor and a Impl class for every IDL interface. For the case of component objects the Impl class inherits from *ORBMaster_Component*. For each service, the system development tool also generates a file which defines a **main**() function. This function instantiates the Adaptor object that implements a ServiceManager for that service.

Common IDL

The following IDL defines common data types used by all services:

```
module ORBMaster {
    typedef OVErrorService::EntryId NonAuthorityReason;
```

typedef sequence< NonAuthorityReason >
NonAuthorityReasonList;
};

File Replication Service

The File Replication Service supports the replication of files within a CORBA installation. The File Replication Service of the embodiment depends on the following:

- the particular CORBA ORB;
- the ODBC Access Layer; and
- a POSIX compliant system interface. This limited dependency enables other services to be built on top of the File Replication Service without the complication of interdependencies. The File Replication Service is implemented using objects that support the standard IDL interfaces ReplicationManager and ReplicationClient. For each host in the installation there is at most one ReplicationManager object. A set of ReplicationManagers can be formed into a peer group. Within a peer group all ReplicationManagers keep object references to all other ReplicationManagers in the peer group.

The purpose of each ReplicationManager is to maintain a local copy of a set of files so that the contents of these local file copies are "approximately" synchronized across the peer group. Clients of the Replication Service define which files are to be replicated. ReplicationClient objects register themselves with ReplicationManager objects. On registration, ReplicationClients specify the files in which they have an interest. Files are specified using identifiers rather than file names allowing the local version of the file to have different names on different hosts. The service treats a group of files as a single entity in that the group is considered to be modified if any file in the group is modified and all files in the group are replicated when the group is modified.

- Assumptions

The File Replication Service supports the replication of files subject to the following restrictions (these typically express orders of magnitude rather then exact limits):

- 10 hosts per peer group;
- 10 files per host;
- 500 characters per file;
- file updates occur rarely (only a few changes per day);
- clocks on all hosts in a peer group are synchronized to within about 10 seconds;
- modifications to one file are propagated to other copies of the file using distributed operations; copies may therefore be out of date until changes propagate to them modifications are propagated to all hosts that are reachable by the host on which the modification was made within 1 minute of the change occurring;
- if a host looses connectivity with the source of a modification then it shall become aligned with it within 5 minutes of regaining connectivity.

- Interface Definition module

The following is an example of an interface definition module:

```
module ORBMasterFR
{
    typedef string FileName;
    typedef sequence<FileName> FileNameList;
    typedef string FileContents;
    typedef sequence<FileContents> FileContentsList;
    typedef string FileGroupId;
    typedef sequence< FileGroupId > FileGroupIdList;

    struct FileAccessProblem
    {
        string fileName;      //the file name
        string hostName;      //the host name
        string operation;     //the system call
                              //which causes the problem      string
errorMsg;     //the error message
    }

    exception FileAccessProblem {
        FileAccessProblem problem //Describes the file problem    };
    exception NoLocalBindings {};
    exception Uninitialized {};

    exception WrongNumberOfFiles {
        unsigned short actualNumber
    };

    // All clients of the Replication Service must support this
    // interface
    interface ReplicationClient
    {
    // indicates that the file group has changed
    void update(in FileGroupId modifiedFileGroup);
};

typedef unsigned long TimeStamp;
interface ReplicationManager
{
    // Define the local file names for a file group.
    // If names already exist for the group then replace them.
    void setNamesForGroup(
        in FileGroupId fileGroupId,
        in FileNameList localNames);

    // Get the local names for a file group.
    // Returns an empty list if no names exist for
```

```
// the given group.
FileNameList getNamesForGroup(
        in FileGroupId fileGroupId);

// Returns a list of all the clints registered.
// If no clients are registered then the list is empty.
ObjectIdList getClients();

// Returns a list of the file groups for a given client.
// Returns an empty list if the client is not registered.
FileGroupIdList getGroupsForClient(
        in ObjectId client);

// Get the version number for the specified file group.
// A value of zero indicates that the ReplicationManager
// does not have any local files for this group.
TimeStamp getVersion(
        in FileGroupId fileGroup);

// Receive a file group and, if the input group is more
// recent than the local group, update the local group and call
// the update operation on the registered clients.
// Updating the file group is atomic so that, if this
// operation returns successfully, the changes to the file
// system have been completed.
void upLoad(
        in FileGroupId fileGroupId,
        in FileContentsList fileGroupContents,
        in string source,       // host name of originator
        in TimeStamp version
)
raises (
        WrongNumberOfFiles,  // A registration exists for the
                             // file group but the number of
                             // elements in FileGroupContents
                             // is not the number of files
                             // in the group FileAccessProblem,
                             // A local file could not be written
);
// Registers a client's interest in a file group.
//Does nothing if the client is already registered
//for the group. Clients can be registered for
//multiple groups.
//The client is identified by clientId NOT clientObj.
void registerClient(
        in ObjectId clientId,
        in ReplicationClient clientObj,
        in FileGroupId fileGroupId)
raises (
```

```
        NoLocalBindings,    // There are no local file
                            // names defined for the file group
        Uninitialized       // clients cannot be registered
                            // until the ReplicationManager
                            // is initialized
    );
    //Removes an existing registrations for this client.
    //Does nothing if the registration does not exist.
    void unregisterClient(
        in ObjectId clientId,
        in FileGroupId fileGroupId );

    // reads well known file/s which stores the
    // object reference of all other ReplicationManagers in a
    // peer group; if the ReplicationManager has not yet been
    // initialized then this operation initializes it
    void loadPeers()
        raises {
            FileAccessProblem //Can't load the peers
        };
    };
};
```

- Defining a Peer Group

All ReplicationManagers in a peer group maintain their own local knowledge of all other ReplicationManagers in the peer group. They do this using CORBA object references. If the server process which instantiates a ReplicationManager is started in install mode it creates a persistent ReplicationManager and stores its CORBA object reference in a file (in a well known directory) with the same name as the host on which the server is executing. When the loadPeers operation is invoked, the complete set of CORBA object references for ReplicationManagers in the peer group must be stored as files in the well-known directory. It is an administrative task to ensure that the contents of the directory containing CORBA object references are identical on every host in the peer group. The loadPeers operation reads the files containing the CORBA object references in order to find out the all the ReplicationManagers in the peer group. The loadPeers operation stores the object references for the peers in the relational database. When ReplicationManagers are started, in other than install mode, the database is used to obtain the peers. This means that the ability for a ReplicationManager to start only depends on the availability of the local database.

The administrative procedure for initialising a peer group is then as follows:

- start all servers supporting ReplicationManagers in the peer group in install mode - each will write their CORBA object reference to a file with the local host name as the file name;
- ensure that each host has a complete set of CORBA object reference files (nothing

required here if the directory containing the CORBA object references is mounted using NFS);

- invoke the loadPeers operation on each ReplicationManager in the peer group.

Services which depend on the File Replication Service can only be started after the peer group is initialized.

- Managing Client Registrations

The registerClient operation on the ReplicationManager interface defines which file groups a particular client object is interested in. The unregisterClient undoes a registration. File groups are identified by strings (not file names). Client registrations are made persistent by ReplicationManagers. Clients are notified when a file group for which they are registered is modified, however, if a client is not contactable by the ReplicationManager when a file group is modified, then no attempt is made to inform the client when it subsequently becomes available. It is the responsibility of the client to obtain the latest copy of its files when it starts up to ensure that it aware of any changes which occurred while it was down.

- Source-push algorithm for file modification

The File Replication Service assumes that the files that it replicates can be modified directly via the file system. Each ReplicationManager polls the file system at regular (configurable) intervals in order to determine if a file group for which it has client registrations has been modified. Once a ReplicationManager has determined that a file group has been modified it pushes the modified file group out to all its peers by invoking the upLoad operation on them. It also notifies all its interested clients by invoking the update operation on them. When it invokes the upLoad operation it uses the most recent file modification time (in seconds since 00:00:00 GMT Jan 1 1970) for files in the group as the file group version number (the version parameter to the upLoad operation). It is the responsibility of the ReplicationManager that detected the modification (the source) to ensure that the new version is pushed out to all its peers. This means that it will retry the upLoad operation until it succeeds (even if it is stopped and subsequently restarted). The retry interval is configurable.

When the upLoad operation is invoked the target ReplicationManager checks whether the input file group is more recent than its local copy. It does this by comparing the input version parameter with its stored value. If these differ by more than the maximum error in system clocks (configurable) then the most recent is taken as the larger version number. Otherwise the version numbers are disregarded and the input host name and local host name are compared. The most recent is then taken as the one strcmp determines is the greater. Using this arbitrary ordering on host name means that if two or more different modifications occur at approximately the same time, only one modification will succeed. If the input is more recent then the ReplicationManager updates its local copy, informs its clients and stores the input version number as the version number for the group. If the input is not more

recent then the ReplicationManager just returns.

As well as version numbers, ReplicationManagers persistently store the time stamps associated with each local file. They do this so that they can detect when a file is modified. On start-up they determine if a file group has been modified while they were down by comparing the persistently stored time-stamps with the values obtained from the file system. By persistently storing these time-stamps they are able to treat the case of "file modification while they were down" as a normal file group modification as described above.

- Peer group start up

When a ReplicationManager is created its stored version numbers and file time stamps are set to zero. As client registrations occur, a ReplicationManager will detect that the stored time-stamps are less than the actual values and so push the local copies of files out to all the peers. The most efficient way to start up a peer group is to have only one copy in the peer group of each file group.

Service Finder Service

The Service Finder Service is implemented using objects that support the interfaces ServiceLocator and ServiceManager. Every host in the installation has at most one ServiceLocator. ServiceLocators manage a repository of ServiceManagers and allow clients to find ServiceManagers based on service name, proximity to the ServiceLocator, and location. The Server Finder Service depends on the following:

- the particular CORBA ORB specified;
- the ODBC Access Layer;
- a POSIX compliant system interface;
- the CORBA Notification Service

- Assumptions

The following assumptions apply to the embodiment:

- the installation consists of about 10 hosts
- the hosts in the installation are defined at installation time and not altered; and
- ServiceManagers may be dynamically added and removed from an installation.

- Interface Definition

```
module ORBMasterSF {
        typedef string ServiceName;
        typedef sequence < ServiceName > ServiceNameList;
        typedef string Location;
        typedef sequence < Location> LocationList;

        interface ServiceManager;
        typedef sequence <ServiceManager> ManagerList;
        interface ServiceLocator;
        typedef sequence <ServiceLocator> LocatorList;

        exception AlreadyRegistered {
```

```
};
exception NotRegistered {
};

interface ServiceManager :
        CosLifeCycle::FactoryFinder
{
        readonly attribute ServiceName serviceName;
        readonly attribute Location location;
};

interface ServiceLocator {

// Registers a ServiceManager, stores the tuple:
//      (serviceName, location, ServiceManager).
// ServiceManager is identified by its serviceName and
// location attributes.
void registerManager(
        in ServiceManager ServiceManagerObj)
raises (
        AlreadyRegistered,
                // a ServiceManager is already
                // registered with this ServiceLocator for the
                // service and location
        );

// used to distribute notifications about the existence
// of a ServiceManager
void notifyAddManager(
        in ServiceName serviceName,
        in Location location,
        in ServiceManager ServiceManagerObj);

// Removes an existing registration.
void unregisterManager(
        in ServiceName serviceName,
        in Location location)
raises(
        NotRegistered
);

void notifyRemoveManager(
        in ServiceName serviceName,
        in Location location);

// gets a locally registered ServiceManager for the
// specified service. If no matching ServiceManager exists
// returns a NIL object reference.
```

```
//If testRequired is true then the ServiceLocator tests the
//availability of ServiceManagers and does not consider
//unavailable ServiceManagers to exist.
ServiceManager getLocalManager(
        in ServiceName service,
        in bool testRequired);

// gets the best ServiceManager for the
// specified service.If no matching ServiceManager exists
// returns a NIL object reference.
// If testRequired is true then the ServiceLocator tests
// the availability of ServiceManagers and does not consider
//unavailable managers to exist.
ORBMaster::ResultStatus getBestManager(
        in ServiceName service,
        in bool testRequired,
        out ServiceManager Manager);

// gets locally registered ServiceManagers.
// The services parameter is used to filter the results
// based on service name.
// If the services list is empty then no filtering by service name
// applies.
// The locations parameter is used to filter the results
// based on location.
// If the locations list is empty then no filtering by location
// applies.
ManagerList getLocalRegisteredManagers(
        in ServiceNameList services,
        in LocationList locations);

// gets all registered ServiceManagers.
// The services parameter is used to filter the results
// based on service name.
// If the services list is empty then no filtering by service name
// applies.
// The locations parameter is used to filter the results
// based on location.
// If the locations list is empty then no filtering by location
```

```
        // applies.
        ORBMaster::ResultStatus getAllRegisteredManagers(
                in ServiceNameList services,
                in LocationList locations,
                out ManagerList Managers);
        };
};
```

- ServiceLocator

All ServiceLocators in a peer group maintain their own local knowledge of all other ServiceLocators in the peer group. They do this in the embodiment using CORBA object references. If the server process which instantiates a ServiceLocator is started in install mode it creates a persistent ServiceLocator and stores its CORBA object reference in a file (in a well known directory) with the same name as the host on which the server is executing.

When the loadPeers operation is invoked, the complete set of CORBA object references for ServiceLocators in the peer group must be stored as files in the well known directory. It is an administrative task to ensure that the contents of the directory containing CORBA object references are identical on every host in the peer group. The loadPeers operation reads the files containing the CORBA object references in order to find out the all the ServiceLocators in the peer group. The loadPeers operation stores the object references for the peers in the relational database. When ServiceLocators are started, in other than install mode, the database is used to obtain the peers. This means that the ability for a ServiceLocator to start only depends on the availability of the local database. A typical administrative procedure for initialising a peer group is then:

- start all servers supporting ServiceLocator in the peer group in install mode, each will write their CORBA object reference to a file with the local host name as the file name;
- ensure that each host has a complete set of CORBA object reference files (nothing required here if the directory containing the CORBA object references is mounted using NFS); and
- invoke the loadPeers operation on each ServiceLocator in the peer group.

Each ServiceLocater maintains a persistent list of the ServiceManagers that it has registered. It stores this list in the relational database using the ODBC access layer. A ServiceLocator does not store the registrations that are managed by its peer ServiceLocators. Clients make requests for ServiceManagers for specific services by invoking the getBestManager on any ServiceLocator. There are typically two circumstances in which a client will request a ServiceManager:

1. the client is binding an agent in order to communicate with a ServiceManager the first time; or
2. the client has just got a system exception when attempting to communicate with a

ServiceManager.

The service is designed on the assumption that most ServiceManagers are available when required (ie. case 1 above is the usual case). This means that the service does not check the availability of ServiceManagers unless the client explicitly requests that it do so. Clients will only request a check when they have reason to think that a ServiceManager has become unavailable (ie case 2 above). There would be little point in using a cache when responding to requests for ServiceManagers if all ServiceManagers were checked for availability. However, since there is typically no availability check made, a ServiceLocator can use a cache to good effect when responding to requests. Each ServiceLocator therefore maintains a local in-memory cache which stores tuples: (service, name, ServiceManager) obtained from the results of previous attempts to locate ServiceManagers. The tuple stored in the cache corresponds to the first response to a broadcast request for managers of the service. Clients set the parameter testRequired to true when they have had a communications failure else they set it to false. Processing a getBestManager request then depends on the value of the testRequired parameter as described below.

When the getBestManager operation is invoked, the following steps are used to determine which ServiceManager to return:

- If there is a ServiceManager for the specified service registered with the ServiceLocator then return it, otherwise;
- If there is an entry in the cache for the specified service then return it, otherwise;
- Broadcast a getLocalManager request to all peer ServiceLocators. The testRequired parameter is set to true for these operations because the extra time to contact the ServiceManager is not very significant given that a remote broadcast is being done. The first response obtained is placed in the cache and returned to the client.

When the getBestManager operation is invoked, the following steps are used to determine which ServiceManager to return:

- If there is a ServiceManager for the specified service registered with the ServiceLocator then attempt to communicate with it (say get its location attribute).
- If it can be contacted then return it, otherwise:
- If there is an entry in the cache for the specified service then attempt to communicate with it. If it can be contacted then return it otherwise invalidate the cache entry and:
- Broadcast a getLocalManager request to all peer ServiceLocators. The testRequired parameter is set to true for these operations. The first response obtained is placed in the cache and returned to the client.

• ServiceManager

Defining a peer group can be considered as follows. The members of a peer group of ServiceManagers can be obtained by invoking the getAllRegisteredManagers operation on any ServiceLocator, specifying the appropriate server name. This operation uses a broadcast to locate the ServiceManagers and is therefore a potentially expensive operation.

It is typically members of a particular peer group themselves who need to determine the other peers in the group. Therefore, as an alternative to the expensive getAllRegisteredManagers operation, ServiceLocators enable ServiceManagers to store their own peer groups. They do this by sending notifications whenever members join or leave the peer group.

If ServiceManagers wish to maintain their own peer group membership lists then they need to register as notification receivers with their local ServiceLocator as notification producer. ServiceLocators distribute the knowledge of when members join and leave the groups using the notifyAddManager and notifyRemoveManager operations, as shown diagrammatically in FIG. 18. The steps in the distribution are as follows:

1. ServiceManager for location X 110 registers the location with the local ServiceLocator 111;
2. The local ServiceLocator 111 invokes a *notifyAddManager* process on each of its peer ServiceLocators 112, 113 and 114;
3. ServiceLocators sends notification containing the ServiceManager's object reference to the notification service 115, which notification is received by peer ServiceManagers 116 and 117.

ServiceManagers can then build a cache of their peers using *getAllRegisteredManagers* to get initial contents and using notifications to maintain it.

Summary

It will be appreciated from the foregoing that the development tool of the invention provides the following benefits for developers of large scale object oriented systems:

(a) reduced training costs for development team members - few have to be CORBA literate, fewer still need to be CORBA expert;

(b) design time is reduced, because the development tool includes packaged usage models to match particular needs;

(c) coding time is reduced because the code generator and libraries provide proven reliable modules, and the tool presents a simplified interface to the client and server application code; and

(d) test time is reduced because developers write less of the code, the code they do write is less complex and they are guided away from erroneous usage.

The system development method and tool of the invention allows developers to focus on providing the functionality of an application, rather than on the distributed, object oriented infrastructure required to deliver the applications services. It is also important to understand that the system development tool and method of the invention may be applied with substantially equal benefits to distributed object oriented system architectures other than OMG's CORBA. The system development tool is suitable for adding distribution to an existing computer application to extend its performance and scalability or for transparently

federating systems to provide unified access methods. The system development tool is particularly suited to developing new systems for distributed telecommunications or financial services.

Throughout the specification the aim has been to describe preferred embodiments of the invention rather than limiting the invention to one particular embodiment or specific collection of features.

CLAIMS

1. A development tool for building a large scale distributed object oriented computer system, which system includes a plurality of clients, a plurality of servers, and a distributed object infrastructure for communicating client requests for services to servers, said development tool comprising:

(a) a series of templates providing predetermined object design patterns, including -

(i) an object identity pattern, facilitating unique identification of each object,

(ii) a collection pattern, facilitating the logical grouping of objects having some commonality,

(iii) a group operation pattern, facilitating operations targeted at a set of objects,

(iv) a friend pattern, facilitating association of one object with another object independently of clients, and

(v) a partition pattern, facilitating physical grouping of objects for system performance purposes;

(b) a code generator arranged to generate, from an object oriented system model created by a user for defining desired server processes to be requested by client processes and incorporating selected ones of the design patterns, the following -

(i) a client access layer for each client process, isolating client application code from the distributed object infrastructure,

(ii) a server access layer for each server process, isolating server application code from the distributed object infrastructure, and

(iii) a stub portion of the server application code for implementing each service, including provision for the user to integrate an implementation of server semantics.

2. The development tool as claimed in claim 1 further comprising a set of basic distributed services including a service finder service for the discovery of the services available in the system.

3. The development tool as claimed in claim 1 or claim 2 wherein the series of templates may further include one or more of the following object design patterns:

(vi) a federation pattern, being a set of collections cooperating to provide an improved service;

(vii) a unified service pattern, facilitating the optimal choice of a collection from the set within a federation; and/or

(viii) a bulk operation pattern, facilitating multiple operations on a particularly identified object.

4. The development tool of any one of claims 1 to 3 wherein object identity is an attribute of an object and is represented using a structured name.

5. The development tool of claim 4 wherein the object identity attribute allows for object replication.

6. The development tool of any one of claims 1 to 5 wherein objects grouped into a collection are known as members and knowledge of a collection's members is kept either explicitly or implicitly.

7. The development tool of any one of claims 1 to 6 wherein a set of objects to which a group operation applies is known as the scope of the group operation, which scope may be explicitly or implicitly defined.

8. The development tool of claim 7 wherein an explicitly defined group operation is based on an underlying operation supported by objects comprising the service and a list of object identifiers defines the scope of the group operation.

9. The development tool of claim 7 wherein an implicitly defined group operation is not based on an underlying operation and a client specified filter or rule is applied to the objects to define the scope of the group operation.

10. The development tool of any one of claims 1 to 9 wherein two objects are friends if they do not appear associated to clients via the distributed object infrastructure, but the two objects appear associated to one another.

11. The development tool of any one of claims 1 to 10 wherein a partition is a physical grouping of objects, wherein each object in the system is associated with only one partition, which partition corresponds to a set of computer hardware.

12. The development tool of any one of claims 3 to 11 wherein collections in a federation are able to delegate operations to each other in order to provide a faster, more extensive or more reliable service.

13. The development tool of any one of claims 3 to 12 wherein a unified service is a federated collection wherein a predetermined sub-set of collections is transparent to clients requesting the unified service.

14. The development tool of any one of claims 1 to 13 wherein the client access layer includes agent classes to access objects which implement a service and other classes to represent data structures manipulated by the agent classes.

15. The development tool of claim 14 wherein the agent classes effectively separate interface code for the distributed object infrastructure from the client application code and encapsulate an object's identity, interface and means to address its implementation(s).

16. The development tool of any one of claims 1 to 15 wherein the server access layer includes service managers for managing objects with respect to any partitions and allows for the creation and deletion of objects.

17. The development tool of claim 16 wherein the server access layer includes adapter classes for providing access to objects which implement a service.

18. The development tool of any one of claims 2 to 17 wherein the set of basic distributed services further includes a file replication service for replicating files within the system.

19. The development tool of any one of claims 1 to 18 wherein the system utilizes the CORBA standard such that:

(a) the distributed object infrastructure comprises an object request broker (ORB);

(b) the object oriented system model is modeled using CORBA concepts; and

(c) the server interface is generated in accordance with CORBA interface design language (IDL).

20. A method for the development of a large scale distributed object oriented computer system, which system includes a plurality of clients, a plurality of servers, and a distributed object infrastructure for communicating client requests for services to servers, said development method including the steps of:

(a) selecting one or more templates, from a series of templates for predetermined object design patterns, which include -

(i) an object identity pattern, facilitating unique identification of each object,

(ii) a collection pattern, facilitating the logical grouping of objects having some commonality,

(iii) a group operation pattern, facilitating operations targeted at a set of objects,

(iv) a friend pattern, facilitating association of one object with another object independently of clients, and

(v) a partition pattern, facilitating physical grouping of objects for performance purposes;

(b) creating an object oriented system model for defining desired server processes to be requested by client processes, which model incorporates selected object design patterns; and

(c) generating, from the object oriented system model, code modules for the following -

(i) a client access layer for each client process, isolating client application code from the distributed object infrastructure,

(ii) a server access layer for each server process, isolating server application code from the distributed object infrastructure, and

(iv) a stub portion of the server application code for implementing each service, including provision for the user to integrate an implementation of server semantics.

21. The system development method as claimed in claim 20 further including the step of providing a set of basic distributed services including a service finder service for the discovery of the services available in the system.

22. The system development method as claimed in either claim 20 or claim 21 wherein the series of templates available for selection in step (c) may further include one or more of the following object design patterns:

(vi) a federation pattern, being a set of collections cooperating to provide an improved service,

(vii) a unified service pattern, facilitating the optimal choice of a collection from the set within a federation, and/or

(viii) a bulk operation pattern, facilitating multiple operations on a particularly identified object.

23. The system development method as claimed in any one of claims 20 to 22 including the step of representing the identity attribute of an object by using a structured name.

24. The system development method as claimed in claim 23 wherein the object identity attribute allows for object replication.

25. The system development method as claimed in any one of claims 20 to 24 including the step of referring to objects grouped into a collection as members and keeping knowledge of a collection's members either explicitly or implicitly.

26. The system development method as claimed in any one of claims 20 to 25 including the step of referring to a set of objects to which a group operation applies as the scope of the group operation, which scope may be explicitly or implicitly defined.

27. The system development method as claimed in any one of claims 20 to 26 including the step of arranging friend objects such that they do not appear associated to clients via the distributed object infrastructure, but they appear associated to one another.

28. The system development method as claimed in any one of claims 20 to 27 including the step of assigning a physical grouping of objects to a partition wherein each object in the system is associated with only one such partition, which partition corresponds to a set of computer hardware.

29. The system development method as claimed in any one of claims 22 to 28 including the step of allowing collections in a federation to delegate operations to one other in order to provide a faster, more extensive or more reliable service.

30. The system development method as claimed in any one of claims 22 to 29 including the step of arranging a predetermined sub-set of collections within a federated collection to be transparent to clients requesting unified service.

31. The system development method as claimed in any one of claims 20 to 30 including the step of step of generating a client access layer includes the step of generating agent classes to access the objects which implement a service and other classes to represent data structures manipulated by the agent classes.

32. The system development method as claimed in claim 31 wherein the step of generating agent classes includes the steps of separating interface code for the distributed object infrastructure from the client application code and encapsulating an object's identity, interface and providing means to address its implementation(s).

33. The system development method as claimed in any one of claims 20 to 32 wherein the step of generating a server access layer includes the provision of service managers for managing objects with respect to any partitions, which service manager facilitates the creation and deletion of objects.

34. The system development method as claimed in claim 33 wherein the step of

generating a server access layer includes the provision of adapter classes to provide access to objects that implement the service.

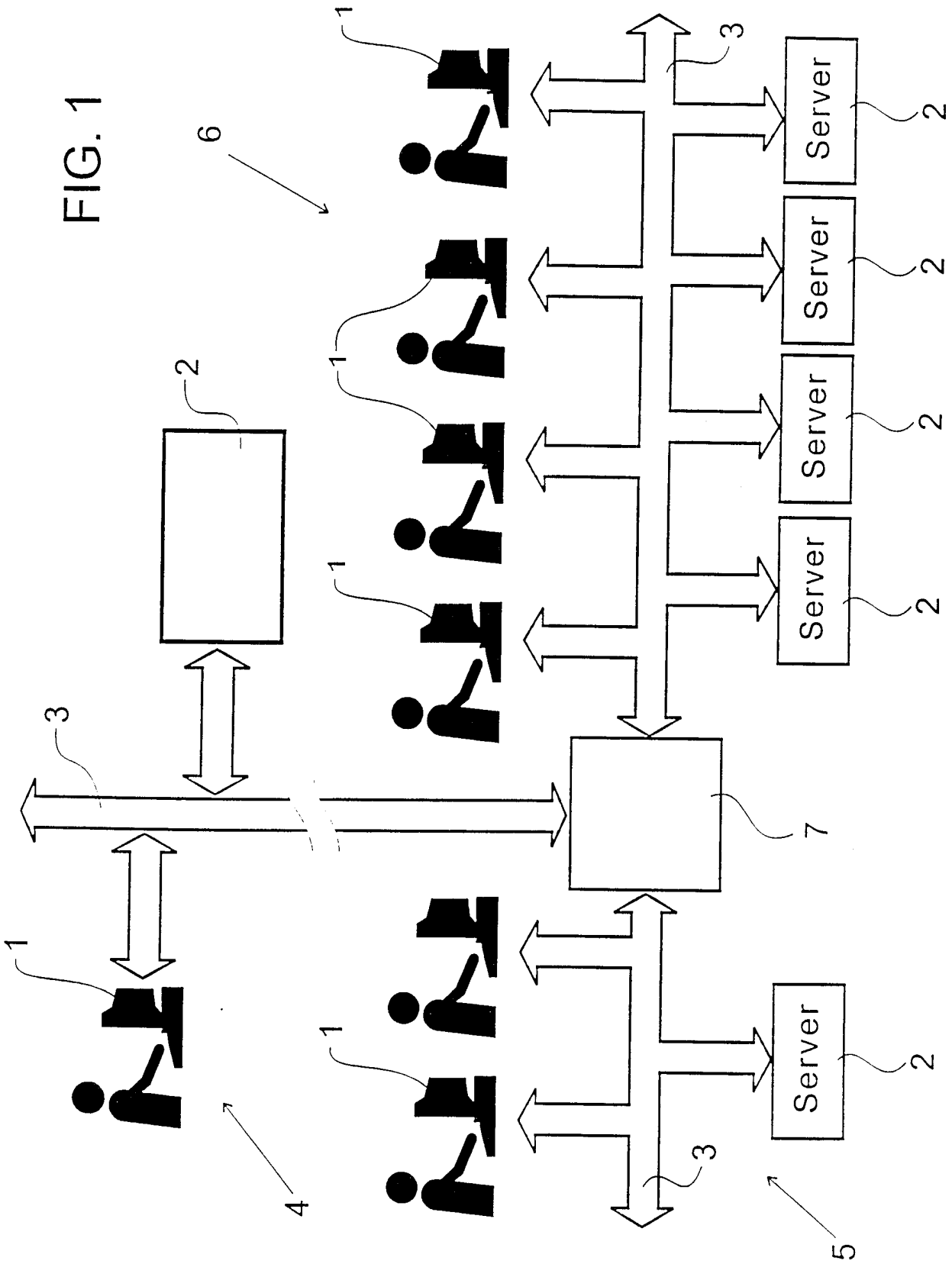
35. The system development method of any one of claims 21 to 34 wherein the step of providing a set of basic distributed services further includes providing a file replication service for replicating files within the system.

36. A large scale object oriented system built using the development tool of any one of claims 1 to 19 or according to the method of any one of claims 20 to 35, wherein the object oriented system includes a common administration interface.

37. The large scale object oriented system of claim 36 wherein the common administration interface facilitates remote management of all unified services in the system, including the provision of test, enable, disable, backup and restart functions.

38. The large scale object oriented system of claim 37 wherein the administration interface also supports a set of attributes for which each unified service may be queried, including one or more of version number, copyright information, status, host machine, process identity or like attributes.

FIG. 1



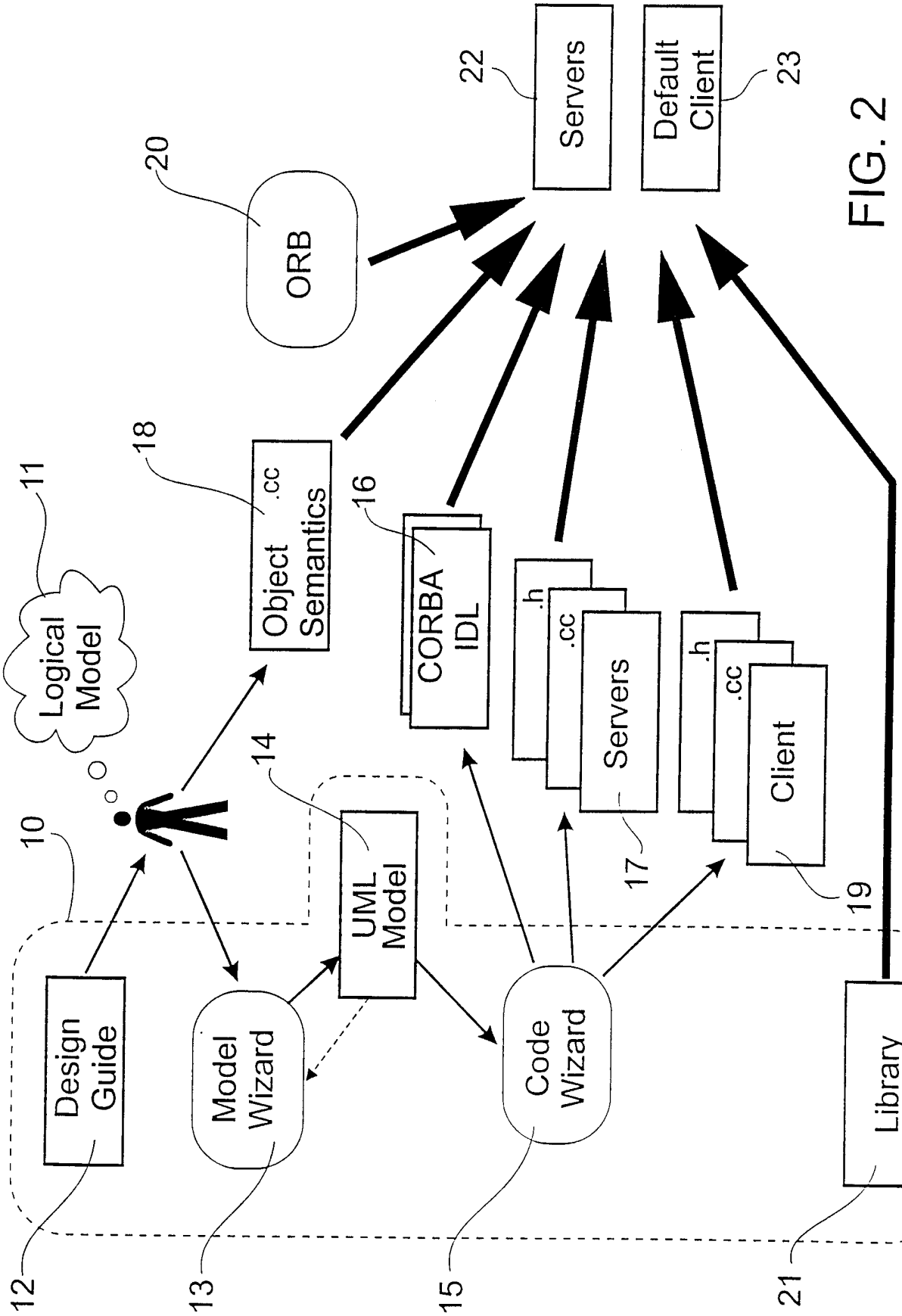

Servers
Default Client
ORB
Object Semantics .cc
CORBA IDL
.h
.cc
Servers
.h
.cc
Client
Logical Model
UML Model
Design Guide
Model Wizard
Code Wizard
Library
22
23
20
18
16
11
10
14
17
19
12
13
15
21

FIG. 2

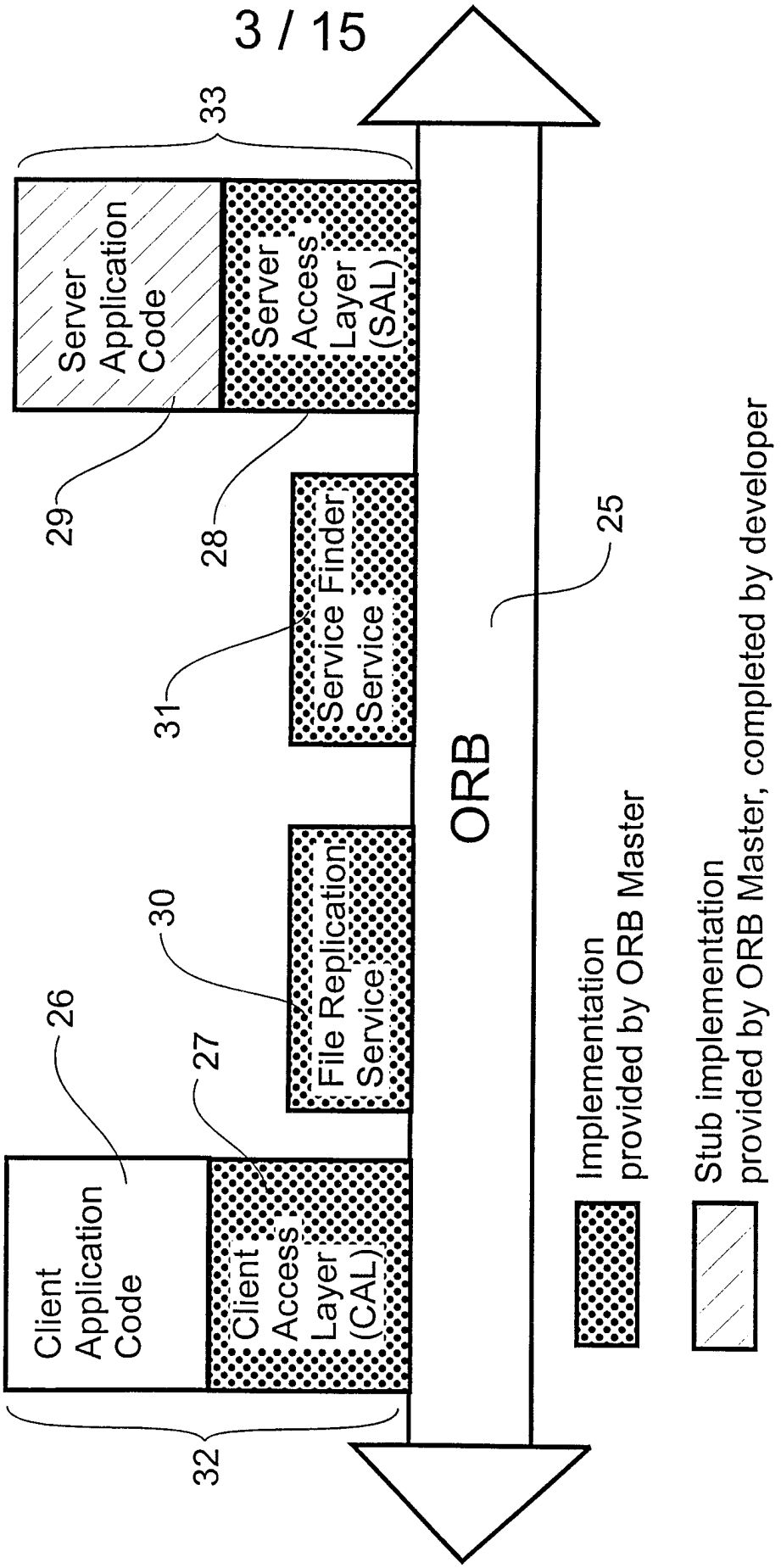

33
Server Application Code
Server Access Layer (SAL)
29
28
31
Service Finder Service
25
ORB
30
File Replication Service
26
27
Client Application Code
Client Access Layer (CAL)
32
Implementation provided by ORB Master
Stub implementation provided by ORB Master, completed by developer

FIG. 3

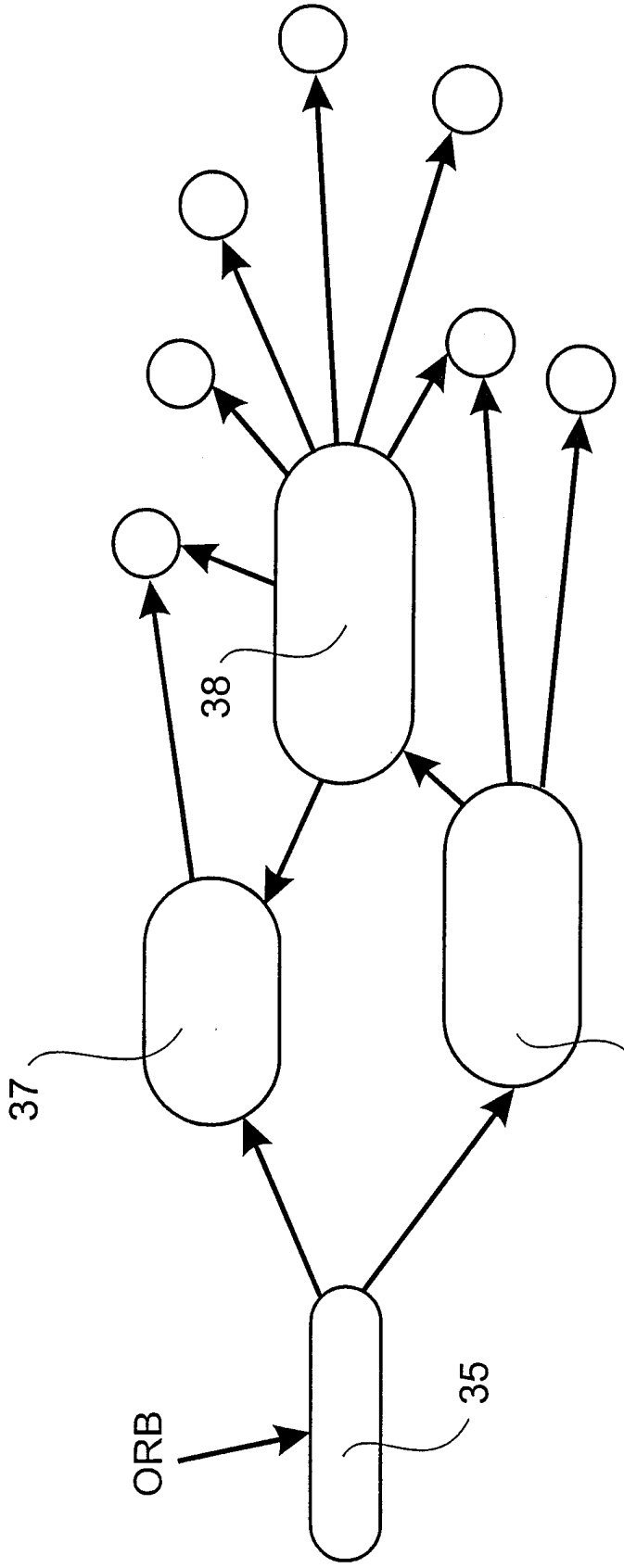


FIG. 4

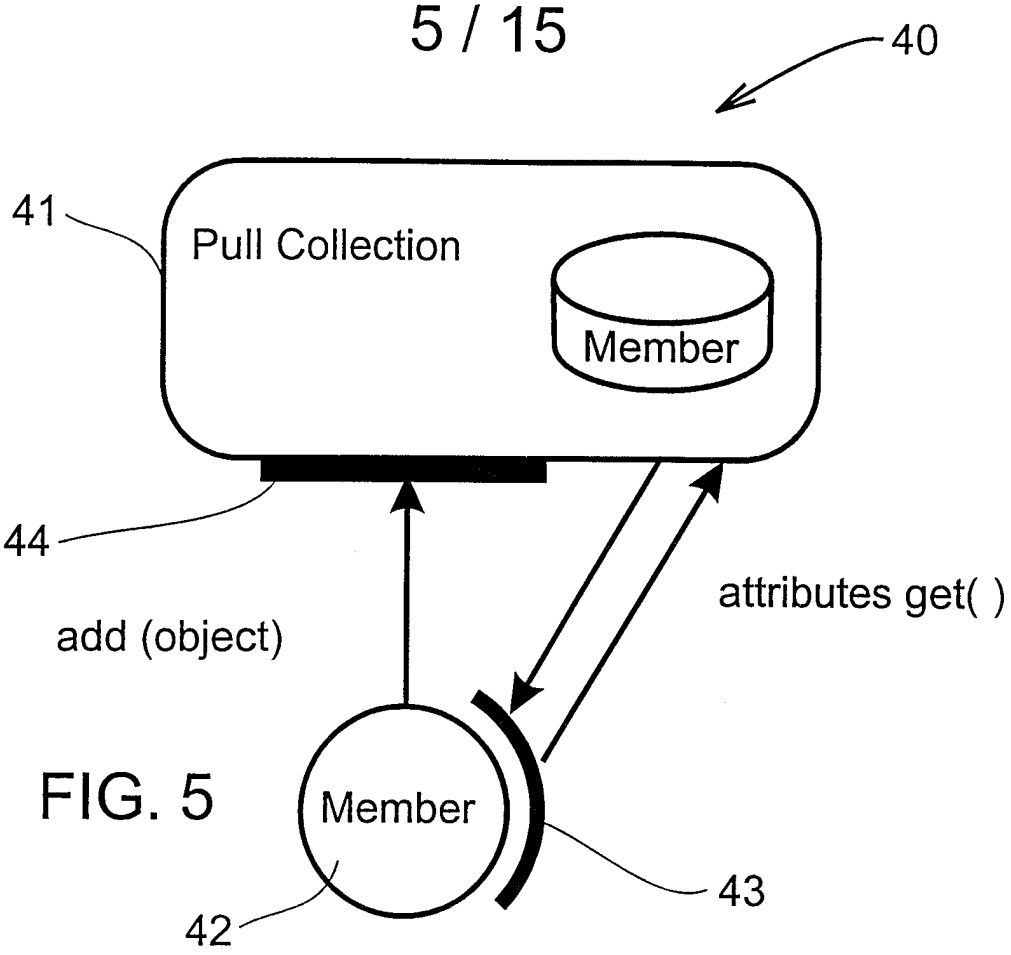

40
41
Pull Collection
Member
44
add (object)
attributes get( )
Member
42
43

FIG. 5

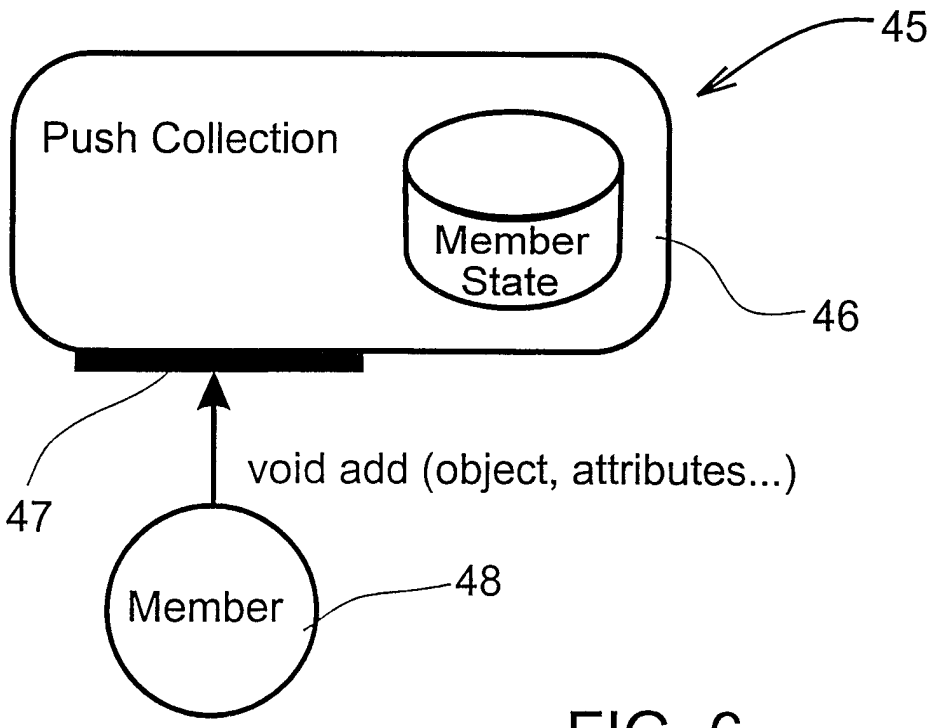

45
Push Collection
Member State
46
47
void add (object, attributes...)
Member
48

FIG. 6

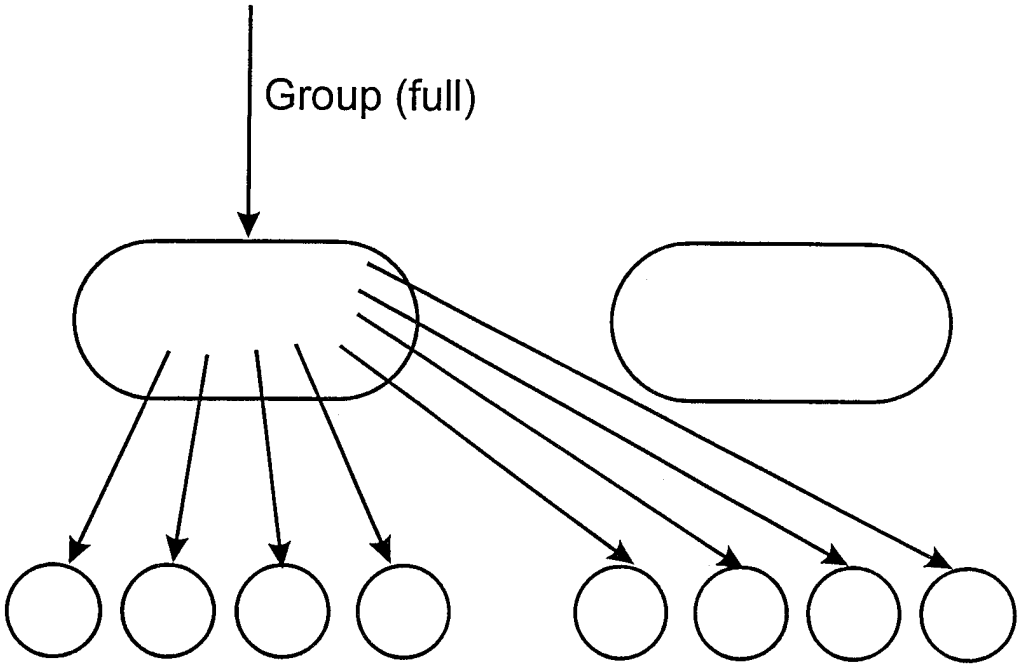


FIG. 7

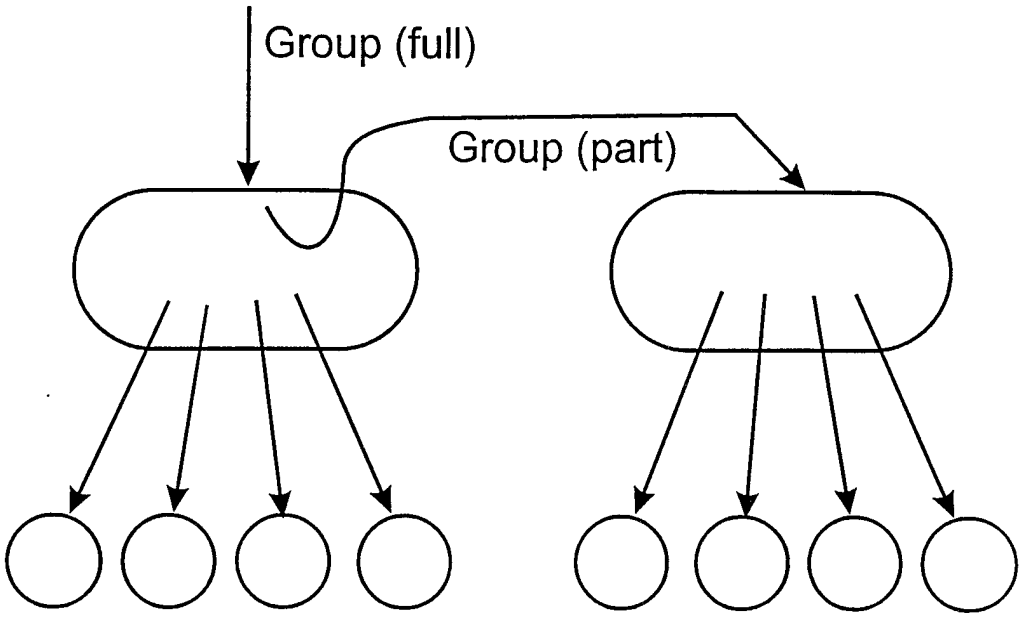


FIG. 8

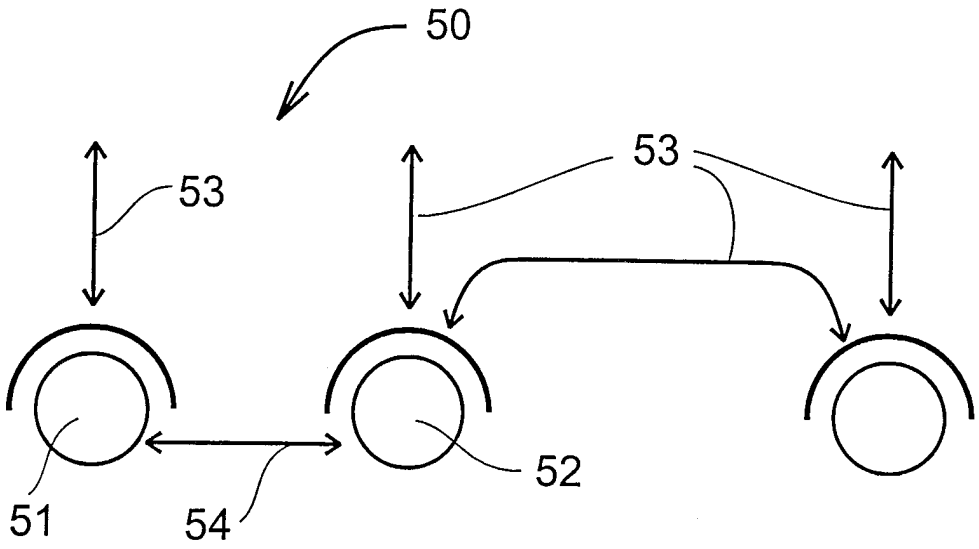


FIG. 9

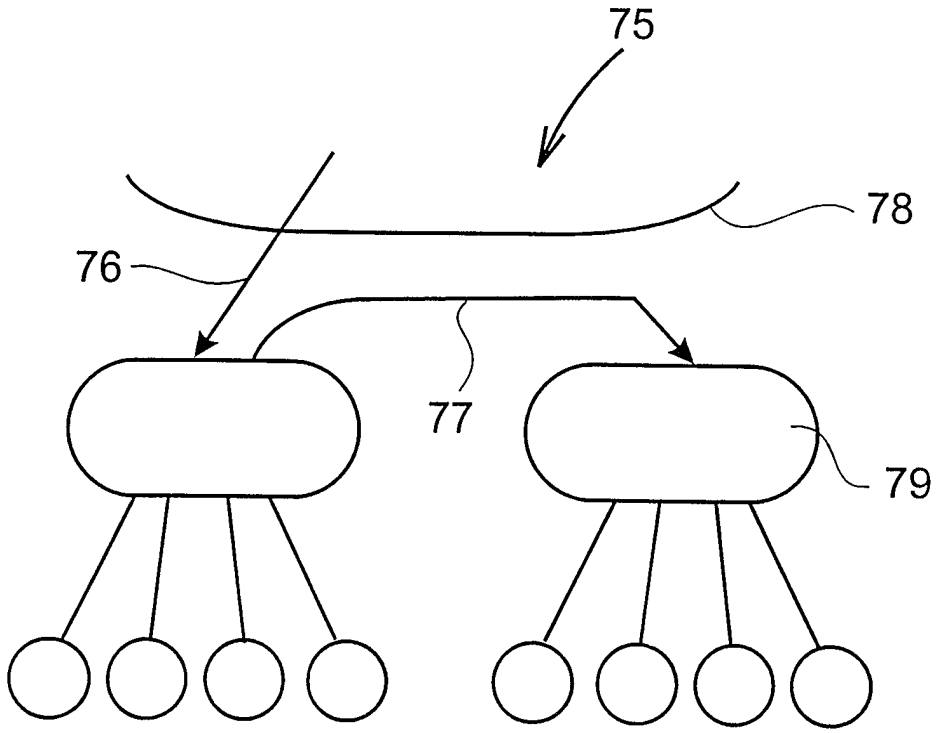


FIG. 14

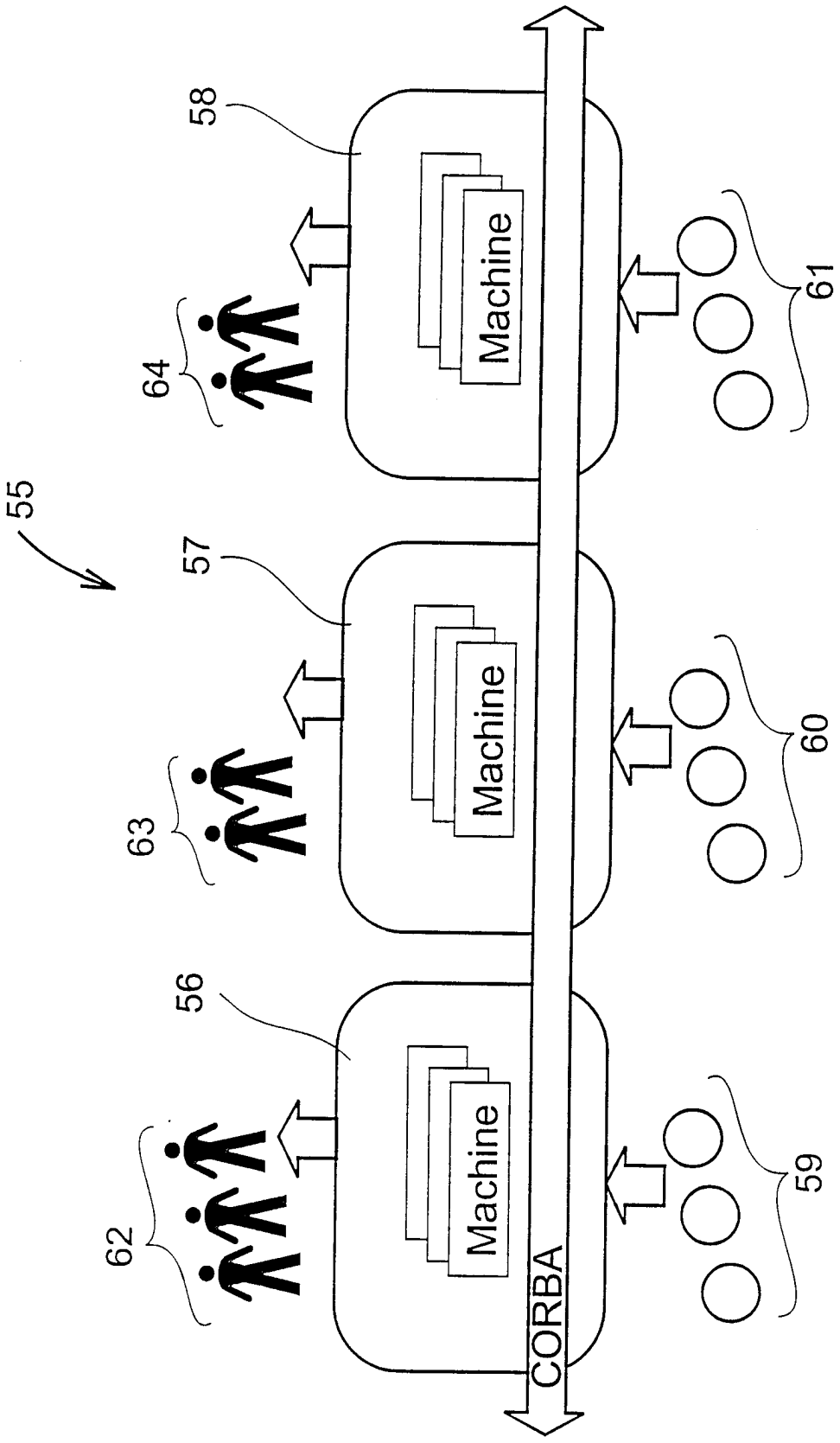


FIG. 10

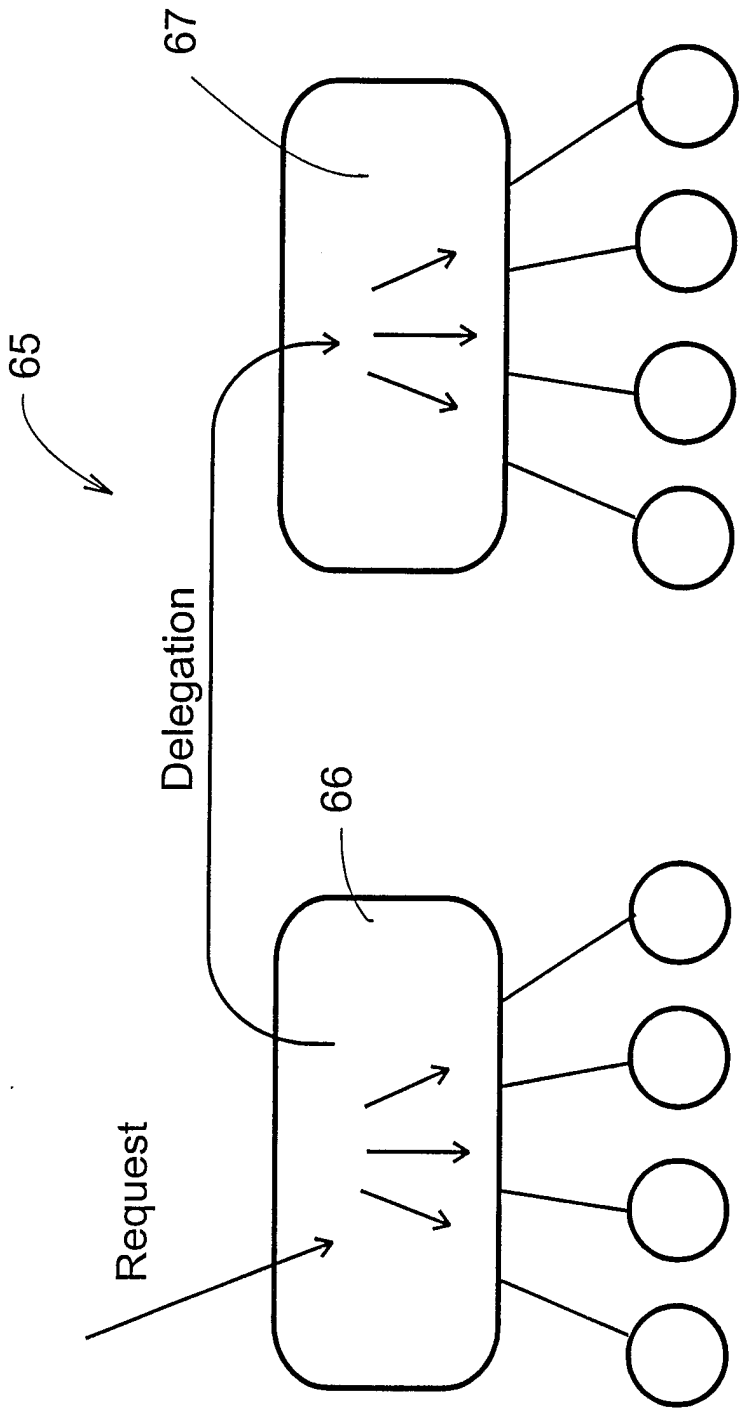


FIG. 11

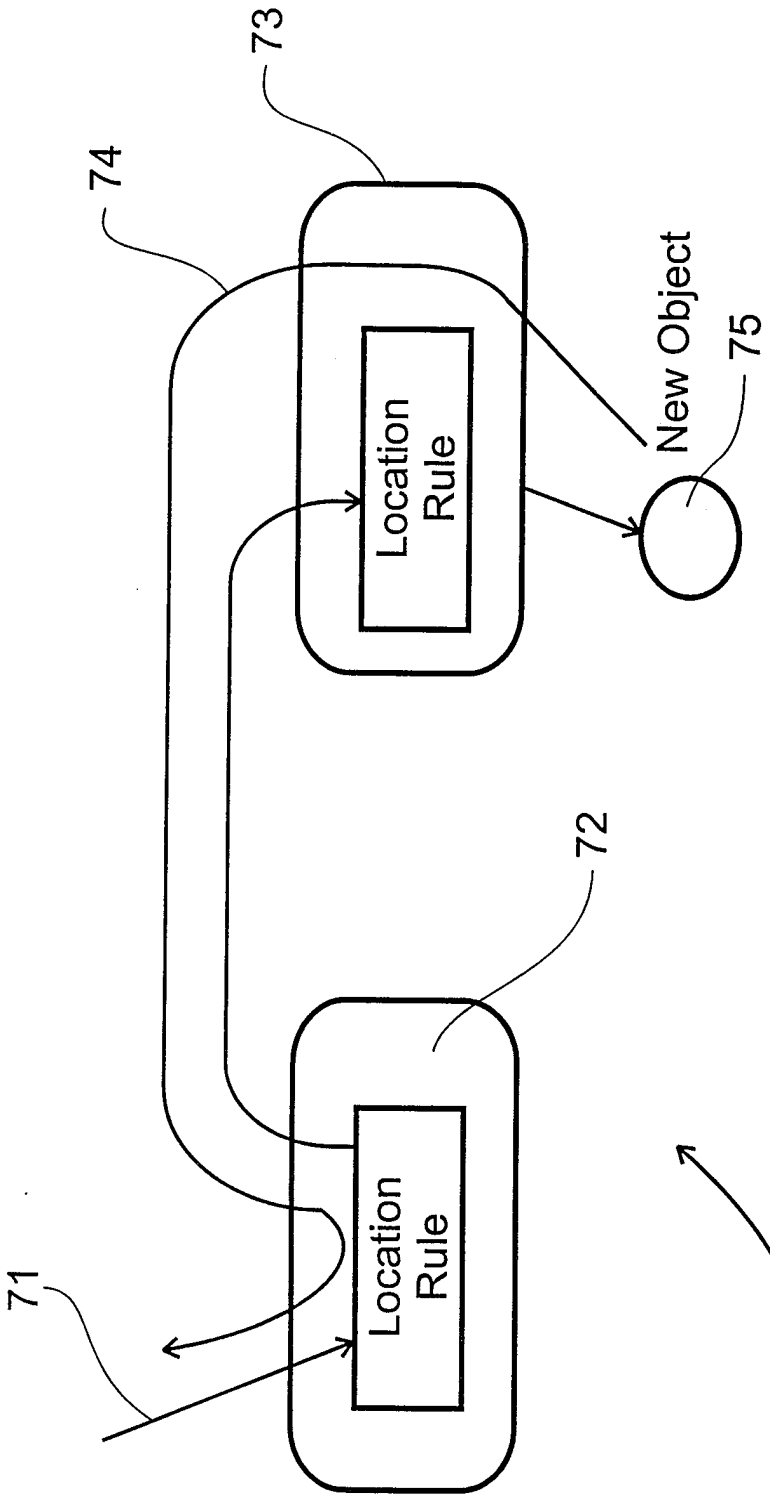


FIG. 12

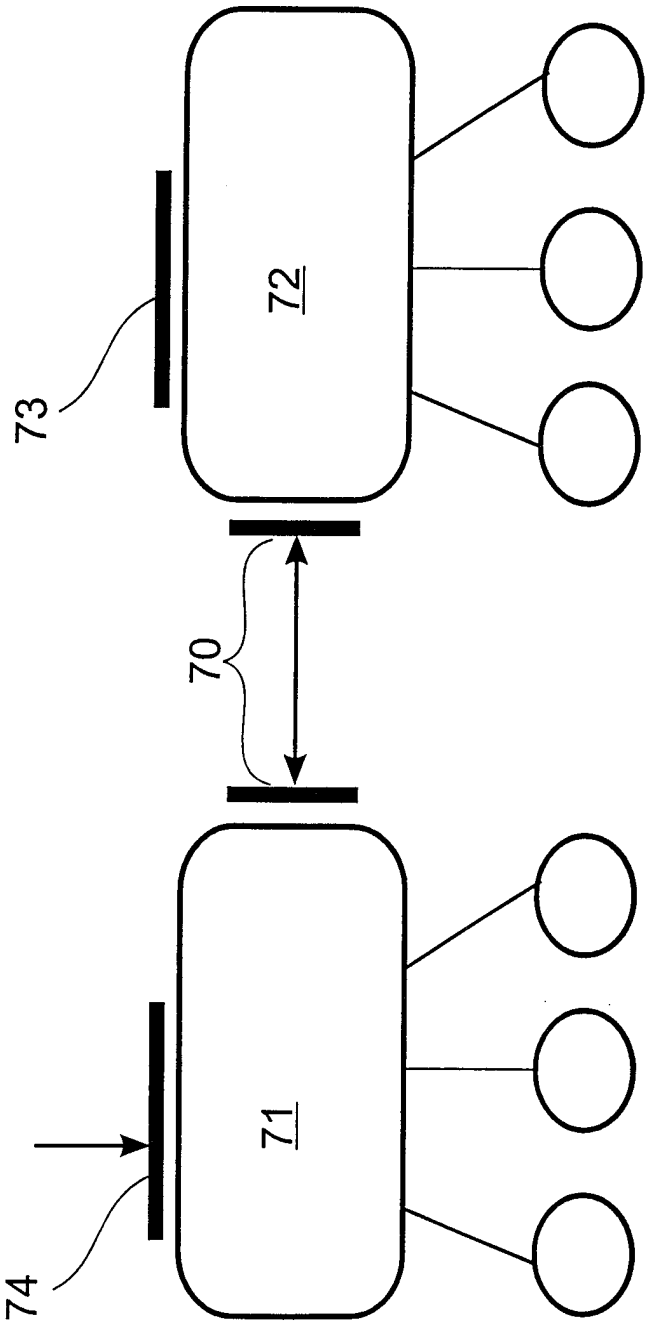


FIG. 13

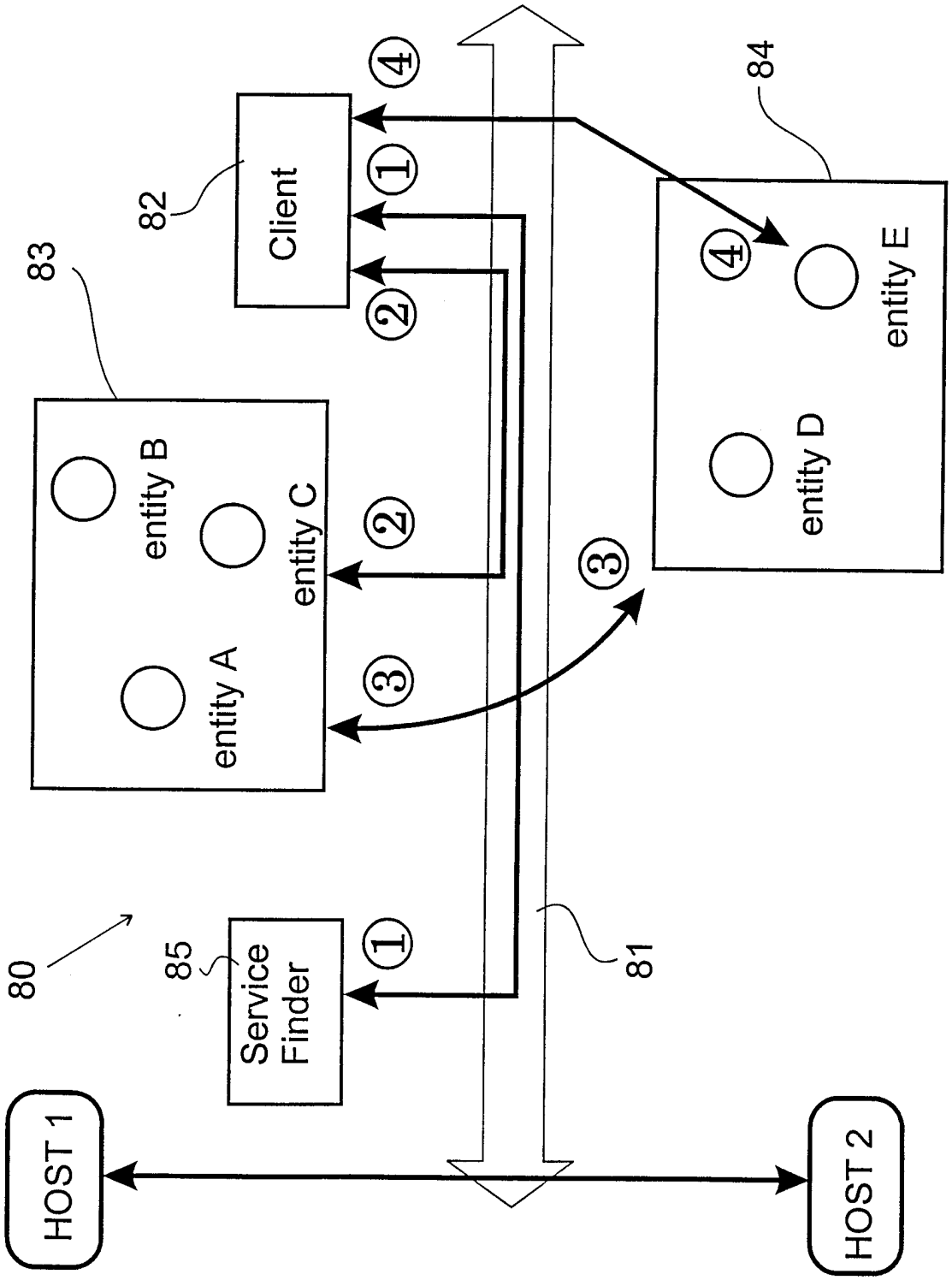

80
85
Service Finder
81
HOST 1
HOST 2
82
Client
83
entity A
entity B
entity C
84
entity D
entity E
1
2
3
4

FIG. 15

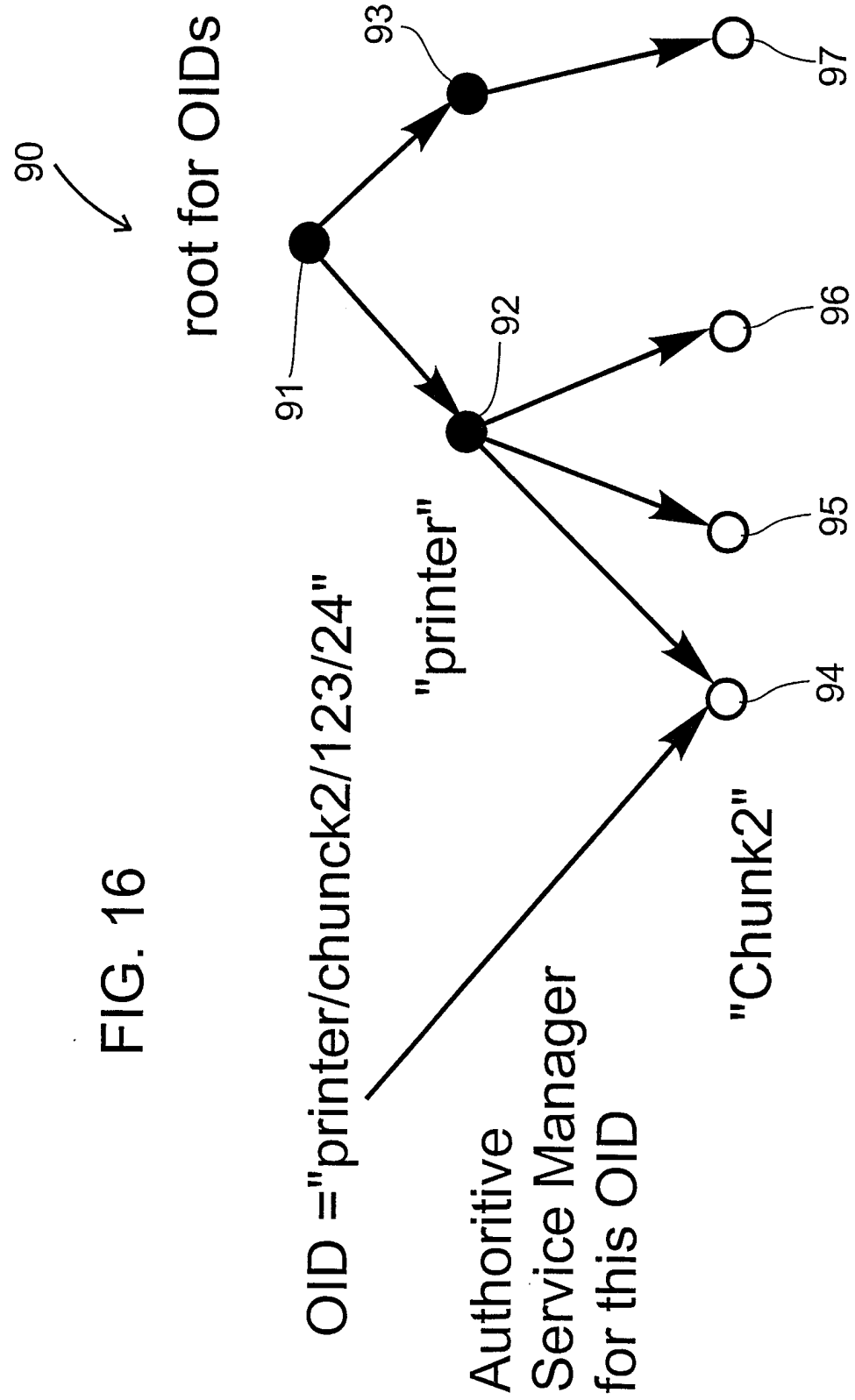

90
root for OIDs
91
93
97
92
96
"printer"
95
OID ="printer/chunck2/123/24"
94
"Chunk2"
Authoritive
Service Manager
for this OID

FIG. 16

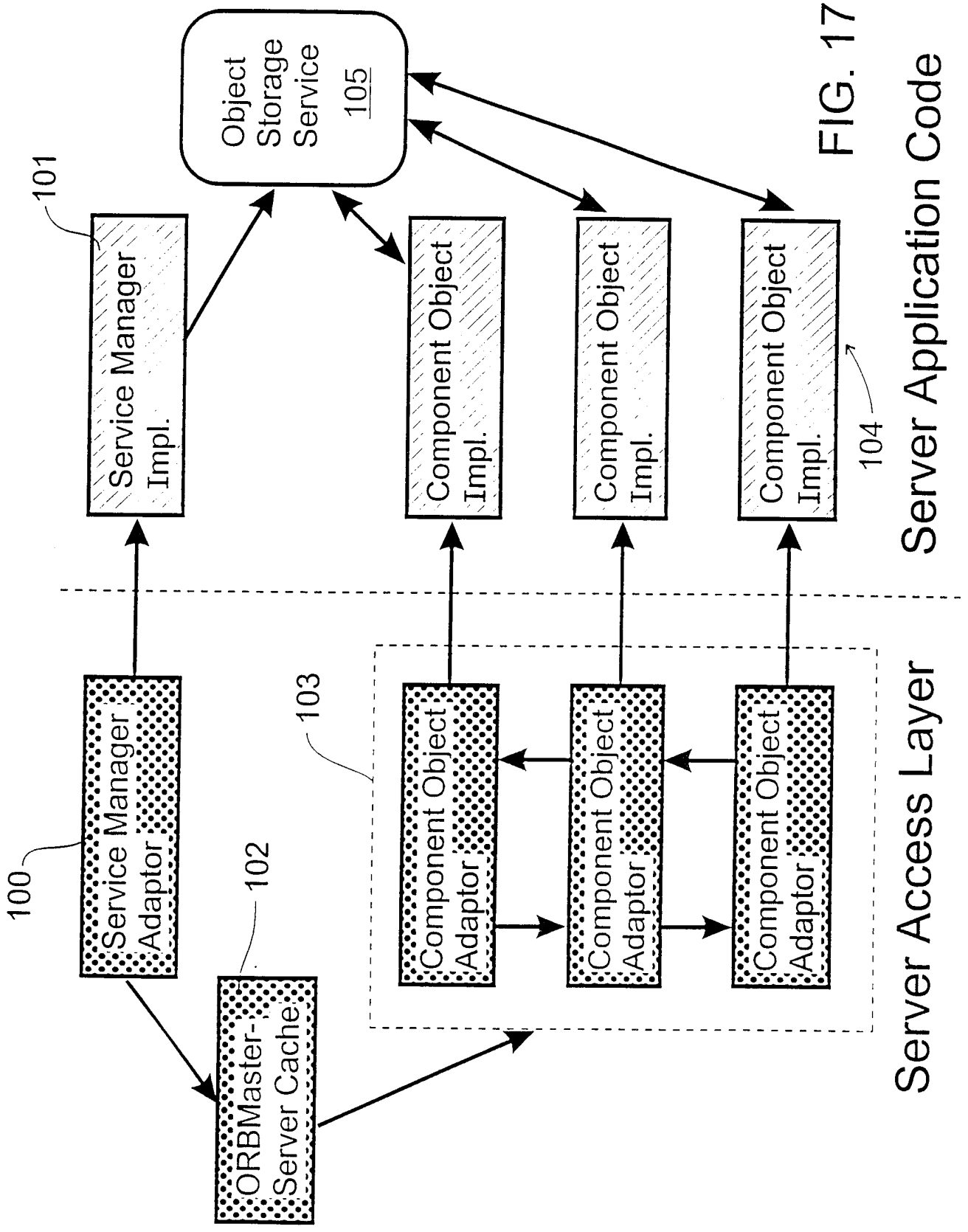

Object Storage Service 105
101
Service Manager Impl.
Component Object Impl.
Component Object Impl.
Component Object Impl.
104
Server Application Code
Server Access Layer
100
Service Manager Adaptor
102
ORBMaster-Server Cache
103
Component Object Adaptor
Component Object Adaptor
Component Object Adaptor

FIG. 17

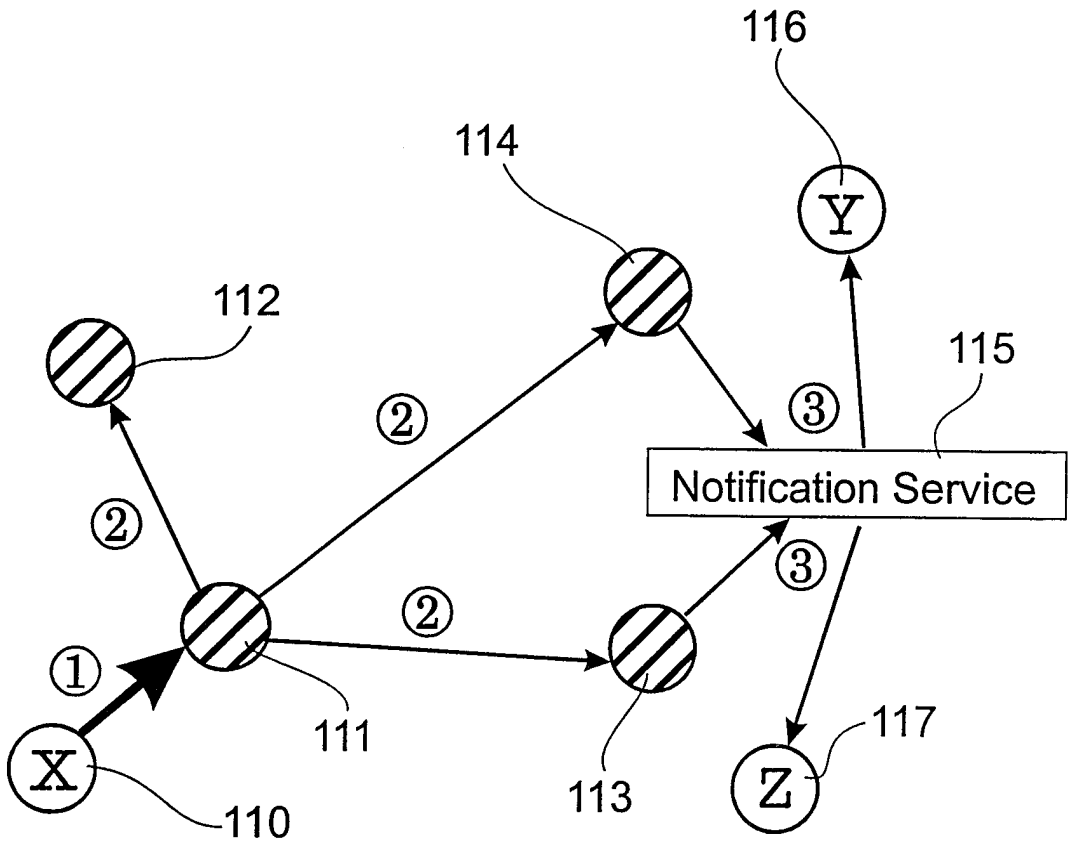

116
114
Y
112
115
②
③
Notification Service
②
③
②
①
111
X
110
113
Z
117

FIG. 18

INTERNATIONAL SEARCH REPORT

International Application No.
PCT/AU 98/00464

A. CLASSIFICATION OF SUBJECT MATTER

Int Cl[6]: G06F 9/45

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC: G06F 9/45

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
WPAT-OBJECT (W) ORIENT: AND (DISTRIBUT: OR DEVELOP: OR SERVER# OR TEMPLATE# OR CLIENT# OR CODE(W) GENERAT:)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
P, X	EP 0817035 (SUN MICROSYSTEMS, INC.) 7 January 1998 See in particular col. 2 line 45 to col. 4 line 52	1, 2, 6, 7, 11, 18, 20, 21, 25, 26, 28, 35
P, A	US 5699310 (GARLOF ET AL) 16 December 1997 See in particular col. 2 line 49 to col. 8 line 25	All

[X] Further documents are listed in the continuation of Box C

[X] See patent family annex

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
3 July 1998	- 8 JUL 1998
Name and mailing address of the ISA/AU AUSTRALIAN PATENT OFFICE PO BOX 200 WODEN ACT 2606 AUSTRALIA Facsimile No.: (02) 6285 3929	Authorized officer **J.W. THOMSON** Telephone No.: (02) 6283 2214

INTERNATIONAL SEARCH REPORT

International Application No.
PCT/AU 98/00464

C (Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	WO 97/22925 (OBJECT DYNAMICS CORP) 26 June 1997 See in particular page 15 line 1 to page 16 line 34	All
A	EP 0727739 (INTERNATIONAL BUSINESS MACHINES CORPORATION) 26 June 1996 See in particualr col. 1 line 54 to column 4 line 34	All

INTERNATIONAL SEARCH REPORT

Information on patent family members

International Application No.
PCT/AU 98/00464

This Annex lists the known "A" publication level patent family members relating to the patent documents cited in the above-mentioned international search report. The Australian Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

Patent Document Cited in Search Report		Patent Family Member	
EP	81735	JP	10091449
WO	9722925	AU	13318/97
EP	722739	JP	8263292

END OF ANNEX