Network Topology Processing Using CORBA Objects

S.A.Khaparde Binu.K.S R.K.Joshi Shubha Pandit S.A.Soman

Department of Electrical Engineering Department of Computer Science Department of Electrical Engineering

Indian Institute of Technology
Bombay, India 400076

Keywords: Object Oriented Programming, CORBA, Topology Processing

Abstract

Object oriented technology has been widely accepted by variuos engineering disciplines. The popularity of the technology stems from the fact that it allows modeling and evolution of large sized applications with ease. New functionality can be integrated into existing object oriented applications with ease.

All power system analysis begins with the processing of the topology of the circuit being analyzed. Hence the implementation of an object based topology processor is the first step in developing object oriented applications for power system analysis. Foley and Bose have provided algorithms for the development of an object oriented topology processor. The objects modeled in the topology processor proposed were physical component objects found in power systems. This paper proposes an enhancement to the object based topology processor by distributing the component objects using CORBA technology.

I. Introduction

The object paradigm has proved to be helpful in developing and maintaining large scale application code. The basic structural building blocks of an object oriented program are classes.

A class is an abstract data type that encapsulates data members and algorithms used to change or query the states of the data members. This creates a strong association between the algorithms and the data they process. The algorithms defined in a class are known as it's methods. The implementation details of a class are hidden behind its interface. The user's perception of the class is defined in it's interface composed of public. Programmers using a class are not required to know the implementation details of the class.

An object is an instance of a class. Since object oriented applications use the public interface of a class without knowing the implementation details, the class implementation can be changed with minimum changes in the application code. Hence large object oriented applications are relatively easy to maintain. Morover, inheritance allows specialized objects to be created from existing objects facilitating extensions with reuse.

Power system applications are usually a set of programs distributed over many computers. Distribution of an application over heterogeneous platforms is a complex task since each platform may have its own representation of data types which may not match with that of the other platforms. Hence passing parameters between processes running on different platforms is a fairly involved task. Any remote procedure call mechanism must take care of the mismatch between data representation across the platforms. Hence communication between objects in these environments requires a broker or middle-ware to take care of the differences at variuos levels. The Common Object Request Broker Architecture (CORBA) specification defines a standard for an object broker [1]. The salient features of the CORBA specification include object orientation, distribution transparency and language, platform and vendor independence.

The analysis of a power system begins with the topology processing. All other applications use the results provided by the topology processor. Hence the first step in creating distributed object power system applications is the implementation of a distributed object network topology processor.

Algorithms for object oriented network topology processors have been provided by Foley and Bose [2]. These algorithms need to be modified to allow distribution of the objects.

Section II covers the features of the CORBA specification. Section III describes the implementation of a CORBA based network topology processor. Section IV discusses the results obtained from the topology processor and the conclusions.

II. CORBA Based Network Topology Processor

This section describes the implementation of a topology processor that uses distributed CORBA objects. The topology processor has been implemented for a few basic power system objects in order to keep the application size small. The processor has been implemented using an algorithm proposed by Foley and Bose.

A. Implemented Objects

The topology processor has been implemented using physically based objects which include components, nodes and stations. The components implemented are circuit breakers, lines, sources and loads. Brief descriptions of the objects implemented are as follows :

- Nodes
  A node object maintains a list of the names of all component objects that are connected to it. It has a flag to indicate whether it is a bus. A node object may be a static node or a dynamic node. A static node is part of the static circuit description of a station. A static node object is created when a station is created and and is destroyed only when the station object is destroyed. A dynamic node is a node which is formed when the topology of a station is resolved. These nodes are dynamic objects which are destroyed when the station resolves its topology again. Static and dynamic nodes are instances of the same node class.

- Lines
  A line object connects two nodes which may belong to different stations. It stores the name of the two static nodes to which it is connected. A line also stores the names of two dynamic nodes to which it is connected.

- Breakers
  The breaker object connects two nodes belonging to the same station. This object stores the names of the two static nodes to which it is connected. A breaker object does not get connected to dynamic nodes, since the dynamic topology of a station does not include circuit breakers. The breaker object has a field to store its state. The breaker's state may be set as on or off.

- Sources and Loads
  Sources and loads are objects connected to a single node. These objects store the name of the static and dynamic nodes to which they are connected.

- Station
  A station object stores three different lists which are :

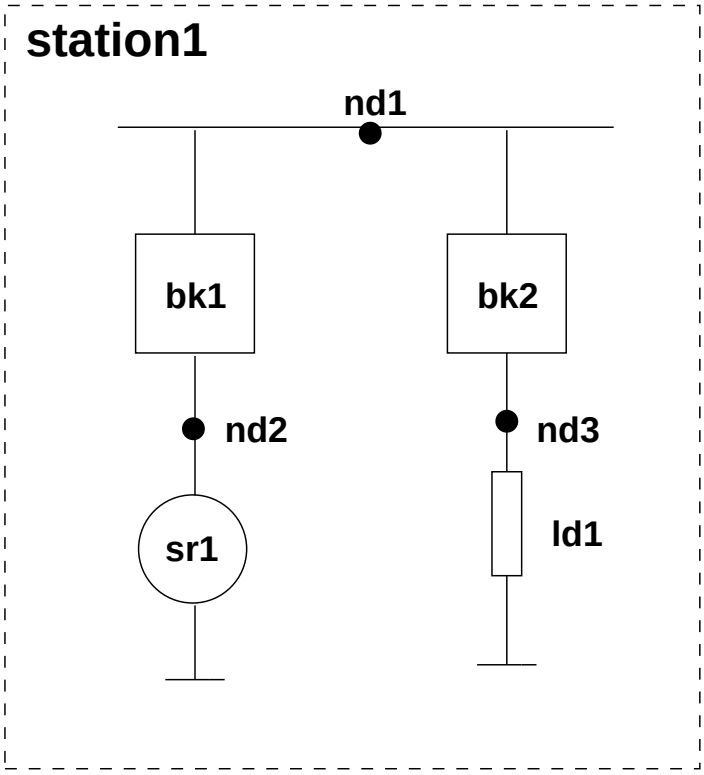

Figure 1: A sample circuit

  * A component list which stores the names of all the components that are part of the station's circuit.

  * A static node list which contains a list of the names of all nodes that are part of its static circuit description.

  * A dynamic node list which contains the names of the nodes that are created when the topology of the station is resolved. This list changes every time the topology of a station is solved after after a change in the status of any breaker.

Figure 1 shows a simple station circuit with three nodes, two breakers, a source and a load. Figure 2 shows the relations between the component, node and station objects that represent this circuit. An arrow pointing to an object represents a reference to the object. Bidirectional arrows show that the objects reference each other. The dynamic node list shown in the figure is for the case when both the breakers are closed (in the on state). A second dynamic node would have been created if one of the breakers were open.

B. Object Naming Scheme

When objects are distributed over a large network, locating a specific object can be simplified by using a naming service. The CORBA standard defines a naming service that enables binding of object addresses to simple names [3]. To make use of the naming service, a naming scheme has to be devised to assign names to all the objects in a network. Figure 3 shows the naming hierarchy that has been devised to name the power system objects. The underlined labels in the figure represent object instances. The object *line1* in the diagram

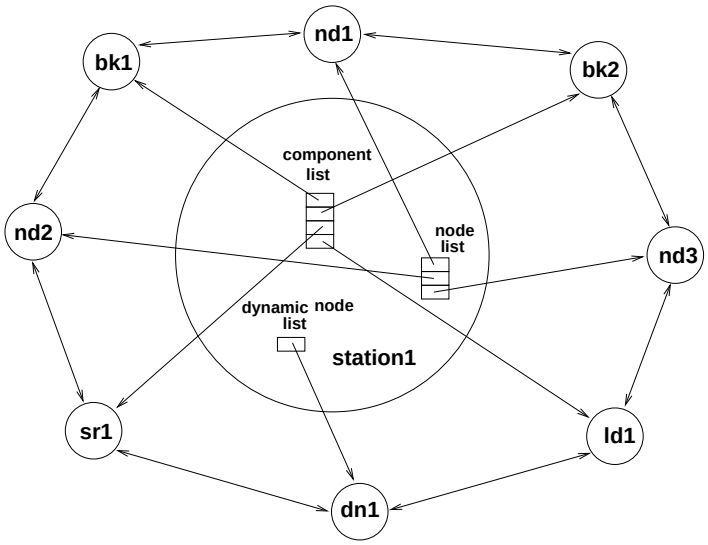

Figure 2: Objects for the sample circuit

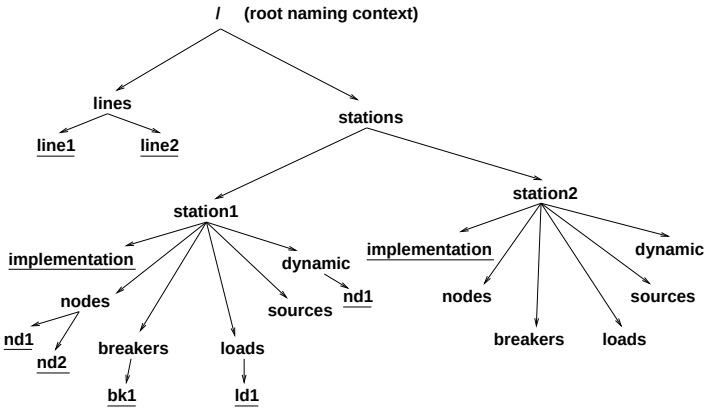

Figure 3: The Naming hierarchy

can be accessed using a name of length two. The name of this object is *lines/line1* . The object *ld1* in the figure has the name *stations/station1/loads/ld1* which is a name of length four. Special objects, called naming contexts, are used to create a naming hierarchy like the one shown in the figure. A naming context is an object defined in the CORBA standard, that allows other objects to create *name-address* bindings. As shown in the figure, the root naming context contains two naming contexts *stations* and *lines*. All line objects are bound to the *lines* naming context. Every station object creates a naming context of its own under the *stations* context when it is instantiated. All station names must be unique. The figure shows two naming contexts created by *station1* and *station2*. The station object binds to a standard name *implementation* under the naming context of its name.

Every station creates five naming contexts under its own naming context for the different kinds components available. A component in a station binds its name to one of these naming contexts depending on its type.

Nodes that are part of the static description of a station bind to the *nodes* naming context of the station to which they belong. They are instantiated when the station object is instantiated. When a station receives a message to resolve its topology, new nodes are instantiated depending on the status of the breaker objects in the station. These nodes are bound to the naming context *dynamic*. The node objects in the *dynamic* naming context give the current topology of the station.

This hierarchical naming scheme prevents clashes in the naming of components that are likely to occur if all names were bound to the root naming context. For instance, both station1 and station2 can have load objects named *ld1* . The full names of the loads would be *stations/station1/loads/ld1* and *stations/station2/loads/ld1* which uniquely identify the two load objects.

C. Topology Processing Algorithm.

The station object forms a new set of dynamic nodes when it receives a message to resolve its topology. All nodes that are part of the previous dynamic circuit are first destroyed. All nodes contain a topology flag that indicates whether the node has joined the topology being formed. The station object clears the topology flags of all the nodes. The station object then creates a dynamic node and asks the first node in its static node list to join this node. The method invoked is called the *join-topology* method. The dynamic node being formed is sent as a parameter to this method. When control returns to the station object, the dynamic node will contain a list of components. If this list is empty, the dynamic node is passed again as a parameter to the *join-topology* method to the next static node object in the list. Otherwise, the dynamic node is added to the list of the station's dynamic node list and a new dynamic node is formed for the next invocation of the *join-topology* method. The algorithm for the resolve method of the station object is listed below :

```
Destroy all the dynamic nodes in the dynamic
 node list.
For all nodes in the static node list
      clear topology flag
Form a new dynamic node and assign it to
 the current node
For all nodes in the static node list
{
      if current node has any component
       attached
      {
            add the current node to the
             dynamic node list
            form a new node and set it
             as the current node
      }
      send node the join-topology message
       with the current node being formed.
}
```

On receiving the *join-topology* message from the station object, a node first checks if it has already received the message using the status of its topology flag. If its topology flag is set, the node returns control to the station without doing

anything. Otherwise it invokes the join-topology method on all the components that are connected to it with the name of the dynamic node that it received from the station object. The algorithm is listed below :

```
If topology flag is set
{
      return
}
Set the topology flag
For all components attached to the node
{
      Pass the join-topology message with
       the current-node as parameter.
}
```

When a breaker object receives the *join-topology message*, it checks its current state. If the breaker is off, it returns control to the node object that invoked the method without doing anything. If the breaker is on, it invokes the join-topology method on the node from which it has not received the current invocation of the method with the dynamic node it received as the parameter. This will result in the components connected to it to join the dynamic node. Hence components connected to the nodes on either side of the breaker will connect to the same dynamic node. The algorithm is listed below :

```
If breaker is on
{
      pass join-topology message to the node from
       which it has not received the invocation.
}
else
{
      return.
}
```

All other components that receive the join-topology message join the node which is passed as a parameter to them. The algorithm used by these components is as follows :

Figure 4 shows the flow of control between the objects for the example circuit shown in figure 1. The control flow diagram is for the case when breaker *bk1* is closed (on) and *bk2* is open (off). The arrows show the invocation of the *join-topology* method. Numbers alongside the arrows indicate the sequence of execution. The name of the dynamic node that is passed as the parameter to the method is written inside parentheses alongside. The return of control to the object that invokes the method is not shown in the figure. The sequence results in the creation of two dynamic nodes *dn1* and *dn2* in the station. The first dynamic node contains the name of the source object and the second dynamic node contains the name of the load object after the whole sequence is executed.

D. Object Interfaces

The algorithms discussed in the previous section require that all component objects support the following two methods:

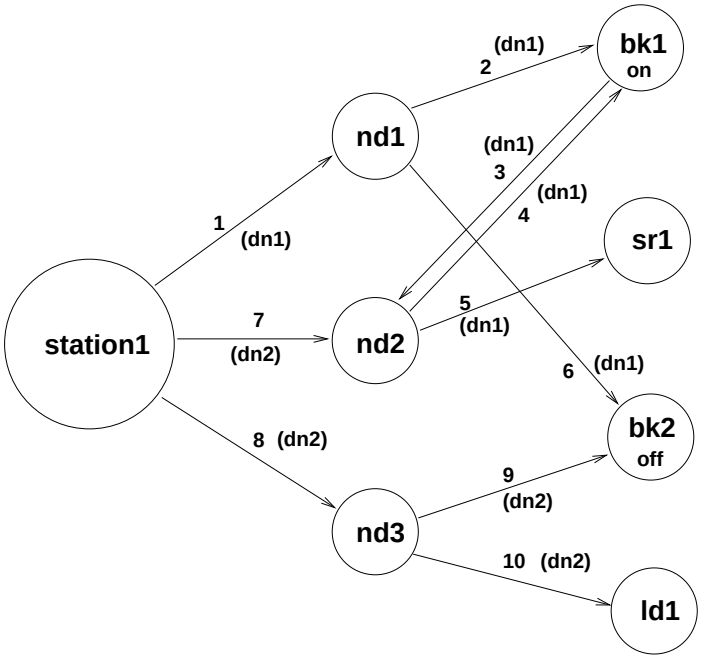

Figure 4: Control flow diagram

- The join method
  The join method acts differently depending on whether the component is a node or a normal component. The method accepts an object name as a parameter. In the case of a node, the parameter is a component name and the node adds the component's name to a list of connected components that it maintains. All other components assume that the name passed is a node object name. The join method of these components stores the name of the node passed and invokes the join method of the node passed with the component's name as the parameter. This method is different from the join-topology method defined in the topology processing algorithms. The method also takes a parameter that indicates the branch number of the component which is to be joined to the node passed.

- The join-topology method
  This method accepts a node name as its parameter. The name passed is the name of the node under formation. The method is implemented differently in each component according to the algorithm presented in the previous section. This method also takes a parameter that gives the branch number of the component that is to join the dynamic node.

In addition, a method called dump is included in the interface. This method takes a string reference as a parameter. The string reference on return points to the status information of the object.

The definition of the component interface which contains the methods mentioned above has to be written in a language called the Interface Definition Language (IDL). IDL makes CORBA language independent. An object interface provided

in IDL can be parsed by an IDL compiler to generate class definitions in any language that is supported by a CORBA implementation. The interface definition of the component object in IDL is listed below :

```
interface component
{
void join_topology(in CosNaming::Name jnode,
 in unsigned short branch);
void join(in CosNaming::Name jnode,
 in unsigned short branch);
void dump(out string str);
};
```

Apart from these common methods for the topology processing, a component object can implement methods to query or modify its state. For instance, the breaker object has the following methods which are unique to it:

* An on method to close the breaker
* An off method to open the breaker
* A status method that returns the current state of the breaker.

The interface to a breaker object is derived from the component interface in the following manner :

```
interface breaker : component
{
void on();
void off();
CORBA::short status();
}
```

This definition implies that the breaker object will have to implement the three methods on, off and status in addition to the methods defined in the component interface. A client application may access the breaker object as a component or as a breaker object. When a client accesses the breaker as a component, the methods defined in the component interface are the only methods that it can invoke. These methods are used by the topology processing application. To access the other three methods, a client application has to access the breaker using the breaker interface. For example, a client aplication that monitors the states of all breakers in a circuit would have to use the breaker interface to query the status of the breakers.

Other components like sources and loads may contain methods for querying their injection or voltage state which can be used by the load flow programs. The advantage of having a common interface for all component objects is that the topology processor's code need not be changed to accommodate any new component types as long as the new type supports the component interface.

The interface definition of the station object is listed below :

```
interface station
{
        void resolve();
        void dump(out string str);
        void dyn-dump(out string str);
};
```

The *resolve* method causes the station object to resolve its topology. The station object clears the topology flags and invokes the join-topology method on all the nodes that it contains.

The *dump* and *dyn-dump* methods return the static node listing and the dynamic node listing of the station respectively.

III. Results

The topology processor described was implemented for testing on a free CORBA implementation called MICO [4]. This implementation supported objects in C++. The test circuit consisted of six stations with thirty three components. The test circuit is shown in figure V. All objects were distributed over three pentium based computers running the Linux operating system. *Station1*, *station4* and all line objects were served by the first computer. The second computer served *station2* and *station5* and the third computer served *station3* and *station6*.

A client application was developed to test the topology processing. The application could be run on any computer in the network and could connect to any station object given its name and a simple interface was offered using which the user could do the following :

- Query the state of any circuit breaker in the station by name.
- Opening and closing any circuit breaker by name.
- Obtaining the static circuit description of the station.
- Obtaining the current dynamic description of the station.
- Sending the resolve method to the station.

The following listing is the output of the client application when it was run on the first computer and used to modify the breaker settings of *station5*.

```
Enter station to connect to ... station5
Connected to station station5

********** Static description station5
*********** Node id 5 **************
stations/station5/sources/1
stations/station5/breakers/4
*********** Node id 4 **************
lines/line_56
stations/station5/breakers/3
*********** Node id 3 **************
lines/line_54
stations/station5/breakers/2
*********** Node id 2 **************
lines/line_45
```

```
stations/station5/breakers/1
********** BUS  id 1 **************
stations/station5/breakers/4
stations/station5/breakers/3
stations/station5/breakers/2
stations/station5/breakers/1

station5>breaker 1 on
station5>breaker 2 off
station5>breaker 3 on
station5>breaker 4 on
station5>resolve
station5>dynamic
********** Dynamic description station5
********** Node id 2 **************
lines/line_54
********** BUS  id 1 **************
lines/line_45
lines/line_56
stations/station5/sources/1

station5>breaker 2 on
station5>resolve
station5>dynamic
********** Dynamic description station5
********** BUS  id 1 **************
lines/line_45
lines/line_54
lines/line_56
stations/station5/sources/1

station5>
```

The topology of the station which consisted of a node and a bus when *breaker2* was open changed when it was closed. Similar tests were carried out on the other stations in the circuit and the topology was resolved successfully. Although all objects were modeled in the C++ language and implemented on the Intel platform running Linux, it is possible to model the objects in other languages on other platforms.

IV. Conclusions

The use of CORBA based distributed object solutions will make the development, maintenance and upgradation of power system objects easier and less prone to errors. The proposed naming scheme for the component objects makes them easily locatable irrespective of their flexible machine bindings. Different application components can be implemented on platforms that are best suited for them and the integration of the applications can be achieved easily. It is envisaged that distributed control and exchange of information among stations would become necessary and feasible with the advent of new technologies. This work would serve as a first step in this direction. With the limited experience gained, this technology seems promising.

References

[1] OMG, CORBA Specificatons, http://www.omg.org//, 1998

[2] M.Foley and A.Bose, "Object Oriented On-Line Network Analysis", IEEE Transactions on Power Systems, Vol. 10, No. 1, Feb. 1995, pp. 125-132.

[3] OMG, Naming Service Specifications, http://www.omg.org//, 1998

[4] Kay Romer and Arno Puder, "The MICO Manual", http://www.vsb.cs.uni˜frankfurt.de/˜mico/, 1997.