Using CORBA Middleware in Finite Element Software

J. Lindemann, O. Dahlblom and G. Sandberg

Division of Structural Mechanics, Lund University
strucmech@byggmek.lth.se

Abstract. Distributed middleware technologies, such as CORBA can enable finite element software to be used in a more flexible way. Adding functionality is possible without the need for recompiling client code. Transfer of data can be done directly, without the need for intermediate input and output files. The CORBA software components can be easily configured and distributed tranparently over the network. A sample structural mechanics code, implemented in C++ is used to illustrate these concepts. Some future directions, such as placing CORBA enabled finite element software on HPC centres are also discussed.

1 Introduction

A complex hardware product often consists of many exchangeable components. As long as a component fits into the product, the internal implementation can differ. Software components are analogous to hardware components. Components in programs can be exchanged without the need for recompilation, as long as the component interface is unchanged. The use of components in software development has increased during the last few years. The reason for this is the need to reduce the size of the client programs. When the first client/server systems appeared, the client software were often large programs. Most of the processing was done in the client program and the database server was used as data storage. The problem with these systems was the cost of installing and maintaining the client software. New systems developed today often use a thin client with little or no data processing capabilities. Instead of calling the database servers directly, they use a set of components placed on central servers for data processing. These components then access the database servers. The advantage of this approach is that the components can be placed on powerful systems, reducing the amount of processing needed at the client. This approach has been successfully applied to database applications. It is of interest to apply this technique to analysis software as well. Using the technique of distributed computing, clients can use components as if they were located on the same machine, making it possible to create integrated programs with transparent access to computational resources, such as available workstations on the network or resources at High Performance Computing (HPC) centres. This would make high performance computing more available to a wider user group.

P.M.A. Sloot et al. (Eds.): ICCS 2002, LNCS 2331, pp. 701–710, 2002.

The present work describes structural analysis software, where the computational parts of analysis codes can be placed as components on remote servers. Before describing the structural analysis code, a brief overview of client/server architecture will be given.

2 Client/server architecture

Three-tier and n-tier applications emerged from the need to shield the client program from changes at the server side by placing a layer between the client and the server. The history of the client/server architecture is described by Schussel [15]. For a more detailed description over the client/server architecture, see Orfali and Harkey [12]. The logical three-tier or n-tier model divides an application into three or more logical components. Each component is responsible for a well-defined task. In a database application there would be a presentation layer for displaying data and modifying data, a logic and rules layer and a database layer responsible for storing the data.

The components of the logical model can be grouped together in different configurations to form a physical model. One of the most interesting combination of the logical model is when the three logical services are placed as separate applications on different computers, forming a physical three-tier application. This implementation enables developers to have a greater flexibility in the choice between different hardware and software configurations.

3 Distributed computing

Distributed computing is defined as a type of computing in which different components and objects comprising an application can be located on different computers connected to a network; for an overview see [10].

Currently, there are today three coexisting technologies for distributed object computing DCOM [2], RMI [14] and CORBA [1]. Microsoft's distributed COM (DCOM) extends the Component Object Model to be used over the network. RMI or Remote Method Invokation [14] is a distributed technology based on the Java language. CORBA is the Object Management Group's [1] specification for interoperability and interaction between objects and applications. Objects and applications can be placed on any platform and accessed from any platform. CORBA is a specification, and therefore platform-independent.

This paper describes an implementation in CORBA. In a previous paper [7] a DCOM based implementation has been studied.

4 CORBA

4.1 Concepts and Terminology

To describe a CORBA based implementation, it is important to understand some terminology and concepts of a CORBA implementation. A more thorough

description can be found in Henning and Vinoski [6]. Some of the more important concepts and terminology is shown below.

- A *client* is an entity that invokes a request on a CORBA object.
- A *CORBA object* is a "virtual" entity capable of being located by an ORB and having client requests invoked on it.
- A *server* is an application with one or more CORBA objects.
- An *object reference* is a handle used to identify, locate and address a CORBA object. Object references is the only way for a client to access CORBA objects.
- A *servant* is a programming language entity that implements one or more CORBA objects.

Communication in CORBA is done by a client invoking requests on a CORBA object through either a statically linked stub in the client application or through the dynamic invocation interface (DII). The requests are dispatched to the local ORB which in turn dispatches these requests to an ORB on the remote machine. The remote ORB then dispatches the request to an object adaptor, which then directs the request to the servant implementation code.

4.2 Interface definition language

To access a CORBA object the client must know which methods and properties it contains. This description is called an interface. To describe such interfaces CORBA uses the Interface Definition Language (IDL). In this language the object interfaces are described. Using a separate language for describing the objects makes CORBA language neutral. This enables CORBA applications to be implemented in a variety of different languages. To implement CORBA clients and objects the IDL definition is compiled using an IDL compiler. This compiler takes the interface definition and generates the implementation code for both client and server, in the desired implementation language.

The following code shows an example of a simple IDL interface, declaring an interface to an `Echo` object. In this case the object echoes the string word back to the calling client.

```
interface Echo {
    string Shout(in string word);
}
```

Compiling this example using a C++ IDL compiler, will generate a header file and an implementation source file for accessing the object described from a C++ based application and the skeleton code for implementing the servant object in C++.

4.3 Name service

One of the biggest benefits of CORBA is location transparency. Information about server location is often not included in the client application. This makes it easy to configure a client server setup. A client only needs an object reference to connect to an object. Object references are unique identifiers, which also include information about the location of objects. To connect to objects the client needs a way of retrieving an object reference. Before the introduction of CORBA 2.3, object references were often transferred using files over a network file system or using a non-standard method of name lookup. In CORBA 2.3 a name service was introduced. The name server stores object references in a human readable form. When a server is started, it creates an entry in the name server for the object reference. The client then queries the server by name to receive the object reference. By using a name server, client/server configuration can be done transparently. Name server location is the only thing that has to be configured for the servers and the clients. Clients and servers get the location of the name server by specifying special command line options.

4.4 Object creation and destruction

Before request to an object can be made, the object implementation (servant) must be instantiated and activated. In CORBA this is done by the object adaptor. Earlier CORBA specifications only included a limited basic object adapter (BOA). To enhance the functionality of this object adaptor many ORB vendors added non-standard extensions. The consequence of this was that the server side of a CORBA application became ORB dependent. With CORBA 2.3 this limitation was removed by the introduction of the Portable Object Adaptor (POA).

Different types of policies for the creation and destruction of objects can be specified using lifetime policies for the portable object adaptor (POA) in CORBA. Figure 1 illustrates the typical lifetime of a CORBA object. The default

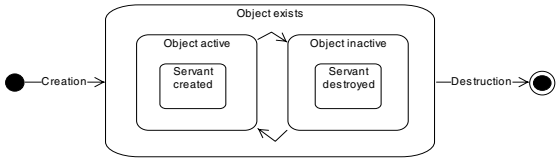

Fig. 1. Object creation and destruction

policy is `TRANSIENT`. In this policy the object can not be reactivated, when it has been deactivated. The object reference of an `TRANSIENT` object is only valid when the object is active. The `PERSISTENT` lifetime policy enables objects to be activated and deactivated multiple times. This requires that the object servants are able to store their state in a persistent form between the activations.

Because CORBA is a distributed technology, the creation of objects must be handled in a different way than it is handled when creating local objects.

In a CORBA system, objects are created by special factory objects. These factory objects can be seen as the equivalent of an object constructor in the C++ language.

The destruction of a CORBA object is not done by the factory, instead a special method is declared in the object interface for removing the object. If the factory was responsible for destroying the object, the client referencing the object would also have to reference the factory when destroying the object. This can be quite complex if the object reference has been passed from object to object. The process of creating and destroying is discussed in detail in Henning and Vinoski [6].

5 Finite element CORBA implementation

The educational software ForcePAD [4] was modified to use a CORBA based finite element solver. The ForcePAD application is an intuitive tool for visualising the behaviour of structures subjected to loading and boundary conditions. ForcePAD uses a bitmap canvas on which the user can draw the finite element model using standard drawing tools. When the calculation is executed the bitmap image is transferred to a finite element grid, which is then solved. The main window is shown in Figure 2. The application consists of four components divided

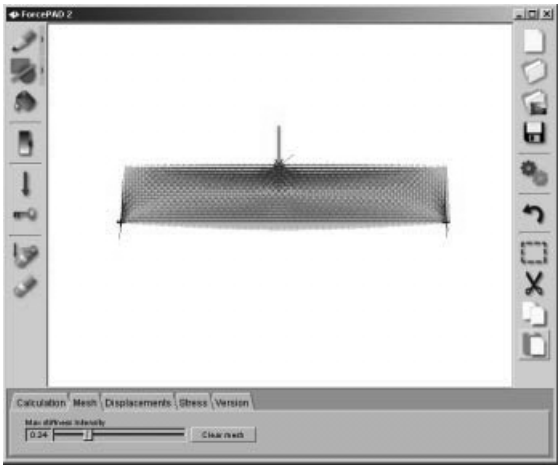

Fig. 2. Sample CORBA application

into three layers, as shown in figure 3. The user interface is responsible for interactively defining the problem. The ForcePadSolver component contains the interfaces used to describe the finite element model used in the application. The name server components handles the location of available CORBA ForcePADSolver components in the network. The FE solver components are responsible for executing the calculations. By providing the functionality of the application

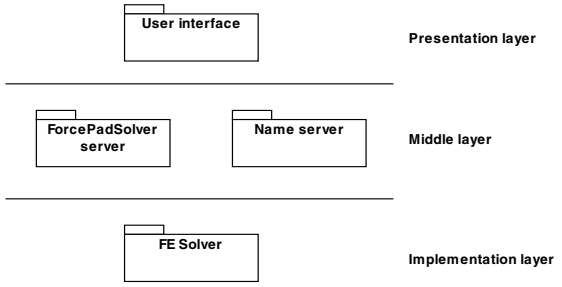

Fig. 3. Application components

in a component based form the application can be configured and maintained in a more flexible way.

5.1 ForcePadSolver server

The middle layer of the application is implemented in a single server. The ORB used in the implementation is ORBacus [11], which is a commercial ORB available with source for multiple platforms including Microsoft Windows and many Unix dialects. It can also be used without cost for non-commercial use. The FE solver is implemented in C++ using the newmat09 [8] library, which is freely available with source code. In this version of the application, the FE solver is statically linked into the ForcePadSolver server, but it is possible to implement the FE Solver as a separate CORBA object or use a standard FE code.

The interface of the ForcePadSolver was designed to reduce the number of requests needed to be made on the CORBA objects. Every request on a CORBA object has cost determined by the latency and marshalling rate. The latency is the cost of sending a message. The marshalling rate is the cost of sending the input and return variables. For a more detailed discussion see chapter 22.3 in Henning and Vinoski [6]. One of the most critical factors for performance is the latency. The latency time of invoking a request on a CORBA object is approximately 500-5000 times higher than doing a function call in C++. The main interface in the server is the `FemSystem` interface. Every time a client connects to the server it will create this object, using the `FemSystemFactory` factory object. The factory object is instantiated and registered in the name server when the server is started. The `FemSystem` object, when instantiated will create an instance of a `FemSolver` object and a `FemGrid` object. These objects are returned from the `FemSystem` object. A ForcePadSolver server can hold one instance of `FemSystem` objects for each client connected to the server, as shown in figure 4.

The code below shows how a `FemSystem` object is created from C++ using the `FemSystemFactory` object.

```
femSystemFactory = ... Get from name server ...
femSystem = femSystemFactory->create();
femGrid = femSystem->getFemGrid();
femSolver = femSystem->getFemSolver();
```

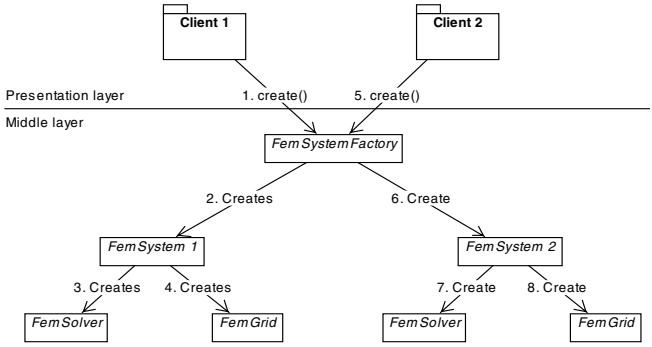

Fig. 4. Object creation using the FemSystemFactory object

The `FemGrid` object defines the finite element model and the `FemCalc` is used to control the calculation of the finite element model.

To reduce the marshalling times for the FE model, data will mainly be transferred using the CORBA data type `sequence`. This data type is a dynamic array of a specified type. The following code illustrates a typical data transfer from the client to a CORBA object in the ForcePAD client application.

```
// CORBA defined datatype:
//     typedef sequence<double> TStiffnessVector;
ForcePadSolver::TStiffnessVector stiffnessVector(nStiffness);
stiffnessVector.length(nStiffness);
// Transfer internal fem model to stiffnessVector
l = 0; float value;
for (i=0; i<rows; i++)
    for (j=0; j<cols; j++)
        for (k=0; k<2; k++) {
            value = m_femGrid->getGridValue(i, j, k);
            stiffnessVector[l++] = (double)value;
        }
// Invoke request on femGrid CORBA object
femGrid->setStiffness(stiffnessVector);
```

When all input data has been transferred to the CORBA object `FemGrid`, the finite element model can be solved. The execution of the finite element solver is controlled by the `FemCalc` object. The following code from the client application shows how the calculation is initiated:

```
femSolver->execute();
error = femSolver->getLastError();
```

In the ForcePadSolver server the `execute()` method is implemented as a blocking call. This means that the execution of the client application will wait until the server is finished. To solve this, the `execute()` could be implemented as an asynchronous method call in CORBA. Additional methods for monitoring the execution would have to be added to the interface as well.

The results from the calculation are also retrieved using the CORBA data type `sequence`. The difference is that the `sequence` vectors now are preallocated and must be transferred back to the C++ class `CFemGrid`. The following client code shows how the results are retrieved from the `FemGrid` object.

```
// CORBA defined datatype:
//         typedef sequence<double> TDisplVector;
ForcePadSolver::TDisplVector* displacements;
// Invoke request on femGrid CORBA object
femGrid->getDisplacements(displacements);
// Store displacement values in local class m_femGrid
m_femGrid->setDisplacementSize(displacements->length()); for (i=0;
i<displacements->length(); i++)
    m_femGrid->setDisplacement(i+1, (*displacements)[i]);
// We are responsible for deleting the return values
delete displacements;
```

The lifetime policy used in the ForcePadSolver server is `TRANSIENT`. A calculation in ForcePAD does not execute over several days, so the policy `PERSISTENT` will not be necessary in this case, it is better suited for applications executing over several days. The client applications can then connect and disconnect to object during the execution.

5.2 Server implementation

The ForcePAD solver server is implemented as a C++ console application using the ORBacus [11] ORB. A skeleton implementation for the server is generated using a special switch in the ORBacus IDL compiler.

To handle object creation and destruction automatically, each servant is also derived from the `RefCountServantBase` base class. This class implements a reference counting scheme which automatically destroys the object servant when there are no connections to the object. Depending on the implementation, more complex schemes of object creation and destruction can be implemented, see [6] for more details.

The process of executing a calculation starts with a request to the `FemSolver` method `execute()`. The `FemSolver` reads the input model from the `FemGrid` object and assembles the finite element model. The solver from the `newmat09` [8] is then called. When the solution is found the results are stored back in the `FemGrid` object. The results are now available to the client application.

5.3 Client/server configurations

The easiest configuration of the finite element system is to install the client application together with the ForcePADSolver server and the finite element solver on a single computer, see Figure 5. This configuration is typically used to do calculations that fit into the memory of the local machine.

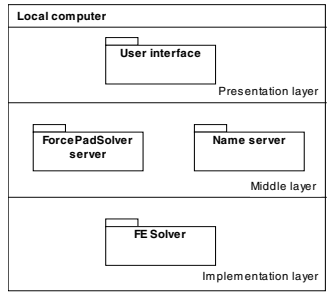

Fig. 5. Local configuration

In the first distributed configuration, the middlelayer and implementation are moved to a separate computer. This configuration requires the server to be able to run a CORBA ORB. If the server running the finite element solver does not support running an ORB, the middlelayer can be placed on a separate computer. Execution of the finite element solver can then be done using `rexec`, `rsh` or `ssh` utilities. Figure 6 show two of the possible configurations. Many more configurations are possible. By providing location transparency, the CORBA objects can be configured in almost any way without needing to recompile the clients and the servers.

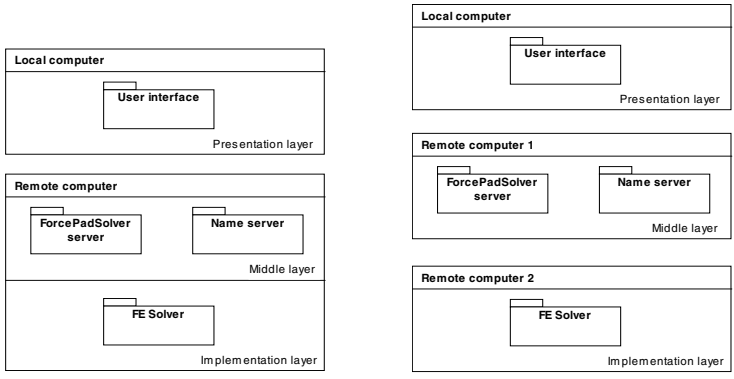

Fig. 6. Remote configuration 1 and 2

5.4 Client application

To create a platform independent application, ForcePAD uses the fast light toolkit (FLTK) [3]. FLTK [3] is a lightweight user interface toolkit written in C++. The toolkit can be used on Windows 98/NT/2000 and most Unix dialects with good performance. The 2D graphics in ForcePAD is implemented using OpenGL [9].

One goal of the client application is to hide the CORBA implementation from the user. The user should not be able to notice that the client is using CORBA for interfacing with the ForcePADSolver server.

6 Conclusion

Using a three-tier implementation with interfaces and components, creates a very flexible finite element application. The three-tier implementation protects the client applications from changes in configuration and solver design. Components are easily configurable and maintainable, reducing the need for further development. By using interfaces when communicating with components, the need to recompile client software when a new functionality is introduced in the solver components is reduced. Interfaces can also be published enabling other software to use the finite element application in an effective way. The CORBA specifications also enable new ways of using software. Client software can easily distribute calculations over available workstations. High Performance Computing (HPC) centres would be able to host a set of applications as CORBA objects. From a web site, users can register themselves as users and download client applications that connect to the objects. This would make high performance computing more available to a wider user group.

References

1. Object Management Group, Inc., http://www.omg.org, 2000
2. Microsoft Corporation, DCOM Technical Overview, 1996
3. B. Spitzak, Fast Light Toolkit FLTK, http://www.fltk.org, 2000
4. Division of Structural Mechanics, Lund Univeristy, ForcePAD, http://www.byggmek.lth.se/bmresources/forcepad, 2001
5. GNU Project, http://www.gnu.org, 2000
6. M. Henning and S. Vinoski, Advanced CORBA Programming with C++, Addison Wesley Longman Inc., 1999
7. J.Lindemann, O. Dahlblom, G. Sandberg, An Approach For Distribution Of Resources In Structural Analysis Software, ECCM 99, München, Germany, 1999
8. R. Davies, Newmat09: C++ matrix library, http://webnz.com/robert/cpp_lib.htm #newmat09, 2001
9. OpenGL, http://www.opengl.org, 2000
10. The Open Group, http://www.opengroup.org/dce, 2000
11. Object Oriented Concepts Inc., ORBacus 4.0, http://www.ooc.com/ob, 2000
12. R. Orfali and D. Harkey, Client/server programming with Java and CORBA. - 2nd ed., John Wiley and Sons Inc., 1998
13. G. Reilly, Developing Active Server Components with ATL, Microsoft Corporation, 1997
14. Sun Microsystems Inc., JavaTM Remote Method Invocation, http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html, 2001
15. G. Schussel, Client/Server: Past, Present and Future, http://www.dciexpo.com/geos/dbsejava.htm, 1996
16. S. Willliams and C. Kindel, Microsoft Corporation, The Component Object Model: A Technical Overview, 1994