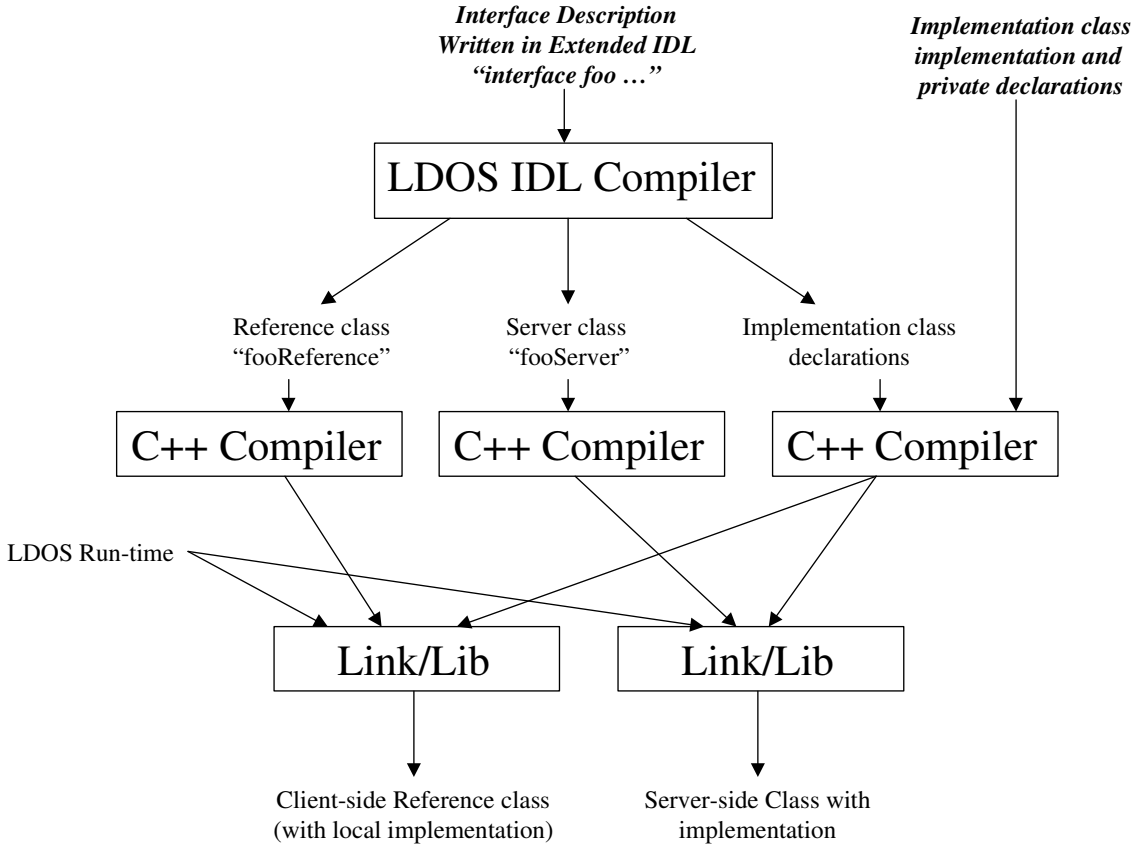
Programmer's view of LDOS system

Peter A. Dinda

LDOS (Lightweight Distributed Object System) lets you build distributed objects without dealing with the details of communication. The idea is that you write a description of your class's interface in the CORBA IDL (Interface Definition Language) and a local implementation of its class. The LDOS IDL Compiler generates the necessary glue code for your implementation to use the LDOS run-time system. The end result is a "server" class which wraps your implementation and makes it possible to be called from anywhere on the network, and a "reference" class through which those calls can be made. The reference class is a "smart pointer" in that it makes the object being called appear to be local, whether it is or not.



Example: An Accumulator Class

Suppose we want to implement a class that has a sum which can be added to and read. First, we define the interface in IDL:

```
interface Accumulator {
   void          Reset();
   void          Add(in unsigned long x);
   unsigned long Get();
};
```

If you are unfamiliar with CORBA IDL, note that this is similar to a C++ class definition, except that "class" is replaced with "interface", arguments have direction ("in"), and no state is declared. At this point, we have pure IDL, which can be compiled by any IDL compiler. We compile the file with the LDOS IDL

compiler, which generates declarations and definitions for the classes AccumulatorServer and AccumulatorReference, and a macro for declaring the class AccumulatorImpl. The only class we need concern ourselves with now is AccumulatorImpl, which is the actual implementation of the class. First, we shall declare our implementation:

```
DECL_CLASS_Accumulator() {
DECL_IDL_GENERATED_Accumulator()
private:
      unsigned sum;
};
```

The DECL macros generate the appropriate declarations. There is no need to declare anything in the IDL file, which why we only add the declaration for the internal state, the sum. Next, we define the member functions:

```
void AccumulatorImpl::Reset()
{
      sum=0;
}

void AccumulatorImpl::Add(unsigned x)
{
      sum+=x;
}

unsigned AccumulatorImpl::Get()
{
      return sum;
}
```

That concludes the programmer's task for making Accumulator a class that LDOS can use. Notice the similarity to writing a header file (the IDL file) and an implementation file. To use the class locally, simply instantiate a server, and a reference to it, and make calls via the reference:

```
AccumulatorServer acc_s;
AccumulatorReference acc(&acc_s);

acc.Reset();
acc.Add(5);
unsigned x=acc.Get();
```

Notice that the AccumulatorReference acts as a pointer to the implementation. However, it is a very smart pointer, as we shall see.

Making A Remotely Accessible Accumulator

To make the accumulator accessible from a remote machine, we must add a network interface to it. To do this, we use an interface option, which is an LDOS extension to IDL:

```
interface Accumulator [TCP] {
```

In this case, we have endowed the Accumulator with a TCP/IP interface. Similarly, we can create UDP, HTTP (with HTML forms), and IP Multicast interfaces. After recompiling, we can instantiate an accumulator object in this way:

```
AccumulatorServer acc_s(portnum);
```

where portnum is the port number at which the server will reside. The object is internally multithreaded and is immediately available both locally and remotely.

A reference to this object can be instantiated and used on any machine on the network:

```
AccumulatorReference acc(hostname,port,objectid);

acc.Reset();
acc.Add(5);
unsigned x=acc.Get();
```

Notice that this differs from using a local server only in how the reference is instantiated. Instead of a local pointer, a hostname, port, and object identifier are supplied. The hostname and port identify a network interface, and the objectid identifies an object served by that interface. By using the TCP interface option in the IDL, we have requested a private interface. For a private interface, the objectid is ignored and should be zero.

Nannyed Objects

A network interface can be shared by many objects by using an ObjectNanny. We can remove the TCP interface option from the accumulator IDL (ie, use "interface Accumulator {" instead of "interface Accumulator [TCP] {") and instantiate two AccumulatorServers that share a single network interface in this way:

```
AccumulatorServer acc_s1, acc_s2;  // Two accumulators, no server
ObjectNanny on(tcpportnum, udpportnum, httpportnum);

ObjectID id1=on.AddObject(&acc_s1);
ObjectID id2=on.AddObject(&acc_s2);
```

The ObjectNanny supplies the network interface and routes requests to the appropriate object based on the request's objectid.

Describing Object State in IDL

If the state of the object and what state each method uses is declared in the IDL file, LDOS can provide many additional services than just RPC. State declarations are an LDOS extension to CORBA IDL. For the accumulator, we can rewrite the IDL thus:

```
interface Accumulator {
   state {
      unsigned long sum;
   } all_state;
   void          Reset() writes {all_state};
   void          Add(in unsigned long x) modifies {all_state};
   unsigned long Get() reads(all_state};
};
```

and we can then remove the state declaration from the implementation:

```
DECL_CLASS_Accumulator() {
DECL_IDL_GENERATED_Accumulator()
};
```

The state is the union of the contents of all the state declarations. By partitioning the object state into several state declarations, the programmer can specify at an arbitrarily fine granularity how he access the state of his object.

Serializability, Mobility, and Persistence

An class whose state is declared in IDL can request that it be made serializable:

```
interface Accumulator [Serializable] {
```

This means that instances of this class can save themselves to and restore themselves from LDOS streams:

```
AccumulatorServer x,y;

x.Serialize(somestream);        // x saves itself
y.UnSerialize(somestream);  // y restored from x
```

LDOS streams include network streams, file streams, and memory streams (buffers). Serializable objects can also be persistent (exist longer than the program that created them), and mobile (able to move from site to site). These features are expressed via interface options:

```
interface Accumulator [Serializable, Persistent, Mobile] {
```

Note that persistence and mobility are not currently implemented. However, the serialization interface and streams can be used for this purpose currently.

Replicable Objects

An object with no state can be replicated. Replicability is an interface option:

```
interface Adder [Replicable] {
      unsigned long Add(unsigned long x, unsigned long y);
};
```

To treat multiple AdderServers as a single, replicated object, a group is created:

```
GroupID gid=theGM->NewGroup();
```

and AdderServers join it:

```
AdderServer add_s(tcpport);
ObjectAddress add_oa={ToIPAddress(gethostname()), tcpport, udpport,
                      httpport, objectid};
theGM->AddAddress(gid,&add_oa);
```

To create a reference to a replicated object, we execute:

```
AdderReference ar(theGM,gid);  // identify the group manager, and the group on that manager.
```

Now, when a call is made via the reference, the mapping of that call to a specific instance is determined by a “member selector”. The programmer can install his own member selector either at compile time by subclassing AdderReference and overriding AdderReference::MemberSelector() or at run time by using the InstallExternalMemberSelector call:

```
unsigned mem_select(GroupMangerReference *gm,  unsigned max_choice);

ar.InstallExternalMemberSelector(mem_select);
```

Distributed Objects

Distributed objects are objects with state which may have more than one instance. In order to make this possible, relevant components of object state are moved by the LDOS run-time to satisfy calls. To create such an object, we rely on several IDL extensions. We tag its interface spec with the Distributed option, specify its state, and define how the methods in the interface use the object's state. Here is a simple example of two accumulators wrapped in a single object:

```
interface TwoAccumulators [Distributed] {
   state {
      unsigned long acc1;
   } acc1;

   state {
      unsigned long acc2;
   } acc2;

   void          Reset1() writes {acc1};
   void          Add1(in unsigned long x) modifies {acc1};
   unsigned long Get1() reads(acc1};

   void          Reset2() writes {acc2};
   void          Add2(in unsigned long x) modifies {acc2};
   unsigned long Get2() reads(acc2};
};
```

We write the implementation of TwoAccumulators in precisely the same way as for the single accumulator example above. To create an instance, we create a group, just like with a replicated object:

```
GroupID gid=theGM->NewGroup();
```

and TwoAccumulatorsServers join it:

```
TwoAccumulatorsServer acc2_s(tcpport);
ObjectAddress acc2_oa={ToIPAddress(gethostname()), tcpport, udpport,
                       httpport, objectid};
theGM->AddAddress(gid,&acc2_oa);
```

We must also initially assign the state to one instance:

theGM->SetAllStateOwnership(gid,membernumber);

To create a reference to a replicated object, we execute:

```
TwoAccumulatorsReference acc(theGM,gid);
```

just like for a replicated object, and, similarly, we can install our own member selector as well. The LDOS run-time, and the code the IDL compiler generated conspire to assure that all state necessary to complete a call is accessible, in effect performing like a small time distributed shared memory system.