3F6 - Software Engineering and Design

Handout 11

Distributed Systems II

With Markup

Ed Rosten

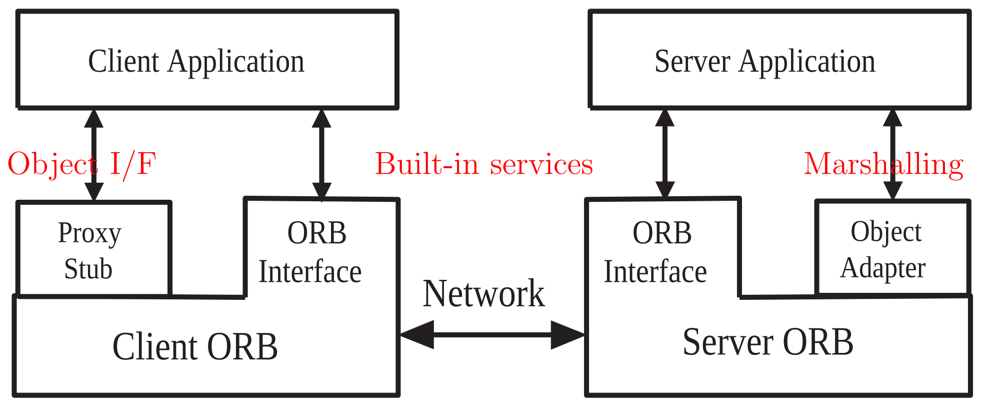
Contents

1. Mapping IDL to C++
2. Client operation
3. IORs and the Naming Service
4. Writing a Server
5. Factories and Load Balancing

Copies of these notes plus additional materials relating to this course can be found at: `http://mi.eng.cam.ac.uk/~er258/teaching`.

CORBA Programming in C++

Review of basic client-server operation in CORBA

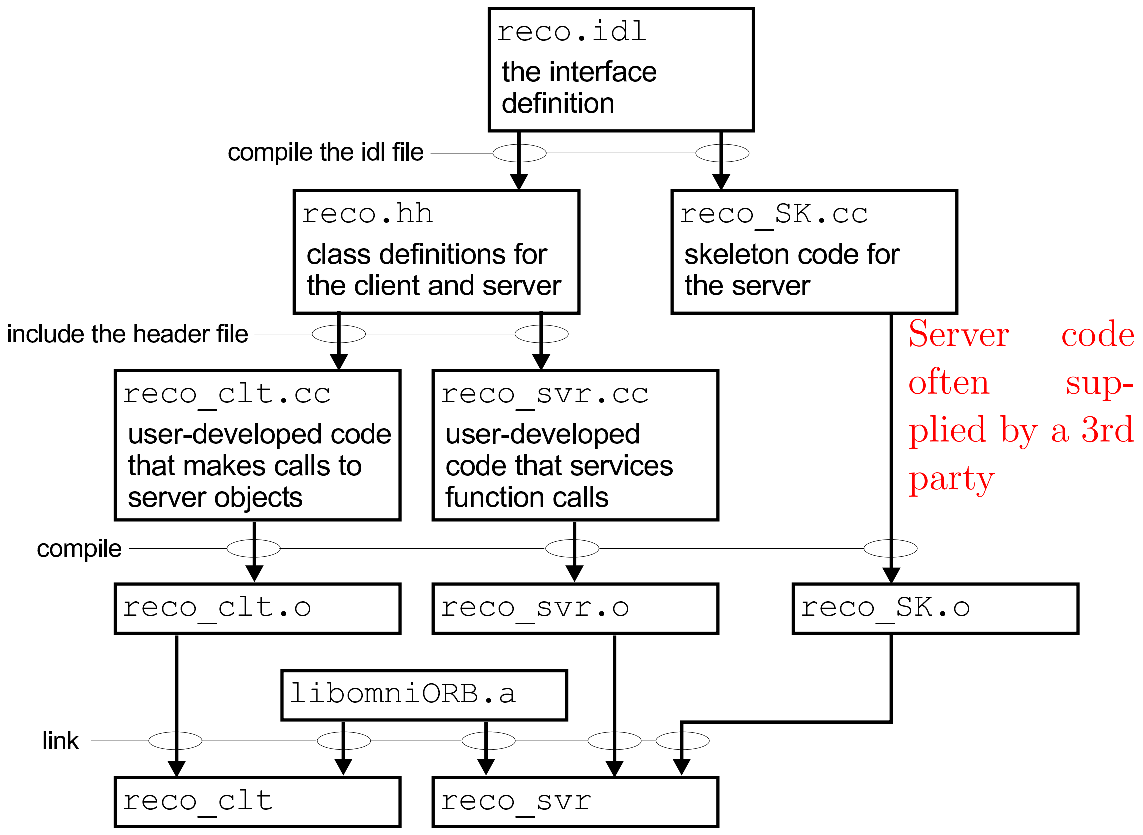


1. The client locates the server and creates a smart-pointer to the server object.
2. The client calls a server method via the smart-pointer.
3. The method stub in the proxy class conveys the request to the client's ORB.
4. The client's ORB transmits the request to the ORB linked to the server.
5. The server's ORB dispatches the request to the object adapter that created the target server object.
6. The object adapter dispatches the request to the server object.
7. The object returns results via the same route.
8. Steps 2 to 6 can be repeated. When the smart-pointer is destroyed, the server releases the target object.

Mapping the IDL into C++

The first step is to compile the IDL interface specification into the required target language. This creates a set of files which form the starting point for the implementation.

For example, suppose that the Recogniser interface is stored in the file `reco.idl`


reco.idl
the interface definition
compile the idl file
reco.hh
class definitions for the client and server
reco_SK.cc
skeleton code for the server
include the header file
reco_clt.cc
user-developed code that makes calls to server objects
reco_svr.cc
user-developed code that services function calls
Server code often supplied by a 3rd party
compile
reco_clt.o
reco_svr.o
reco_SK.o
libomniORB.a
link
reco_clt
reco_svr

NB: the details of IDL compiler operation will vary for different CORBA implementations. The description here is for omniORB.

Client Operation

To access a remote CORBA object of type `Recogniser`, a client must take the following steps

1. Create an ORB

   ```
   CORBA::ORB_var orb = CORBA::ORB_init(0,0,"omniORB4");
   ```

2. Locate the remote object and create a generic pointer to it.

   ```
   string IORstr = "...";  // text version of an IOR
   CORBA::Object_var obj = orb->string_to_object(IORstr);
   ```

3. Check that the object we have located matches the IDL interface specification (dynamic type-casting).

   ```
   Recogniser_var rec = Recogniser::_narrow(obj);
   if (rec == NULL) error();
   ```

4. the remote `Recogniser` object is now ready for use

   ```
   rec->init(English);   rec->SetBeam(250.0);
   ```

Two questions remain:

- How do we access IDL types from C++ ?
- What is an IOR and how do we obtain them?

IDL to C++ Mapping - Simple Types

For simple types, there is a simple one-one mapping. For example,

IDL	C++
`short`	`CORBA::Short`
`long`	`CORBA::Long`
`unsigned short`	`CORBA::UShort`
`float`	`CORBA::Float`
`boolean`	`CORBA::Boolean`
`etc`	

CORBA types and C++ types are assignment compatible and share the same operators. For example,

```
CORBA::Float beam = 200;    // CORBA type
float inc = 50;             // C++ type
...

beam += inc;                // implicit type conversion
rec->SetBeam(beam);
```

However, for portability reasons, it is best to avoid mixing them if possible.

IDL to C++ Mapping - Structured Types

Fixed size structures are similarly straightforward. eg the IDL definition

```
   typedef short Audio[AudioMax];
```

becomes

```
   typedef CORBA::Short Audio[AudioMax];
```

in C++ which can then be manipulated and accessed as for any other C++ array.

Variable length types are a little more complex.

```
  typedef sequence<string> Words;
```

creates a class called `Words` with a number of operations to create, extend, shrink and destroy sequences. The following example illustrates some of the main operations:

```
    Words  w;      // create empty sequence
    w.length(2);   // w is now a sequence of 2 empty strings
    w[0] = CORBA::string_dup("hello");   // assign values
    w[1] = CORBA::string_dup("world");

    int len = w.length();    // len = 2
    w.length(1);   // length now 1, string "world" deleted
```

Interoperable Object Reference(IOR)

An IOR is the CORBA equivalent of a URL. It contains

- a *RepositoryID* - this is the name of the object used by the ORB. For example, `IDL:Epic/Recogniser/V1.0`
- host IP number of the object's server
- port number on which the server listens for requests

The ORB provides functions to convert an IOR to a string form which can be written out and passed to clients who can then convert it back to an IOR. Eg. In the server,

```
Recogniser_impl rec_servant;   // create a Recogniser object
// then create a reference to it
Recogniser_var rec = rec_servant._this();
// convert this reference to a string
string str = orb->object_to_string(rec);
// and write it out
cout << str << endl;
```

On the client, if this stringified IOR is stored in "ior.txt", it can be reloaded

```
ifstream ifs("ior.txt");
string ior;   ifs >> ior;
CORBA::Object_var obj = orb->string_to_object(ior);
```

However, this is all rather clumsy. So CORBA also provides a *Naming Service.*

The Naming Service

Using text files to transmit object references can be cumbersome (and, at times, very difficult). An alternative approach is to use the *Naming Service* built into CORBA:

- server registers its object(s) with the Naming Service giving each a name
- clients request the Naming Service for object references by sending the registered name
- Naming Service returns a reference to the object

The Naming Service provides a hierarchical naming system so that objects appear as if they are stored in a hierarchical filing system. This simplifies the organisation of very large systems, and it helps to avoid name clashes.

The Trading Service

CORBA also provides a *Trading Service* which is a search engine for CORBA objects. It stores object references according to their attributes and services.

Writing the Server Code

The basic process is as follows:

1. Implement the Recogniser Object
2. Create an ORB
3. Register the Recogniser Object with the ORB
4. Start the ORB

Once started, the ORB waits for requests to the Recogniser's methods.

For each request, the ORB

1. unpacks the parameters
2. calls the specified object method
3. packs any results and returns them to the caller

Note that the packing/unpacking of remote method call parameters is called *marshalling*

Implementation of Recogniser Object

All of the CORBA related code is generated automatically by the IDL compiler (stored in `recoSK.cc`). The interface between CORBA and the actual object is via the *Portable Object Adapter* which is represented in `reco.hh` by a virtual `POA_Recogniser` class. The code that has to be written is simply a concrete class which implements the actual object:

```
#include "reco.hh"

class Recogniser_impl : public virtual POA_Recogniser {
public:
  virtual void init(Lang l);
  virtual void setBeam(CORBA::Float beam);
  virtual Words* listen(const Audio a);
};

void Recogniser_impl::init(Lang l)
{
   // actual implementation of init method
}

void Recogniser_impl::setBeam(CORBA::Float beam)
{
   // actual implementation of setBeam method
}

Words* Recogniser_impl::listen(const Audio a)
{
   // actual implementation of listen method
}
```

Factories in CORBA

As described so far, if the server received multiple requests for a Recogniser object, it would return the same actual object to each of them.

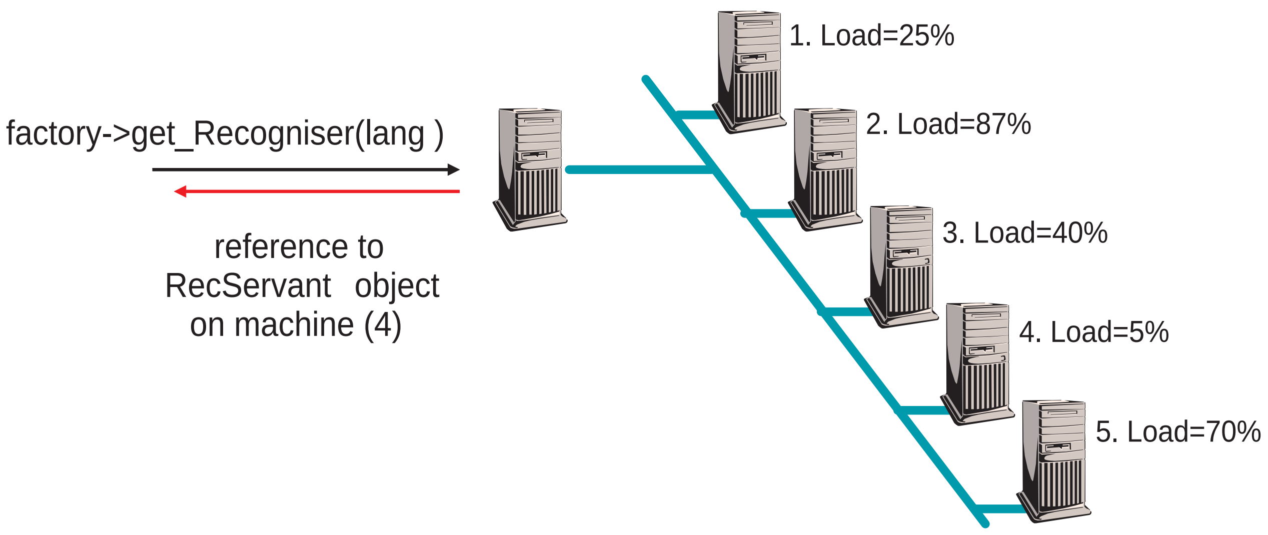
To avoid this, a CORBA server typically follows the *Factory design pattern*. Here is the revised interface definition for our Epic Recogniser using the Factory pattern.

```
module EpicV1 {

   const long AudioMax = 100000;  // max size of audio chunk

   enum Lang {English, French, German, Chinese};

   typedef short Audio[AudioMax];
   typedef sequence<string> Words;

   interface RecServant {
      void setBeam(in float beam);
      // adjust the beam width to control the search

      Words listen(in Audio a);
      // invoke the recogniser to convert the segment of
      // audio in a into a sequence of words
   };

   interface RecogniserFactory {
      RecServant get_Recogniser(in Lang l);
   };

};
```

Load Balancing

Now that the Recogniser server can provide multiple Recogniser instances, issues of compute load must be considered. As shown earlier, each instance of `RecServant` will consume approx 15% of a cpu. Hence, for large scaleable systems, the Recogniser Factory must have access to a stack of computers, and allocate instances on different machines depending on load.

Graphically this is illustrated as follows:

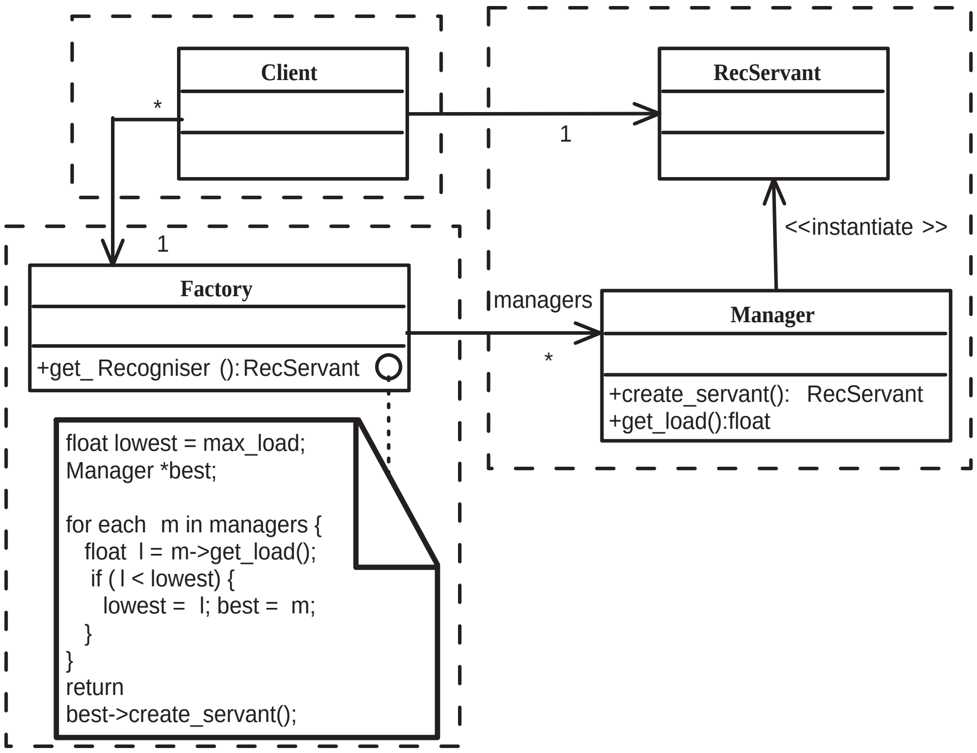

factory->get_Recogniser(lang)
reference to RecServant object on machine (4)
1. Load=25%
2. Load=87%
3. Load=40%
4. Load=5%
5. Load=70%

Note that once the factory server has returned the reference to the actual `RecogniserServant`, all future communication with the client goes directly to the machine hosting the instantiated object.

Implementation of Load Balancing

This is a simple extension of the Factory design pattern.

- each stack computer runs a server which maintains a *Manager* object which acts like a mini-factory creating object instances on its own machine. It also has a facility for monitoring the cpu load.
- the primary Factory access is now via a Factory object which scans each manager to find the one which is least heavily loaded. It then asks the least-loaded manager to instantiate an object and return a reference to it. This reference is then returned to the client.



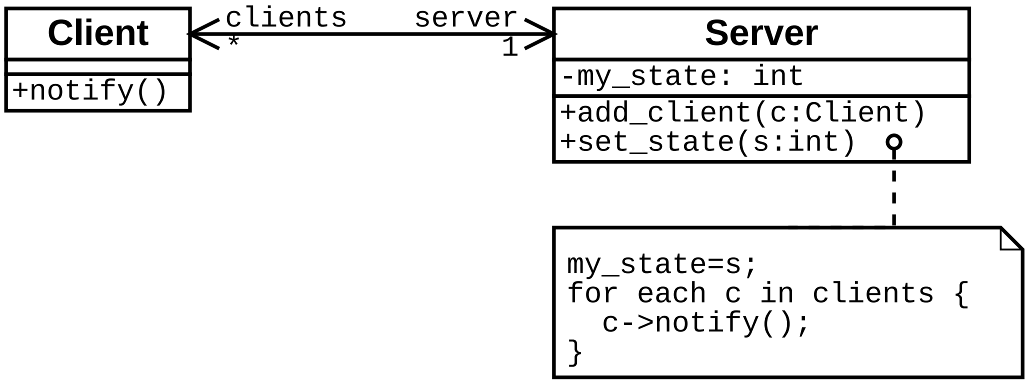
```
float lowest = max_load;
Manager *best;

for each  m in managers {
   float  l = m->get_load();
    if ( l < lowest) {
      lowest =  l; best =  m;
   }
}
return
best->create_servant();
```

Note that the load balancing is invisible to the client. Hence, this type of solution can be scaled over time to meet increasing demand with no changes to the client.

Client-Server Relationships

The client-server relationship is not necessarily static. In any system, the roles of client and server may change.

Example a server wishes to notify all of its clients that the data has changed. This can be done by giving each 'client' their own interface (thus making them a server!):



In this example, each client registers itself with the server by calling `add_client`, passing their smart pointer. The server keeps a list of these and when the state changes, it can call the `notify` method on each client. The IDL for this is simple:

```
interface Client {
  void notify();
}
interface Server {
  void add_client(in Client c);
  void set_state(in long s);
}
```

(This is an example of the *observer* design pattern.)

Footnote: Smart Pointers

Smart pointers such as `Recogniser_var` are crucial to the clean implementation of the proxy design pattern. The key idea is to wrap a class in some other class in order to redefine the behaviour of certain operations.

The most common example of this is to ensure that allocated objects are deleted automatically. For example, suppose that `Object` is a regular class. Standard use would be

```
void foo() {
   Object *p = new Object();
   p->DoSomething();
   delete p;   // but this is easy to forget!
}
```

Using a smart pointer you would do something like

```
void foo() {
   SmartPtr p(new Object());
   p->DoSomething();
}
```

Here is a simple implementation of `SmartPtr`

```
class SmartPtr {
   Object * ptr;  // the actual object
public:
   SmartPtr(Object *p=0):ptr(p){}
   ~SmartPtr(){delete ptr}
   Object & operator*() {return *ptr;}
   Object * operator->() {return ptr;}
};
```

Summary

- *Distributed systems* spread the processing over several computers, which has advantages in processing power, geographical access, and usability.
- Distributed systems are complex and use *middleware* to handle the problems of communication between computers.
- CORBA is an object-oriented middleware specification. It uses *proxy objects*, and an *interface definition language (IDL)* to provide location transparency and language transparency. *Interfaces* define the functions that can be called on objects.
- It is usual to talk about *clients* and *servers.* However, clients can also have their own interfaces, which means they act as servers themselves.
- *Factories* can be used to create objects and provide load-balancing, and *derived interfaces* allow interfaces to be modified without breaking existing clients.