3F6 - Software Engineering and Design

Handout 10

Distributed Systems I

With Markup

Ed Rosten

Contents

1. Distributed systems
2. Client-server architecture
3. CORBA
4. Interface Definition Language (IDL)
5. Interface inheritance
6. Upgrade and Maintenance

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

Distributed Systems

Distributed Systems are systems where the processing is spread over several computers. Virtually all large computer systems are now distributed systems, and this approach brings many advantages:

- **More processing power**
  e.g. Grid computing, Google, seti@home
- **Scalable**
  easy to grow and upgrade the system e.g. the world-wide web, Google
- **Resource sharing**
  e.g. printers, e-mail servers, databases
- **Fault tolerance**
  errors in one part of the system do not necessarily affect the rest of the system
- **Simplifies software engineering**
  simplifies the design, implementation and maintenance of naturally concurrent processes

Examples of Distributed Systems

- Distributed databases e.g. air-line reservation systems
- Banking systems
- Web-based systems
- Car engine management systems
- Computer operating systems

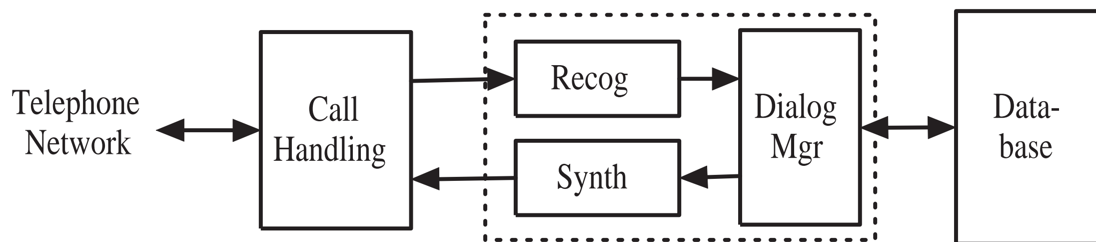
Note that there is not necessarily a one-one mapping between the logical system as expressed by the software and the physical system as implemented by the hardware.

- a physical processing unit can execute multiple programs (processes) simultaneously
- a modern computer "chip" can contain multiple processing units
- a computer motherboard can contain multiple computer "chips"
- a computer can contain multiple motherboards

A key aim of software engineering is to abstract the logical process structure from the physical hardware.

Case Study - Call Centre Automation

Consider a Call Centre automation system in which routine user inquiries are handled automatically using a speech recogniser to understand what the user is saying and a speech synthesiser to allow the system to speak back to the user.



Client Requirements:

- initial support for 4 simultaneous users
- capability to scale to 24 simultaneous users
- must provide 24x7 reliability
- must allow simple maintenance and upgrade of constituent software components

Software Component Requirements:

Component Name	Fixed Mem (MB)	Per Instance Mem (MB)	Load (%CPU)	Implementation Language
Dialog Manager	0	1	0.01	C++
Recogniser	40	20	0.15	C++
Synthesiser	1000	2	0.04	Java
Totals	**1040**	**23**	**0.2**	

Note: 1cpu can support 4 users.

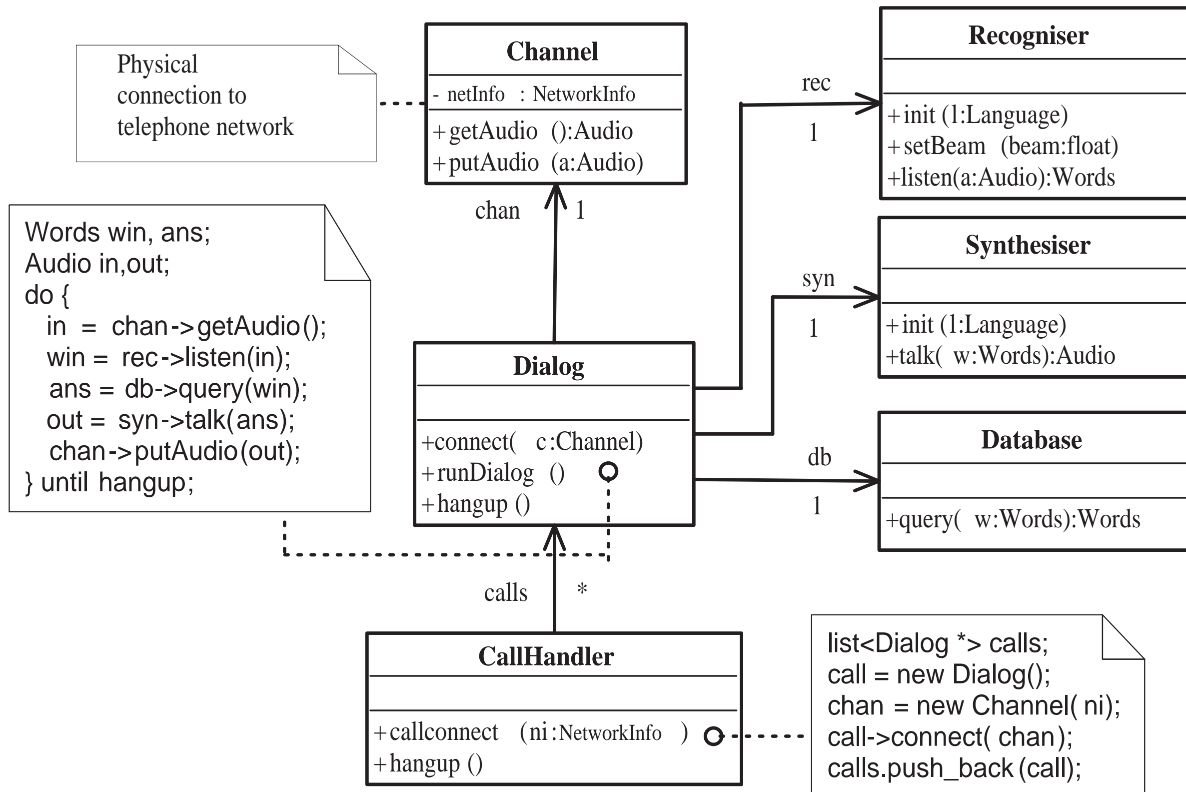
Two possible configurations to achieve 24+ simultaneous users:

	Config A		Config B	
CPU	# Insts	Memory (Mb)	# Insts	Memory (Mb)
1	24 s	1048	5 rs	1150
2	6 r	160	5 rs	1150
3	6 r	160	5 rs	1150
4	6 r	160	5 rs	1150
5	6 r	160	5 rs	1150
6	24 dm	24	25 dm	25

r = Recogniser   rs = Recogniser plus Synthesiser
s = Synthesiser   dm = Dialog Manager

Note that keeping the DM on a separate CPU allows further expansion by simply adding more cpus for syns and recs. A looks best but if tight coupling between Rec and Syn (eg sharing resource) then B is better

A basic software design might be as follows



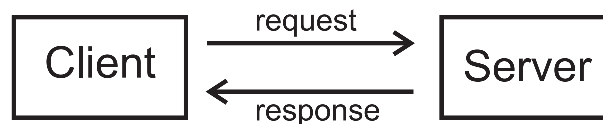
Note that this design will meet the initial requirement for 4 simultaneous users. But how do we add the flexibility to provide support for

- scaling to many simultaneous users
- providing 24x7 reliability
- interfacing software written in different languages
- upgrades to individual components

The answer is to design the Call Centre Automation as a distributed system.

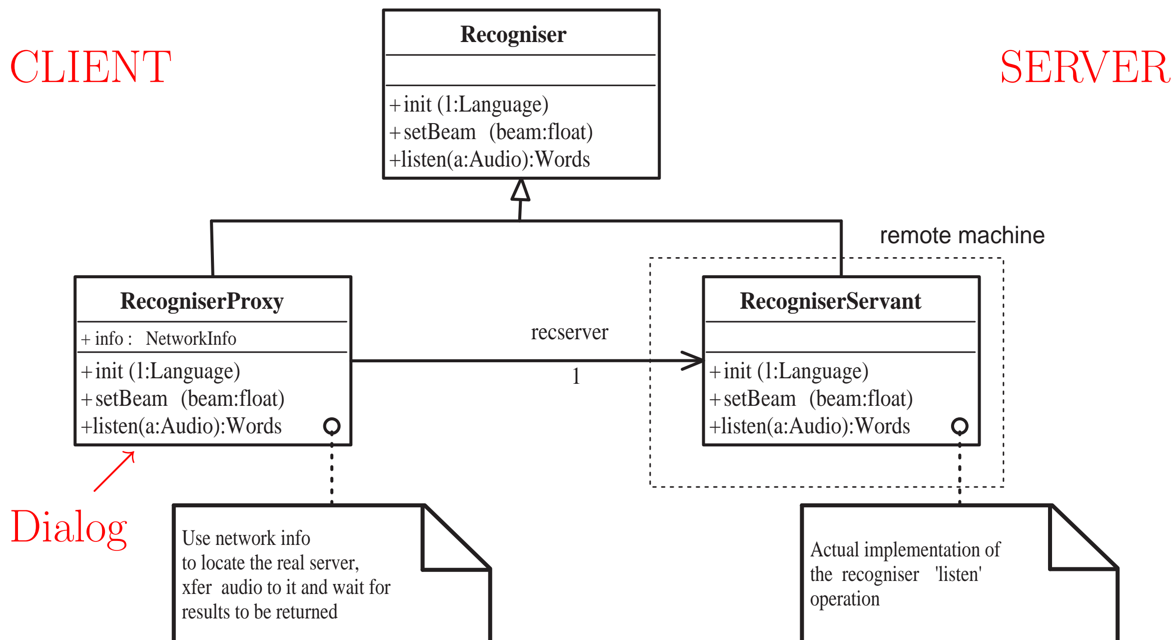
Client-Server Architecture

The simplest relationship in a distributed system is the client-server relationship. The server provides data and services to the client.



At the hardware level, the client and server are often located on different computers.

At the software level, we require *location transparency*. This is achieved using the Proxy Design Pattern.



However, implementation of remote object management is complex - *Middleware* can provide much of the support needed.

CORBA

CORBA (Common Object Request Broker Architecture) is a set of standards for middleware defined by the Object Management Group (OMG), a consortium of over 500 companies, including IBM, Sun, Hewlett-Packard, Oracle, Ford, Boeing and Xerox.

CORBA specifies standards for

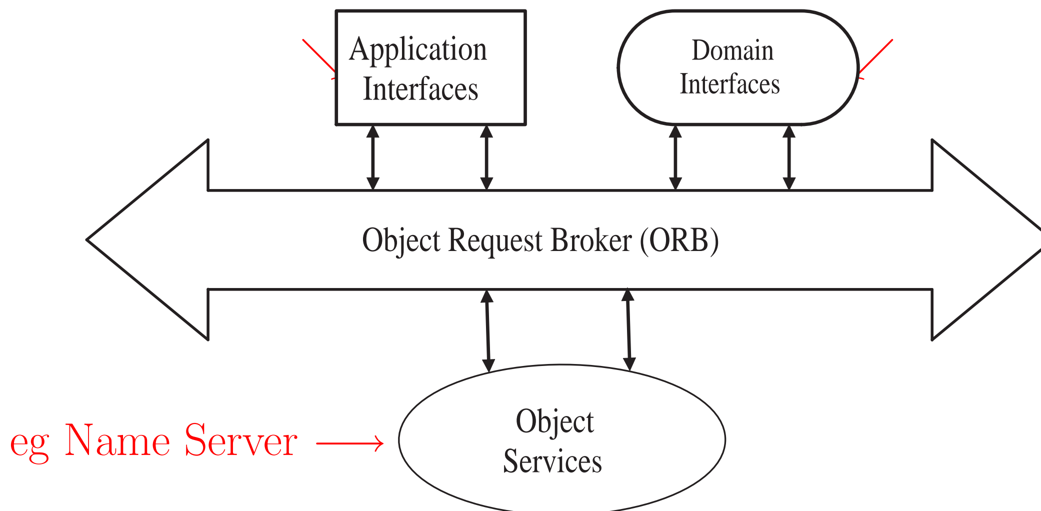
- describing the interface that an application presents to the network
- mapping that interface into C++, Java, Visual Basic and many other languages
- using that network interface to communicate between programs

There many implementations of the CORBA standard available. The 3F6 laboratory experiment uses omniORB, a freely-available version for C++ and Python. It supports over 15 different platforms, including Microsoft Windows, Mac OS X, Linux, and most other forms of Unix.

Object Request Broker

The Object Request Broker (ORB)

eg 3F6 Lab PostIt Server eg NHS Medical Records



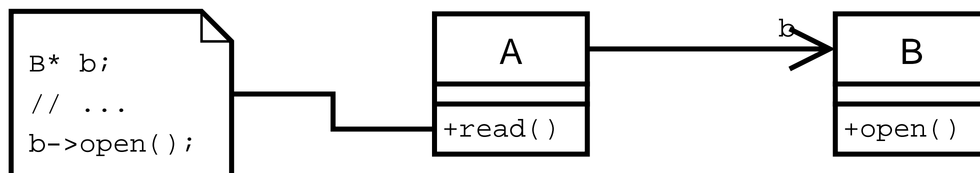
- enables communication between clients and objects
- provides location transparency
- provides access to a set of built-in object services

There are two classes of interfaces

- Domain interfaces - standards agreed by collaborating organisations and registered with the OMG.
- Application interfaces - developed for specific applications, interfaces are specific to each application.

Proxies in CORBA

Object-oriented programs consist of linked objects calling methods on each other. For example, here object `A` is connected to object `B`, and calling the method **`read`** on `A` results in the method **`open`** being called on `B`:

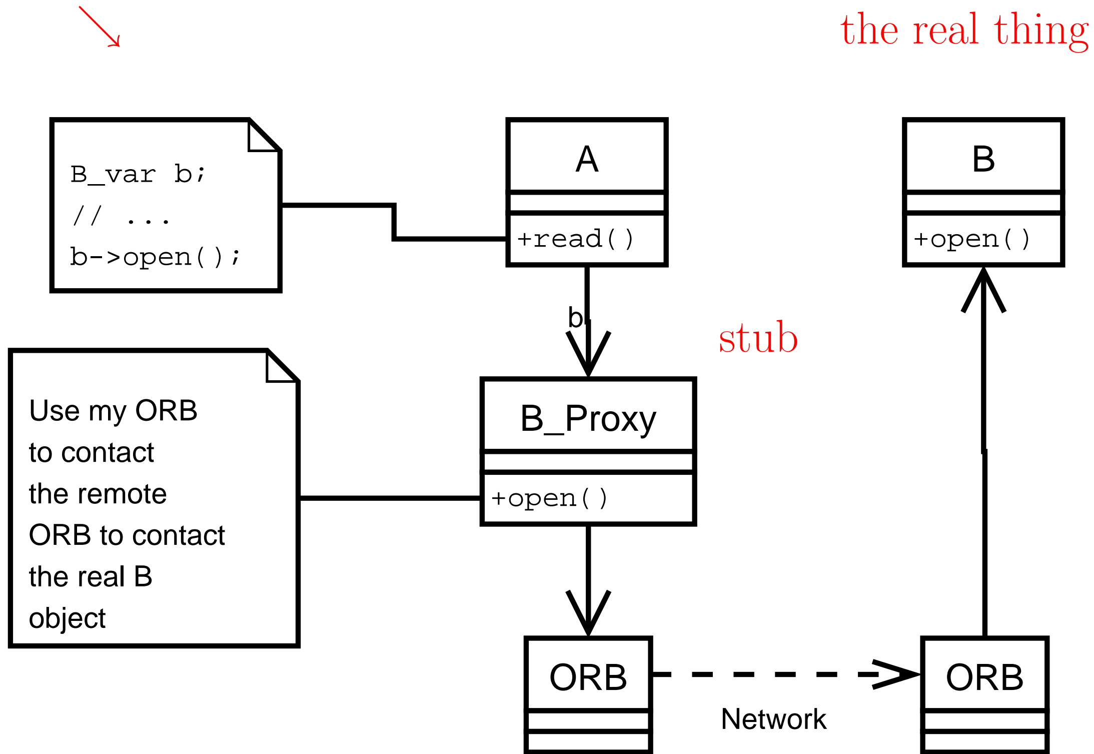


In a distributed application, we might want these objects to live on different computers. This would make calling `B::open` rather complex.

CORBA provides a remote proxy mechanism which provides the necessary *location transparency.*

It also makes programming simple by hiding the proxy class from the programmer. In C++, instead of using standard pointer references, CORBA provides *smart pointers*. These hide the proxy class and provide built-in reference counting so that the remote objects can be automatically deallocated when no longer required.

Special version of `B *b;`



Using this design pattern, `A` references `B` using a smart pointer `b` (of type `B_var`).

Method calls via `b->` are then directed to the proxy for `B`.

When `A` calls `open` on the `B_Proxy` object, this contacts the local ORB, which determines where the real `B` object lives. The ORB then makes a network connection to the ORB on `B`'s computer and passes the request on to it. The remote ORB then contacts the real `B` object, which services the request. The results (any return values) are passed back to `A` via the same mechanism.

The Interface Definition Language

Remote objects may be implemented in different programming languages.

CORBA provides a language independent mechanism for specifying an object's interface called the *Interface Definition Language (IDL).*

The basic structure of an IDL interface definition is as follows:

```
module ModName {
   // define constants
   const type ConstName = constant_expression;

   // define types
   typedef  type  Typename;

   // define interfaces
   interface ObjectName {

      // define local consts and types

      // define methods
      returntype functionName(mode Typename arg, ...);
      ...
   };
};
```

Mode allows efficient parameter xfer across network

Apart from the need to specify the *mode* (`in, out, inout`), the IDL is very similar to C++.

IDL Type Specification

Basic types:
has to be precise so that mixed language operation works properly

Type	Min Bits	Type	Min Bits
`(unsigned) short`	16	`float`	32
`(unsigned) long`	16	`double`	64
`char`	8	`octet`	8
`string`	–	`boolean`	1 (T/F)

Strings are variable length.

Enumerations, fixed-size arrays and structures are supported exactly as in C++

```
enum Colour {red, green, blue};
typedef short RGB[3];
struct Pixel {
   RGB rgb;
   short x; short y;
};
```

The IDL does not support pointers. Instead it provides *sequences* which can be used to define variable length arrays and recursive data structures

```
struct TreeNode {
   string nodeContents;
   sequence<TreeNode> children;
};
```

Case Study - Remote Recogniser Object

In the Call Centre Automation system, the Recogniser is implemented as a CORBA object to allow the design to be scaled to support an increasing number of simultaneous users. In addition, the recogniser itself is supplied by a 3rd party supplier called Epic. CORBA provides an effective mechanism for them to integrate their product into larger systems.

```
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 Recogniser {
      void init(in Lang l);
      // initialise ready for given language

      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 Inheritance

As in C++, IDL interfaces can inherit from existing interfaces.

For example,

```
module People {

   interface Person {
      void  init(in string name, in short age);
      short getAge();
      string getName();
   };


   interface Child : Person {
      void  init(in string name, in short age,
                 in string guardian);
      string getGuardian();
   };


};
```

Here the *derived interface* `child` redefines the `init` operation, inherits `getAge` and `getName` and adds a new operation `getMaidenName`.

Upgrade and Maintenance

A significant problem in Software Engineering is maintaining software when constituent components are upgraded. Usually, an upgrade will do two things:

1. add new features
2. fix existing bugs

For upgrades to be useful they should

- allow existing applications to continue to function correctly and take advantage of bug fixes without recompilation or changes to the client code.
- allow new applications to take advantage of the new features.

Using a derived interface solves both these problems: it ensures backwards compatibility and makes it simple to specify extensions.

Case Study - Recogniser Upgrade

Shortly after commissioning the first version of the Call Centre Automation system using *Epic V1*, Epic decides to add a new function called **`nextBest`** which when called after **`listen`** returns the next best matching sequence of words. At the same time, they fix some bugs and release a new version *Epic V2*. Adding a derived interfaces allows this upgrade to be done safely and efficiently:

```
module EpicV2 {

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

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

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

   interface Recogniser {
      // as before
   };

   interface Recogniser2 : Recogniser {
      // this new interface supports all of the existing
      // functionality - plus the following new function

      Words nextBest();
      // return next best word sequence
   };

};
```

NB Original interface preserved but implementation improved, and functionality extended