Distributed objects

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

Evolution of client-server systems

- In distributed systems the asymmetric client and server roles show up commonly:

  client active part, initiates conversation;
  server passive part; waits for requests and replies.

- Initially, request-reply patterns were codified directly with some low level mechanism; e.g., sockets.
- When procedural languages were common, RPCs were introduced to hide distribution and facilitate programming.
- The success of object-oriented languages motivated the appearance of the distributed objects paradigm.

Objects, remote invocation and distributed systems

- Object as an instance of a distributed service;
- Remote method invocation to access the service;
- Some relevant characteristics of objects revisited;
  - separation between interface and implementation;
  - state encapsulation;
  - identity;

Separation between interface and implementation

Separation is useful considering heterogeneity of architectures and languages in distributed systems:

- fashionable languages change with time;
- hardware architectures evolve;
- different components end up written in different languages and run in different platforms;
- interface serves as a contract between parts;
- using neutral Interface Definition Languages allows a service to be described independently from languages used.
- homogeneous distributed systems can avoid the use of IDLs (e.g., if using only Java RMI).

Interface inheritance and the subtype relationship

- An interface *S* that inherits from interface *I* is a subtype of *I* if it offers at least the same service, being able to be used by all clients that need an *I*.
- Allows extending existing services with new features.
- A service can be seen by different clients as having diferent interfaces:
  - an old client knows and uses the original interface;
  - new clients know the derived interface and exploit extra features.
- This is completely orthogonal to implementation inheritance or even to the use of OO languages in the implementation.

Implementations and class inheritance

- Class inheritance in OO languages is useful to reuse code while implementing new features or improving the implementation.
- Much less important to distributed object systems than the existence of interfaces and interface inheritance.
- As an example, in CORBA we can have multiple interface inheritance together with:
  - implementations in languages that support only single class inheritance, like Java;
  - implementatios in non OO languages, like C.

Subtype polymorphism

- The subtype relation allows a client that needs an interface to reference objects that implement some subtype of that interface.
- The same context can be client of different objects, derived from a common interface.
- It allows polymorphic containers, that reference subtypes of a given interface, as the classic vector of shapes:

```
for f in array_of_shapes:
    f.draw()
```

- This concept fits naturally to distributed objects.

Static versus dynamic binding

- In OO languages we have two ways of binding which method is executed for a given operation invocation:

  static binding the method depends uniquely on the type of the reference. It is not natural and has dangerous semantics. Motivated mainly by efficiency reasons; the default in C++ or C#.

  dynamic binding the method depends on the class of the referenced object. The natural and desired way to associate operations with methods. Used in Java, or can be chosen in C++ or C# through `virtual`.

- In distributed object systems the choice of method is local to the implementation; we have the desired semantics of dynamic binding.

State encapsulation

- The state of an object is manipulated by a thread that handles a method invocation. In the more pure model the thread can:
  - access the state of the receiver;
  - perform invocations on other objects;
- Global state is undesirable; e.g., class (e.g., static) variables.
- Protection models used in different languages may be more or less suitable to distributed systems:
  - Object based versus class/module based;
  - Object based are suitable; e.g., Smalltalk;
  - Class based cause problems; e.g, Java/C++;

Object identity

- Identity is what distinguishes objects, even if they have the same state.
- One advantage of distributed object systems when compared with traditional RPC is the ease of passing and returning references to distributed objects.
- References encapsulate object (and therefore, service) identity.
- In classic RPC systems the identity of services cannot be passed easily around as a normal type of data; applications end up encoding identity explicitly.
- In distributed systems, identity is useful for:
  - service discovery;
  - distinguish instances of some type of service;
  - promote location transparency.

Object identity

In a distributed object system we have to ways of handling identity:

- uniformity:
  - all objects can be used remotely;
  - existing applications could be easily ported;
  - however, costly in terms of performance;
- distinction between local and remote objects:
  - only remote objects have global identity and allow remote invocation;
  - implies bigger changes when porting applications;
  - allows better performance through less transparency and more awareness of distribution;
  - normally adopted; e.g., in Java RMI.

Parameter passing

- Fundamental aspect of distributed object systems.
- Options: copy versus reference.
- The general case contemplates distinction between local and remote objects. Basic principles:
  - remote objects are passed by reference, the are globally accessed;
  - local objects cannot be accessed remotely and are copied.
- Impact in terms of modification of the semantics of programs.

Remote object passing

- For remote objects a remote reference is passed: a representation that encapsulates the global object identity.
- A reference must contain all information that allows the receiving context to perform a remote invocation upon the object.
- References to remote objects point physically to some local object: a proxy for the remote object, instanced from a stub class.
- When passing a remote object, a proxy is passed and stored in the receiving host, which uses it for remote invocations.

Stubs and skeletons

- To support transparent remote invocations on distributed objects, stubs and skeletons are used.
- An instance of a stub receives the invocation, serializes arguments an sends the invocation to a communication module.
- In the receiver, a dispatcher forwards the invocation to an instance of a skeleton, that identifies the method, recovers parameters, invokes the method, receives the result and sends the reply back.
- Stubs and skeletons are normally generated by an IDL compiler.
- Skeletons can be avoided by having dynamic dispatch mechanisms, and stubs can be avoided if client and object implementation reside in the same host.

Proxy sharing and object identity

- A distributed object system may keep tables which map remote object identity to local proxies.
- This allows instancing a proxy only when no previous proxy exists on that host for the given remote object.
- It allows sharing of the proxy object by local references to the remote object.
- The guarantees concerning proxy sharing impact how identity comparison for remote objects can be performed.

Passing local objects

- Local objects are passed by copy as a reference makes no sense in the remote host.
- In the receiving end new objects are created, and kept locally.
- Parameter passing for local objects must be thought of as passing values and not object identities in the traditional sense.

Transitive local state

- The transitive local state of an object is made up of the state in the instance variables, and in the transitively reachable local objects.
- The transitive state serves to define a self-contained unit.
- In a remote invocation the transitive state of local objects in arguments is passed.
- It allows passing a sizeable piece of information at once, avoiding a too fine granularity in some cases.

Modification of semantics: invocations over the argument

- Passing local objects implies significant modifications in semantics compared with the traditional model.
- In the traditional model, the receiver can invoke an operation which modifies the argument.
- When passing a local object in a remote invocation, the receiver modifies a local copy, and the update is not seen by the client.

Client:

```
r.met(l)
```

Implementation:

```
met(p){ p.update(); this.i = p;}
```

Modification of semantics: attempt to solve problem

- To attempt to correct the modification, the interface can be modified, making the method return the modified object as result:

Client:

```
nl = r.met(l)
l = nl;
```

Implementation:

```
met(p) {
  p.update();
  return p;
}
```

- The modified object is serialized back to the client, which stores it in the original variable.
- This does not mean that all impact is corrected and that the application behaves as originally . . .

Modificações na semântica: problemas devido a aliasing

- The problems resulting from the modification of semantics is made worse if there is aliasing to the local object: when it is shared in the client context.

Client:

```
nl = r.met(l)
l = nl;
```

Implementation:

```
met(p) {
  p.update();
  return p;
}
```

- Even though *l* references the new object *L''*, other variables in the client context, like *v*, still reference the original object *L*.

Attribute access and lookup operations

- Some care should be taken when writing distributed object applications, or when porting existing applications:
  - avoid fine grained attribute access operations (get/set);
  - make available coarse grained operations with richer abstractions.
  - explicitly store local copies of information obtained remotely, when appropriate, to avoid repeated remote lookup operations.
- Unfortunate real-life example:

```
def notify_logout(u):
  for i in users:
    i.notify("User " + u.get_nick() + "logged out.")
```

Semantics under faults

- Distributed object systems typically try to offer at-most-once semantics.
- E.g., in Java RMI:
  - if a remote invocation completes with no exception, the method was executed exactly once;
  - if some exception is raised, it may have been executed once or not at all.
- Distributed object systems may distinguish between application exceptions and communication exceptions, often denoted by remote exceptions.

Remote exceptions

- Part of what makes distributed objects complicated is the handling of remote exceptions.
- A remote exception may occur due to several factors, e.g., sending arguments, or sending result back.
- Under a remote exception it may not be possible for the client to know if the method was executed.
- Traditional solutions:
  - design the interface to have idempotent operations;
  - design the application so that the remote object can detect duplicate requests.

Distributed garbage collection

- Programmers are used to having GC in OO languages.
- Extension of traditional GC to a distributed object system.
- Involves cooperation between local GC and the module that manages remote identifiers.
- Generally based on reference counting.
- Cicles are often not recicled.