Life Cycle of Proxies, Servants, and Ice Objects

It is important to be aware of the different roles of proxies, servants, and Ice objects in a system. Proxies are the client-side representation of Ice objects and servants are the server-side representation of Ice objects. Proxies, servants, and Ice objects have completely independent life cycles. Clients can create and destroy proxies with or without a corresponding servant or Ice object in existence, servers can create and destroy servants with or without a corresponding proxy or Ice object in existence and, most importantly, Ice objects can exist or not exist regardless of whether corresponding proxies or servants exist. Here are a few examples to illustrate this:

C++

```
{
    Ice::ObjectPtr obj = communicator?>stringToProxy("hello:tcp ?p 10000");
    // Proxy exists now.

} // Proxy ceases to exist.
```

This code creates a proxy to an Ice object with the identity `Hello`. The server for this Ice object is expected to listen for invocations on the same host as the client, on port 10000, using the TCP/IP protocol. The proxy exists as soon as the call to `stringToProxy` completes and, thereafter, can by used by the client to make invocations on the corresponding Ice object.

However, note that this code says nothing at all about whether or not the corresponding Ice object exists. In particular, there might not be any Ice object with the identity `Hello`. Or there might be such an object, but the server for it may be down or unreachable. It is only when the client makes an invocation on the proxy that we get to find out whether the object exists, does not exist, or cannot be reached.

Similarly, at the end of the scope enclosing the `obj` variable in the preceding code, the proxy goes out of scope and is destroyed. Again, this says nothing about the state of the corresponding Ice object or its servant. This shows that the life cycle of a proxy is completely independent of the life cycle of its Ice object and the servant for that Ice object: clients can create and destroy proxies whenever they feel like it, and doing so has no implications for Ice objects or servant creation or destruction.

Here is another code example, this time for the server side:

C++

```
{
    FileIPtr file = new FileI("DraftPoem", root);
    // Servant exists now.

} // Servant ceases to exist.
```

Here, the server instantiates a servant for a File object by creating a `FileI` instance. The servant comes into being as soon as the call to `new` completes and ceases to exist as soon as the scope enclosing the `file` variable closes. Note that, as for proxies, the life cycle of the servant is completely independent of the life cycle of proxies and Ice objects. Clearly, the server can create and destroy a servant regardless of whether there are any proxies in existence for the corresponding Ice object.

Similarly, an Ice object can exist even if no servants exist for it. For example, our Ice objects might be persistent and stored in a database; in that case, if we switch off the server for our Ice objects, no servants exist for these Ice objects, even though the Ice objects continue to exist — the Ice objects are temporarily inaccessible, but exist regardless and, once their server is restarted, will become accessible again.

Conversely, a servant can exist without its corresponding Ice object. The mere creation of a servant does nothing, as far as the Ice run time is concerned. It is only once a servant is added to the Active Servant Map (ASM) (or a servant locator returns the servant from `locate`) that the servant incarnates its Ice object.

Finally, an Ice object can exist independently of proxies and servants. For example, returning to the database example, we might have an Ice server that acts as a front end to an online telephone book: each entry in the phone book corresponds to a separate Ice object. When a client invokes an operation, the server uses the identity of the incoming request to determine which Ice object is the target of the request and then contacts the back-end database to, for example, return the street address of the entry. With such a design, entries can be added to and removed from the back-end database quite independently of what happens to proxies and servants — the server finds out whether an Ice object exists only when it accesses the back-end database.

The only time that the life cycle of an Ice object and a servant are linked is during an invocation on that Ice object: for an invocation to complete successfully, a servant must exist for the *duration of the invocation*. What happens to the servant thereafter is irrelevant to clients and, in general, is irrelevant to the corresponding Ice object.

It is important to be clear about the independence of the life cycles of proxies, servants, and Ice objects because this independence has profound implications for how you need to implement object life cycle. In particular, to destroy an Ice object, a client cannot simply destroy its proxy for an object because the server is completely unaware when a client does this.

Distributed object systems such as DCOM implement these semantics. However, this design is inherently non-scalable because of the cost of globally tracking proxy creation and destruction.

See Also

- The Active Servant Map
- Servant Locators
- Object Identity