Implementing a destroy Operation

As far as the Ice run time is concerned, the act of destroying an Ice object is to remove the mapping between its proxy and its servant. In other words, an Ice object is destroyed when we remove its entry from the Active Servant Map (ASM). Once the ASM entry is gone, incoming operations for the object raise `ObjectNotExistException`, as they should.

On this page:

Object Destruction and Concurrency

Here is the simplest version of `destroy`:

C++

```
void
PhoneEntryI::destroy(const Current& c)
{
    try {
        c.adapter?>remove(c.id);
    } catch (const Ice::NotRegisteredException&)
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);
    }
}
```

The implementation removes the ASM entry for the servant, thereby destroying the Ice object. If the entry does not exist (presumably, because the object was destroyed previously), `destroy` throws an `ObjectNotExistException`, as you would expect.

The ASM entry is removed as soon as `destroy` calls `remove` on the object adapter. Assuming that we implement `create` as we saw earlier, so no other part of the code retains a smart pointer to the servant, this means that the ASM holds the only smart pointer to the servant, so the servant's reference count is 1.

We use smart pointers in C++, which are analogous to object references in languages such as Java and C#.

Once the ASM entry is removed (and its smart pointer destroyed), the reference count of the servant drops to zero. In C++, this triggers a call to the destructor of the servant, and the heap-allocated servant is deleted just as it should be; in languages such as Java and C#, this makes the servant eligible for garbage collection, so it will be deleted eventually as well.

Things get more interesting if we consider concurrent scenarios. One such scenario involves concurrent calls to `create` and `destroy`. Suppose we have the following sequence of events:

1. Client A creates a phone entry.
2. Client A passes the proxy for the entry to client B.
3. Client A destroys the entry.
4. Client A calls `create` for the same entry (passing the same name, which serves as the object identity) and, concurrently, client B calls `destroy` on the entry.

Clearly, something is strange about this scenario, because it involves two clients asking for conflicting things, with one client trying to create an object that existed previously, while another client tries to destroy the object that — unbeknownst to that client — was destroyed earlier.

Exactly what is seen by client A and client B depends on how the operations are dispatched in the server. In particular, the outcome depends on the order in which the calls on the object adapter to `add` (in `create`) and `remove` (in `destroy`) on the servant are executed:

- If the thread processing client A's invocation executes `add` before the thread processing client B's invocation, client A's call to `add` succeeds. Internally, the calls to `add` and `remove` are serialized, and client B's call to `remove` blocks until client A's call to `add` has completed. The net effect is that both clients see their respective invocations complete successfully.
- If the thread processing client B's invocation executes `remove` before the thread processing client A's invocation executes `add`, client B's thread receives a `NotRegisteredException`, which results in an `ObjectNotExistException` in client B. Client A's thread then successfully calls `add`, creating the object and returning its proxy.

This example illustrates that, if life cycle operations interleave in this way, the outcome depends on thread scheduling. However, as far as the Ice run time is concerned, doing this is perfectly safe: concurrent access does not cause problems for memory management or the integrity of data structures.

The preceding scenario allows two clients to attempt to perform conflicting operations. This is possible because clients can control the object identity of each phone entry: if the object identity were hidden from clients and assigned by the server (the server could assign a UUID to each entry, for example), the above scenario would not be possible. We will return to a more detailed discussion of such object identity issues in Object Identity and Uniqueness.

Concurrent Execution of Life Cycle and Non-Life Cycle Operations

This section applies to C++ only.

Another scenario relates to concurrent execution of ordinary (non-life cycle) operations and `destroy`:

- Client A holds a proxy to an existing object and passes that proxy to client B.
- Client B calls the `setNumber` operation on the object.
- Client A calls `destroy` on the object *while Client B's call to* `setNumber` *is still executing*.

The immediate question is what this means with respect to memory management. In particular, client A's thread calls `remove` on the object adapter while client B's thread is still executing inside the object. If this call to `remove` were to delete the servant immediately, it would delete the servant while client B's thread is still executing inside the servant, with potentially disastrous results.

The answer is that this cannot happen. Whenever the Ice run time dispatches an incoming invocation to a servant, it increments the servant's reference count for the duration of the call, and decrements the reference count again once the call completes. Here is what happens to the servant's reference count for the preceding scenario:

1. Initially, the servant is idle, so its reference count is at least 1 because the ASM entry stores a smart pointer to the servant. (The remainder of these steps assumes that the ASM stores the *only* smart pointer to the servant, so the reference count is exactly 1.)
2. Client B's invocation of `setNumber` arrives and the Ice run time increments the reference count to 2 before dispatching the call.
3. While `setNumber` is still executing, client A's invocation of `destroy` arrives and the Ice run time increments the reference count to 3 before dispatching the call.
4. Client A's thread calls `remove` on the object adapter, which destroys the smart pointer in the ASM and so decrements the reference to 2.
5. Either `setNumber` or `destroy` may complete first. It does not matter which call completes — either way, the Ice run time decrements the reference count as the call completes, so after one of these calls completes, the reference count drops to 1.
6. Eventually, when the final call (`setNumber` or `destroy`) completes, the Ice run time decrements the reference count once again, which causes the count to drop to zero. In turn, this triggers the call to `delete` (which calls the servant's destructor).

The net effect is that, while operations are executing inside a servant, the servant's reference count is always greater than zero. As the invocations complete, the reference count drops until, eventually, it reaches zero. However, that can only happen once no operations are executing, that is, once the servant is idle. This means that the Ice run time guarantees that a servant's destructor runs only once the final operation invocation has drained out of the servant, so it is impossible to "pull memory out from underneath an executing invocation".

For garbage-collected languages, such as C# and Java, the language run time provides the same semantics: while the servant can be reached via any reference in the application or the Ice run time, the servant will not be reclaimed by the garbage collector.

See Also

- The Active Servant Map
- Object Identity and Uniqueness
- Servant Activation and Deactivation