Server Implementation Techniques

As mentioned in Servant Locators, instantiating a servant for each Ice object on server start-up is a viable design, provided that you can afford the amount of memory required by the servants, as well as the delay in start-up of the server. However, Ice supports more flexible mappings between Ice objects and servants; these alternate mappings allow you to precisely control the trade-off between memory consumption, scalability, and performance. We outline a few of the more common implementation techniques here.

On this page:

Incremental Server Initialization

If you use a servant locator, the servant returned by `locate` is used only for the current request, that is, the Ice run time does not add the servant to the Active Servant Map (ASM). Of course, this means that if another request comes in for the same Ice object, `locate` must again retrieve the object state and instantiate a servant. A common implementation technique is to add each servant to the ASM as part of `locate`. This means that only the first request for each Ice object triggers a call to `locate`; thereafter, the servant for the corresponding Ice object can be found in the ASM and the Ice run time can immediately dispatch another incoming request for the same Ice object without having to call the servant locator.

An implementation of `locate` to do this would look something like the following:

C++

```
Ice::ObjectPtr
MyServantLocator::locate(const Ice::Current& c, Ice::LocalObjectPtr&)
{
    // Get the object identity. (We use the name member
    // as the database key.)
    //
    std::string name = c.id.name;

    // Use the identity to retrieve the state from the database.
    //
    ServantDetails d;
    try {
        d = DB_lookup(name);
    } catch (const DB_error&)
        return 0;
    }

    // We have the state, instantiate a servant.
    //
    Ice::ObjectPtr servant = new PhoneEntryI(d);

    // Add the servant to the ASM.
    //
    c.adapter->add(servant, c.id);      // NOTE: Incorrect!

    return servant;
}
```

This is almost identical to the implementation seen in our earlier example — the only difference is that we also add the servant to the ASM by calling `ObjectAdapter::add`. Unfortunately, this implementation is wrong because it suffers from a race condition. Consider the situation where we do not have a servant for a particular Ice object in the ASM, and two clients more or less simultaneously send a request for the same Ice object. It is entirely possible for the thread scheduler to schedule the two incoming requests such that the Ice run time completes the lookup in the ASM for both requests and, for each request, concludes that no servant is in memory. The net effect is that `locate` will be called twice for the same Ice object, and our servant locator will instantiate two servants instead of a single servant. Because the second call to `ObjectAdapter::add` will raise an `AlreadyRegisteredException`, only one of the two servants will be added to the ASM.

Of course, this is hardly the behavior we expect. To avoid the race condition, our implementation of `locate` must check whether a concurrent invocation has already instantiated a servant for the incoming request and, if so, return that servant instead of instantiating a new one. The Ice run time provides the `ObjectAdapter::find` operation to allow us to test whether an entry for a specific identity already exists in the ASM:

Slice

```
module Ice {
    local interface ObjectAdapter {
        // ...

        Object find(Identity id);

        // ...
    };
};
```

`find` returns the servant if it exists in the ASM and null, otherwise. Using this lookup function, together with a mutex, allows us to correctly implement `locate`. The class definition of our servant locator now has a private mutex so we can establish a critical region inside `locate`:

C++

```
class MyServantLocator : public virtual Ice::ServantLocator {
public:

    virtual Ice::ObjectPtr locate(const Ice::Current& c, Ice::LocalObjectPtr&);

    // Declaration of finished() and deactivate() here...

private:
    IceUtil::Mutex _m;
};
```

The `locate` member locks the mutex and tests whether a servant is already in the ASM: if so, it returns that servant; otherwise, it instantiates a new servant and adds it to the ASM as before:

C++

```
Ice::ObjectPtr
MyServantLocator::locate(const Ice::Current& c, Ice::LocalObjectPtr&)
{
    IceUtil::Mutex::Lock lock(_m);

    // Check if we have instantiated a servant already.
    //
    Ice::ObjectPtr servant = c.adapter.find(c.id);

    if (!servant) {     // We don't have a servant already

        // Instantiate a servant.
        //
        ServantDetails d;
        try {
           d = DB_lookup(c.id.name);
        } catch (const DB_error&) {
           return 0;
        }
        servant = new PhoneEntryI(d);

        // Add the servant to the ASM.
        //
        c.adapter->add(servant, c.id);
    }

    return servant;
}
```

The Java version of this locator is almost identical, but we use the `synchronized` qualifier instead of a mutex to make `locate` a critical region:

Java

```
synchronized public Ice.Object
locate(Ice.Current c, Ice.LocalObjectHolder cookie)
{
    // Check if we have instantiated a servant already.
    //
    Ice.Object servant = c.adapter.find(c.id);

    if (servant == null) { // We don't have a servant already

        // Instantiate a servant
        //
        ServantDetails d;
        try {
            d = DB.lookup(c.id.name);
        } catch (DB.error&) {
            return null;
        }
        servant = new PhoneEntryI(d);

        // Add the servant to the ASM.
        //
        c.adapter.add(servant, c.id);
    }

    return servant;
}
```

In C#, you can place the body of `locate` into a `lock(this)` statement.

Using a servant locator that adds the servant to the ASM has a number of advantages:

- Servants are instantiated on demand, so the cost of initializing the servants is spread out over many invocations instead of being incurred all at once during server start-up.
- The memory requirements for the server are reduced because servants are instantiated only for those Ice objects that are actually accessed by clients. If clients only access a subset of the total number of Ice objects, the memory savings can be substantial.

In general, incremental initialization is beneficial if instantiating servants during start-up is too slow. The memory savings can be worthwhile as well but, as a rule, are realized only for comparatively short-lived servers: for long-running servers, chances are that, sooner or later, every Ice object will be accessed by some client or another; in that case, there are no memory savings because we end up with an instantiated servant for every Ice object regardless.

Implementing a Server using Default Servants

Default servants are a very effective tool for conserving memory when a server hosts a large number of Ice objects.

To create a default servant implementation, we need as many default servants as there are non-abstract interfaces in the system. For example, for our file system application, we require two default servants, one for directories and one for files. In addition, the object identities we create use the `category` member of the object identity to encode the type of interface of the corresponding Ice object. The value of the category field can be anything that identifies the interface, such as the 'd' and 'f' convention we suggested earlier. Alternatively, you could use `"Directory"` and `"File"`, or use the type ID of the corresponding interface, such as `"::Filesystem::Directory"` and `"::Filesystem::File"`. The `name` member of the object identity must be set to whatever identifier we can use to retrieve the persistent state of each directory and file from secondary storage. (For our file system application, we used a UUID as a unique identifier.)

Registration of the default servants is as follows:

C++

```
adapter->addDefaultServant(new DirectoryI, "d");
adapter->addDefaultServant(new FileI, "f");
```

All the action happens in the implementation of the operations, using the following steps for each operation:

1. Use the passed `Current` object to get the identity for the current request.
2. Use the `name` member of the identity to locate the persistent state of the servant on secondary storage. If no record can be found for the identity, throw an `ObjectNotExistException`.
3. Implement the operation to operate on that retrieved state (returning the state or updating the state as appropriate for the operation).

This might look something like the following:

C++

```
Filesystem::NodeSeq
Filesystem::DirectoryI::list(const Ice::Current& c) const
{
    // Use the identity of the directory to retrieve
    // its contents.
    DirectoryContents dc;
    try {
        dc = DB_getDirectory(c.id.name);
    } catch(const DB_error&) {
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);
    }

    // Use the records retrieved from the database to
    // initialize return value.
    //
    FileSystem::NodeSeq ns;
    // ...

    return ns;
}
```

Note that the servant implementation is completely stateless: the only state it operates on is the identity of the Ice object for the current request (and that identity is passed as part of the `Current` parameter).

Overriding `ice_ping`

We recommended that a default servant implementation take steps to preserve the semantics of the `ice_ping` operation, which is used to test whether an Ice object exists. If a default servant fails to override `ice_ping`, clients may mistakenly believe that a non-existent Ice object still exists. The code below demonstrates how we can override the operation in our file system application:

C++

```
void
Filesystem::DirectoryI::ice_ping(const Ice::Current& c) const
{
    try {
       d = DB_lookup(c.id.name);
    } catch (const DB_error&) {
       throw Ice::ObjectNotExistException(__FILE__, __LINE__);
    }
}
```

It is good practice to override `ice_ping` if you are using default servants.

Combining Server Implementation Techniques

Depending on the nature of your application, you may be able to steer a middle path that provides better performance while keeping memory requirements low: if your application has a number of frequently-accessed objects that are performance-critical, you can add servants for those objects to the ASM. If you store the state of these objects in data members inside the servants, you effectively have a cache of these objects.

The remaining, less-frequently accessed objects can be implemented with a default servant. For example, in our file system implementation, we could choose to instantiate directory servants permanently, but to have file objects implemented with a default servant. This provides efficient navigation through the directory tree and incurs slower performance only for the (presumably less frequent) file accesses.

This technique could be augmented with a cache of recently-accessed files, along similar lines to the buffer pool used by the Unix kernel [1]. The point is that you can combine use of the ASM with servant locators and default servants to precisely control the trade-offs among scalability, memory consumption, and performance to suit the needs of your application.

See Also

References

1. McKusick, M. K., et al. 1996. The Design and Implementation of the 4.4BSD Operating System. Reading, MA: Addison-Wesley.