Proxies

Building on the `Clock` example, we can create definitions for a world-time server:

Slice

```
exception GenericError {
    string reason;
};

struct TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
};

exception BadTimeVal extends GenericError {};

interface Clock {
    idempotent TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time) throws BadTimeVal;
};

dictionary<string, Clock*> TimeMap; // Time zone name to clock map

exception BadZoneName extends GenericError {};

interface WorldTime {
    idempotent void addZone(string zoneName, Clock* zoneClock);
    void removeZone(string zoneName) throws BadZoneName;
    idempotent Clock* findZone(string zoneName) throws BadZoneName;
    idempotent TimeMap listZones();
    idempotent void setZones(TimeMap zones);
};
```

The `WorldTime` interface acts as a collection manager for clocks, one for each time zone. In other words, the `WorldTime` interface manages a collection of pairs. The first member of each pair is a time zone name; the second member of the pair is the clock that provides the time for that zone. The interface contains operations that permit you to add or remove a clock from the map (`addZone` and `removeZone`), to search for a particular time zone by name (`findZone`), and to read or write the entire map (`listZones` and `setZones`).

The `WorldTime` example illustrates an important Slice concept: note that `addZone` accepts a parameter of type `Clock*` and `findZone` returns a parameter of type `Clock*`. In other words, interfaces are types in their own right and can be passed as parameters. The `*` operator is known as the *proxy operator*. Its left-hand argument must be an interface (or class) and its return type is a proxy. A proxy is like a pointer that can denote an object. The semantics of proxies are very much like those of C++ class instance pointers:

- A proxy can be null.
- A proxy can dangle (point at an object that is no longer there).
- Operations dispatched via a proxy use late binding: if the actual run-time type of the object denoted by the proxy is more derived than the proxy's type, the implementation of the most-derived interface will be invoked.

When a client passes a `Clock` proxy to the `addZone` operation, the proxy denotes an actual `Clock` object in a server. The `Clock` *Ice object* denoted by that proxy may be implemented in the same server process as the `WorldTime` interface, or in a different server process. Where the `Clock` object is physically implemented matters neither to the client nor to the server implementing the `WorldTime` interface; if either invokes an operation on a particular clock, such as `getTime`, an RPC call is sent to whatever server implements that particular clock. In other words, a proxy acts as a local "ambassador" for the remote object; invoking an operation on the proxy forwards the invocation to the actual object implementation. If the object implementation is in a different address space, this results in a remote procedure call; if the object implementation is collocated in the same address space, the Ice run time uses an ordinary local function call from the proxy to the object implementation.

Note that proxies also act very much like pointers in their sharing semantics: if two clients have a proxy to the same object, a state change made by one client (such as setting the time) will be visible to the other client.

Proxies are strongly typed (at least for statically typed languages, such as C++ and Java). This means that you cannot pass something other than a `Clock` proxy to the `addZone` operation; attempts to do so are rejected at compile time.

See Also

- Classes