Removing Cyclic Dependencies

We mentioned earlier that factoring the `_names` set and its mutex into a separate class instance does not really solve the cyclic dependency problem, at least not in general. To see why, suppose that we want to extend our factory with a new `getDetails` operation:

Slice

```
// ...

struct Details {
    PhoneEntry* proxy;
    string name;
    string phNum;
};

sequence<Details> DetailsSeq;

interface PhoneEntryFactory {
    // ...

    DetailsSeq getDetails();
};
```

This type of operation is common in collection managers: instead of returning a simple list of proxies, `getDetails` returns a sequence of structures, each of which contains not only the object's proxy, but also some of the state of the corresponding object. The motivation for this is performance: with a plain list of proxies, the client, once it has obtained the list, is likely to immediately follow up with one or more remote calls for each object in the list in order to retrieve their state (for example, to display the list of objects to the user). Making all these additional remote procedure calls is inefficient, and an operation such as `getDetails` gets the job done with a single RPC instead.

To implement `getDetails` in the factory, we need to iterate over the set of entries and invoke the `getNumber` operation on each object. (These calls are collocated and therefore very efficient, so they do not suffer the performance problem that a client calling the same operations would suffer.) However, this is potentially dangerous because the following sequence of events is possible:

- Client A calls `getDetails`.
- The implementation of `getDetails` must lock `_namesMutex` to prevent concurrent modification of the `_names` set during iteration.
- Client B calls `destroy` on a phone entry.
- The implementation of `destroy` locks the entry's mutex `_m`, sets the `_destroyed` flag, and then calls `remove`, which attempts to lock `_namesMutex` in the factory. However, `_namesMutex` is already locked by `getDetails`, so `remove` blocks until `_m` is unlocked again.
- `getDetails`, while iterating over its set of entries, happens to call `getNumber` on the entry that is currently being destroyed by client B. `getNumber`, in turn, tries to lock its mutex `_m`, which is already locked by `destroy`.

At this point, the server deadlocks: `getDetails` holds a lock on `_namesMutex` and waits for `_m` to become available, and `destroy` holds a lock on `_m` and waits for `_namesMutex` to become available, so neither thread can make progress.

To get rid of the deadlock, we have two options:

- Rearrange the locking such that deadlock becomes impossible.
- Abandon the idea of calling back from the servants into the factory and use *reaping* instead.

We will explore both options in the following pages.

Topics

- Acquiring Locks without Deadlocks
- Reaping Objects

See Also

- Life Cycle and Collection Operations