Avoiding Server-Side Garbage

On this page:

Creating Garbage

We have discussed the implementation of object life cycle, that is, how to correctly provide clients with the means to create and destroy objects. However, throughout this discussion, we have tacitly assumed that clients actually call `destroy` once they no longer need an object. What if this is actually not the case? For example, a client might intend to call `destroy` on an object it no longer needs but crash before it can actually make the call.

To see why this is a realistic (and serious) scenario, consider an on-line retail application. Typically, such an application provides a shopping cart to the client, into which the client can place items. Naturally, the cart and the items it contains will be modelled as server-side objects. The expectation is that, eventually, the client will either finalize or cancel the purchase, at which point the shopping cart and its contents can be destroyed. However, the client may never do that, for example, because it crashes or simply loses interest.

The preceding scenario applies to many different applications and shows up in various disguises. For example, the objects might be session objects that encapsulate security credentials of a client, or might be iterator objects that allow a client to iterate over a collection of values. The key point is that the interactions between client and server are stateful: the server creates state on behalf of a client, holds that state for the duration of several client-server interactions, and expects the client to inform the server when it can clean up that state. If the client never informs the server, whatever resources are associated with that client's state are leaked; these resources are termed *garbage*.

The garbage might be memory, file descriptors, network connections, disk space, or any number of other things. Unless the server takes explicit action, eventually, the garbage will accumulate to the point where the server fails because it has run out of memory, file descriptors, network connections, or disk space.

In the context of Ice, the garbage are servants and their associated resources. In this section, we examine strategies that a server can use avoid drowning in that garbage.

Garbage Collection Solutions

The server is presented with something of a dilemma by garbage objects. The difficulty is not in how to remove the garbage objects (after all, the server knows how to destroy each object), but how to identify whether a particular object is garbage or not. The server knows when a client uses an object (because the server receives an invocation for the object), but the server does not know when an object is no longer of interest to a client (because a dead client is indistinguishable from a slow one).

One approach to dealing with garbage is to avoid creating it in the first place: if all interactions between client and server are stateless, the garbage problem does not arise. Unfortunately, for many applications, implementing this approach is infeasible. The reason is that, in order to turn interactions that are inherently stateful (such as updating a database) into stateless ones, designers are typically forced to keep all the state on the client side, transmit whatever state is required by the server with each remote procedure call, and return the updated state back to the client. In many situations, this simply does not work: for one, the amount of state that needs to be transmitted with each call is often prohibitively large; second, replicating all the state on the client side creates other problems, such as different clients concurrently making conflicting updates to the same data.

The remainder of this section ignores stateless designs. This is not to say that stateless designs are undesirable: where suitable, they can be very effective. However, because many applications simply cannot use them, we focus instead on other ways to deal with garbage.

Mechanisms that identify and reclaim garbage objects are known as garbage collectors. Garbage collectors are well-understood for non-distributed systems. (For example, many programming languages, such as Java and C#, have built-in garbage collectors.)

Non-distributed garbage collectors keep track of all objects that are created, and perform some form of connectivity analysis to determine which objects are still reachable; any objects that are unreachable (that is, objects to which the application code no longer holds any reference) are garbage and are eventually reclaimed.

Unfortunately, for distributed systems, traditional approaches to garbage collection do not work because the cost of performing the connectivity analysis becomes prohibitively large. For example, DCOM provided a distributed garbage collector that turned out to be its Achilles' heel: the collector did not scale to large numbers of objects, particularly across WANs, and several attempts at making it scale failed.

An alternative to distributed garbage collection is to use timeouts to avoid the cost of doing a full connectivity analysis: if an object has not been used for a certain amount of time, the server assumes that it is garbage and reclaims the object. The drawback of this idea is that it is possible for objects to be collected while they are still in use. For example, a customer may have placed a number of items in a shopping cart and gone out to lunch, only to find on return that the shopping cart has disappeared in the mean time.

Yet another alternative is to use the evictor pattern: the server puts a cap on the total number of objects it is willing to create on behalf of clients and, once the cap is reached, destroys the least-recently used object in order to make room for a new one. This puts an upper limit on the resources used by the server and eventually gets rid of all unwanted objects. But the drawback is the same as with timeouts: just because an object has not been used for a while does not necessarily mean that it truly is garbage.

Neither timeouts nor evictors are true garbage collectors because they can collect objects that are not really garbage, but they do have the advantage that they reap objects even if the client is alive, but forgets to call `destroy` on some of these objects.

Simple Mechanisms for Garbage Collection

Traditional garbage collection fails in the distributed case for a number of reasons:

- Garbage collectors require connectivity analysis, which is prohibitively expensive. Furthermore, for distributed object systems that permit proxies to be externalized as strings, such as Ice, connectivity analysis is impossible because proxies can exist and travel by means that are invisible to the run time. For example, proxies can exist as records in a database and can travel as strings inside e-mail messages.
- Garbage collectors consider all objects in existence but, for the vast majority of applications, only a small subset of all objects actually ever needs collecting. The work spent in examining objects that can never become garbage is wasted.
- Garbage collectors examine connectivity at the granularity of a single object. However, for many distributed applications, objects are used in groups and, if one object in a group is garbage, all objects in the group are garbage. It would be useful to take advantage of this knowledge, but a garbage collector cannot do this because that knowledge is specific to each application.

In the remainder of this section, we examine a simple mechanism that allows you to get rid of garbage objects cheaply and effectively. The approach has the following characteristics:

- Only those objects that potentially can become garbage are considered for collection.
- Granularity of collection is under control of the application: you can have objects collected as groups of arbitrary size, down to a single object.
- Objects are guaranteed not to be collected prematurely.
- Objects are guaranteed to be collected if the client crashes or suffers loss of connectivity.
- The mechanism is simple to implement and has low run-time overhead.

It is equally important to be aware of the limitations of the approach:

- The approach collects objects if a client crashes, but offers no protection against clients that are still running, but have neglected to destroy objects that they no longer need. In other words, the server is protected against client-side hardware failure and catastrophic client crashes, but it is not protected against faulty programming logic of clients.
- The approach is not transparent at the interface level: it requires changes (albeit minor ones) to the interface definitions for an application.
- The approach requires the client to periodically call the server, thus consuming network resources even if the client is otherwise idle.

Despite the limitations, this approach to garbage collection is applicable to a wide variety of applications and meets the most pragmatic need: how to clean up in case something goes badly wrong (rather than how to clean up in case the client misbehaves).

Using an Extra Level of Indirection to Collect Garbage

Object factories are typically singleton objects that create objects on behalf of various clients. It is important for our garbage collector to know which client created what objects, so the collector can reap the objects created by a specific client if that client crashes. We can easily deal with this requirement by adding the proverbial level of indirection: instead of making a factory a singleton object, we provide a singleton object that creates factories. Clients first create a factory and then create all other objects they need using that factory:

Slice

```
interface Item { /* ...*/ };

interface Cart
{
    Item* create(/* ... */);
    idempotent string getName();
    void destroy();
    idempotent void refresh();
};

interface CartFactory // Singleton
{
    Cart* create(string name);
};
```

Clients obtain a proxy to the `CartFactory` singleton and call `create` to create a `Cart` object. In turn, the `Cart` object provides a create operation to place new objects of type `Item` into the cart. Note that the shopping cart name allows a client to distinguish different carts — the `name` parameter is *not* used to identify clients or to provide a unique identifier for carts. The `getName` operation on the `Cart` object returns the name that was used by the client to create it.

Each `Cart` object remembers which items it created. Because each client uses its own shopping cart, the server knows which objects were created by what client. In normal operation, a client first creates a shopping cart, and then uses the cart to create the items in the cart. Once the client has finished its job, it calls `destroy` on the cart. The implementation of `destroy` destroys both the cart and its items to reclaim resources.

To deal with crashed clients, the server needs to know when a cart is no longer in use. This is the purpose of the `refresh` operation: clients are expected to periodically call `refresh` on their cart objects. For example, the server might decide that, if a client's cart has not been refreshed for more than ten minutes, the cart is no longer in use and reclaim it. As long as the client calls `refresh` at least once every ten minutes, the cart (and the items it contains) remain alive; if more than ten minutes elapse, the server simply calls `destroy` on the cart. Of course, there is no need to hard-wire the timeout value — you can make it part of the application's configuration. However, to keep the implementation simple, it is useful to have the same timeout value for all carts, or to at least restrict the timeouts for different carts to a small number of fixed choices — this considerably simplifies the implementation in both client and server.

Server-Side Changes for Garbage Collection

The implementation of the interfaces on the server side almost suggests itself:

- Whenever `refresh` is called on a cart, the cart records the time at which the call was made.
- The server runs a reaper thread that wakes up once every ten minutes. The reaper thread examines the timestamp of all carts and, if it finds a cart last time-stamped more than ten minutes ago, it calls `destroy` on that cart.
- Each cart remembers the items it contains and destroys them as part of its `destroy` implementation.

Here then is the reaper thread in outline. (Note that we have simplified the code to show the essentials. For example, we have omitted the code that is needed to make the reaper thread terminate cleanly when the server shuts down. See the code in `demo/Ice/session` for more detail.)

C++

```
class ReapThread : public IceUtil::Thread,
                   public IceUtil::Monitor<IceUtil::Mutex>
{
public:
    ReapThread();
    virtual void run();
    void add(const CartPrx&, const CartIPtr&);

private:
    const IceUtil::Time _timeout;
    struct CartProxyPair
    {
        CartProxyPair(const CartPrx& p, const CartIPtr& c) :
              proxy(p), cart(c) { }
        const CartPrx proxy;
        const CartIPtr cart;
    };
    std::list<CartProxyPair> _carts;
};

typedef IceUtil::Handle<ReapThread> ReapThreadPtr;
```

Note that the reaper thread maintains a list of pairs. Each pair stores the proxy of a cart and its servant pointer. We need both the proxy and the pointer because we need to invoke methods on both the Slice interface and the implementation interface of the cart. Whenever a client creates a new cart, the server calls the `add` method on the reaper thread, passing it the new cart:

C++

```
void ReapThread::add(const CartPrx& proxy, const CartIPtr& cart)
{
    Lock sync(*this);
    _carts.push_back(CartProxyPair(proxy, cart));
}
```

The `run` method of the reaper thread is a loop that sleeps for ten minutes and calls `destroy` on any session that has not been refreshed within the preceding ten minutes:

C++

```
void ReapThread::run()
{
    Lock sync(*this);
    while (true) {
        timedWait(_timeout);
        list<CartProxyPair>::iterator p = _carts.begin();
        while (p != _carts.end()) {
            try {
                //
                // Cart destruction may take some time.
                // Therefore the current time is computed
                // for each iteration.
                //
                if ((IceUtil::Time::now() - p->cart->timestamp()) > _timeout) {
                    p->proxy->destroy();
                    p = _carts.erase(p);
                } else {
                    ++p;
                }
            } catch (const Ice::ObjectNotExistException&) {
                p = _carts.erase(p);
            }
        }
    }
}
```

Note that the reaper thread catches `ObjectNotExistException` from the call to `destroy`, and removes the cart from its list in that case. This is necessary because it is possible for a client to call `destroy` explicitly, so a cart may be destroyed already by the time the reaper thread examines it.

The `CartFactory` implementation is trivial:

C++

```
class CartFactoryI : CartFactory
{
public:
    CartFactoryI(const ReapThreadPtr&);
    virtual CartPrx create(const std::string&, const Ice::Current&);

private:
    ReapThreadPtr _reaper;
};
```

The constructor is passed the instantiated reaper thread and remembers that thread in the `_reaper` member.

The `create` method adds each new cart to the reaper thread's list of carts:

C++

```
CartPrx CartFactoryI::create(const string& name, const Ice::Current& c)
{
    CartIPtr cart = new CartI(name);
    CartPrx proxy = CartPrx::uncheckedCast(
                        c.adapter->addWithUUID(cart));
                        _reaper->add(proxy, cart);
    return proxy;
}
```

Note that each cart internally has a unique ID that is unrelated to its name — the name exists purely as a convenience for the application.

The server's `main` function starts the reaper thread and instantiates the cart factory:

C++

```
ReapThreadPtr reaper = new ReapThread();
CartFactory factory = new CartFactoryI(reaper);
reaper->start();
adapter->add(factory, Ice::stringToIdentity(CartFactory));
adapter->activate();
```

This completes the implementation on the server side. Note that there is very little code here, and that much of this code is essentially the same for each application. For example, we could easily turn the `ReapThread` class into a template class to permit the same code to be used for something other than shopping carts.

Client-Side Changes for Garbage Collection

On the client side, the application code does what it would do with an ordinary factory, except for the extra level of indirection: the client first creates a cart, and then uses the cart as its factory.

As long as the client-side calls `refresh` at least once every ten minutes, the cart remains alive and, with it, all items the client created in that cart. Once the client misses a `refresh` call, the reaper thread in the server cleans up the cart and its items.

To keep the cart alive, you could sprinkle your application code with calls to `refresh` in the hope that at least one of these calls is made at least every ten minutes. However, that is not only error-prone, but also fails if the client blocks for some time. A much better approach is to run a thread in the client that automatically calls `refresh`. That way, the calls are guaranteed to happen even if the client's main thread blocks for some time, and the application code does not get polluted with `refresh` calls. Again, we show a simplified version of the refresh thread here that does not deal with issues such as clean shutdown and a few other irrelevant details:

C++

```
class CartRefreshThread : public IceUtil::Thread,
                          public IceUtil::Monitor<IceUtil::Mutex>
{
public:
    CartRefreshThread(const IceUtil::Time& timeout, const CartPrx& cart) :
        _cart(cart),
        _timeout(timeout) {}

    virtual void run() {
        Lock sync(*this);
        while(true) {
            timedWait(_timeout);
            try {
                _cart->refresh();
            } catch(const Ice::Exception& ex) {
                return;
            }
        }
    }

private:
    const CartPrx _cart;
    const IceUtil::Time _timeout;
};

typedef IceUtil::Handle<CartRefreshThread> CartRefreshThreadPtr;
```

The client's `main` function instantiates the reaper thread after creating a cart. We assume that the client has a proxy to the cart factory in the `factory` variable:

C++

```
CartPrx cart = factory->create(name);
CartRefreshThreadPtr refresh = new CartRefreshThread(IceUtil::Time::seconds(480), cart);
refresh->start();
```

Note that, to be on the safe side and also allow for some network delays, the client calls refresh every eight minutes; this is to ensure that at least one call to `refresh` arrives at the server within each ten-minute interval.

See Also

- Object Life Cycle