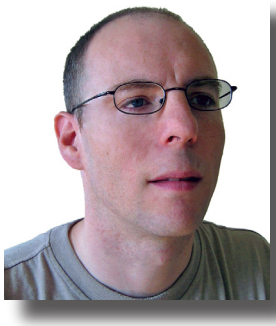
Connections

ZeroC's Newsletter for the Ice Community

Issue Number 25, July 2007



Bridge to Nowhere?

Many moons ago, I was asked by my employer to integrate credit card processing into an application that ran on a cluster of UNIX machines. Unfortunately, at that time, the only way to talk with our credit card processing center was via a proprietary protocol, and the only way to interface to that protocol was via a COM library provided by the center. Since I was in UNIX land, this naturally didn't work. So what to do? Our application was built on CORBA and, as it turned out, our CORBA vendor offered a COM-CORBA bridge.

A bridge is a piece of software that transparently translates calls originating in one system to calls on the target system by performing protocol conversion or by intercepting calls and re-issuing them via the API of the target system. This seemed like heaven—a perfect solution that would allow me to live in my UNIX/CORBA world and not deal with all the Windows COM complexity.

As it turned it out, the solution was anything but perfect. After using the bridge for a while, it became clear that the bridge did not work as expected. For one, the bridge itself had bugs that were hard to track down. Second, the COM-CORBA mapping was awkward and resulted in unnatural and difficult-to-use APIs. But, worst of all, I ended up coding in a half-CORBA, half-COM world—a distinctly weird and counter-intuitive place indeed.

After thinking about the problem some more, I began to realize why my bridge (and bridges in general) caused such problems. A bridge, by its very nature, restricts the type system to the lowest common denominator of both systems and, worse, ignores differences in the object model. For example, COM controls object life time with reference counting, whereas CORBA has no such mechanism. As a result, a bridge rarely (if ever) makes a good fit with existing systems and makes interoperation quite difficult, if not impossible. (Personally, I believe that automatic interoperation of distributed systems is a pipe dream unless the distributed systems in question are indistinguishable.) Finally, even if it works, a bridge tends to be inefficient because it burns a lot of CPU cycles and funnels communications through a single choke point, with negative performance impact.

So, why would anyone want to use a bridge, despite these problems? Usually, the motivation is that a particular system has complex APIs, poor documentation or support, or does not support a particular platform or programming language. A bridge is often seen as a low-cost way to cross this chasm, even though automatic interoperability among distributed systems is theoretical and, pragmatically, does not exist. At best, the bridge reaches only halfway.

In my case, I ended up solving the problem by writing a CORBA server that provided natural and easy-to-use CORBA interfaces to clients, and that implemented the operations on these interfaces by calling the into the COM library. That way, I avoided the COM-CORBA bridge's ugly APIs as well as its lowest-common-denominator type system, and I could also take care of matching the semantics of the two object models appropriately. Of course, my server was not a bridge; instead, it was a credit card processing service that naturally formed part of the application—the server just so happened to use a COM library as part of its implementation, hidden from view behind CORBA interfaces.

The moral of this story is that, before you reach for a generic bridge that most likely will disappoint you, consider solving the problem the way I described. The wide-ranging platform and programming language support of Ice make it almost certain that Ice will run in the target environment, thereby eliminating the need for a bridge altogether. In turn, you will enjoy natural APIs and better performance. And that beats being left stranded on a bridge that almost, but not quite, reaches the other side.

Matthew Newhook, Senior Software Engineer

Issue Features

Beyond Freeze—Persistence with Ice Part 2: Advanced Topics

Part 2 of Stephan Stapel's article discusses concurrency, transactions, connection management, and scalability issues.

Who's Counting?

Michi Henning shows you how to use Ice smart pointers to ease application development and reduce defect rates.

Contents

Beyond Freeze—Persistence with Ice Part 2: Advanced Topics 2

Who's Counting? 10

FAQ Corner 24

Beyond Freeze—Persistence with Ice Part 2: Advanced Topics

Stephan Stapel, Freelance Systems Architect
S2 Industries

In part 1 of this article series, I discussed the basics of how to integrate Ice applications with a relational database management system (RDBMS) and how to implement the necessary object-relational (O/R) mapping. In this article, I examine more advanced topics, in particular, concurrency and transactions, using O/R mappers, connection management, and how to scale an application to very large numbers of objects.

Implementing Concurrency

Besides entity-relationship modeling and its implementation in relational databases such as Oracle, MySQL, and PostgreSQL, databases also provide the concept of transactions, which are responsible for maintaining consistency of the data that is stored in a database. Transactions ensure that concurrent modifications to data are either performed completely or revoked to the state prior to a transaction. The famous ACID paradigm formalizes the idea of transactions:

- *A* stands for *atomicity* which makes sure that—as mentioned before—all of the tasks of a transaction are either performed successfully or revoked completely.
- *C* stands for *consistency*, which ensures the consistent state of a database after performing a transaction, that is, enforces the database's integrity constraints.
- *I* stands for *isolation*, which guarantees that transactions only see the committed state of other transactions but not intermediate (and possibly inconsistent) state internal to some transaction.
- *D* stands for *durability*, which guarantees that state changes are persistent and will not be lost, for example, in the case of system failure.

In order to guarantee isolation, databases use locking. A lock grants exclusive access to a section of data to a transaction and prevents, for example, one transaction reading data while another transaction is modifying the same data. Depending on the database, locking takes place at various granularities; a particular implementation may lock an entire table at a time, whereas another implementation may lock only particular rows of a table. The most fine-grained locking mechanism locks specific attributes.

So-called *pessimistic locking* is implemented by acquiring locks early during a transaction's life cycle, which makes lock conflicts more likely but, if a conflict does occur, the conflict is discovered early. Pessimistic locking is not always the best approach to achieve isolation because, depending on the implementation, locks not only affect clients that modify data, but also clients that read data.

Another approach, known as *optimistic locking*, delays locking data until a transaction actually performs an operation on the data. This makes lock conflicts less likely but delays their discovery. Code development for databases that use optimistic locking is generally easier.

Optimistic Locking

Optimistic locking generally works well if conflicting data access is expected to be infrequent. The basic approach is as follows: when a pre-existing entity, in our case a descriptor, is modified and needs to be written to the database, the only check we perform is whether someone else modified the same descriptor in the meantime. This check is performed by the application, that is, there is no additional database logic involved in handling this task. We can achieve the check by adding an attribute to both the entity and the descriptor, known as the *optimistic concurrency control attribute*. When retrieving a row from the database and storing it into the corresponding descriptor, we retrieve this attribute with the data and store it in the descriptor.

There are various ways to implement such an attribute. For example, we can use a simple integer value that is increased by each update and acts as a serial number or, alternatively, we can use a `datetime` field that reflects the date and time of the last update.

During a save attempt, we compare the database value of the attribute with the (possibly dirty) attribute value of the descriptor. If the database value of the attribute differs from the value in the descriptor, it is clear that the row was modified in the meantime and thus we reject the update with an exception. (Object-relational mappers, such as Hibernate, have this functionality built-in and so can be used with very little additional coding.)

Note that, with optimistic locking, we still need transactions at the database level: while we do not need transactions to implement optimistic locking, transactions are necessary to maintain clean database state during `SELECT`/`UPDATE` operations.

To use optimistic locking with the example application from part 1 of this article, we add a new field `ocadate` (OCA stands for *O*ptimistic *C*oncurrency *A*ttribute):

```
-- PostgreSQL
CREATE TABLE Project (
    id integer,
    name text,
    description text,
    creatorid integer,
    creationdate timestamp without time zone,
    ocadate timestamp without time zone
);
```

As mentioned, the OCA attribute must also be part of the descriptor. Thus, we add a new attribute to `CProjectDesc`:

```
// Slice
exception SaveFailedException {};

struct CProjectDesc
{
    CProject* projectproxy;
    string szName;
    // ...
    // datetime value to store the last save date
    DateTime ocaDate;
};

interface CProject
{
    idempotent CProjectDesc describe();
    void saveUpdate(CProjectDesc descriptor)
        throws SaveFailedException;
};
```

`CProject` now also contains a `saveUpdate` operation that attempts to atomically update the descriptor. If an update fails because `ocaDate` indicates that the descriptor is dirty, the operation throws `SaveFailedException`.

At the database layer, we update an existing row using:

```
-- PostgreSQL
update name, ..., ocadate (newname, ..., current_
timestamp) from Project where id = x and ocadate =
oldocadate
```

Alternatively, if using an integer `ocaversion`:

```
-- PostgreSQL
update name, ..., ocaversion (newname, ...,
oldocaversion + 1) from Project where id = x and
ocaversion = ocaversion
```

Depending on whether the application updates a single descriptor or several, it can either use implicit transactions (omitting calls to `begin` and `commit`) or create explicit transaction boundaries (bracketing several updates with `begin` and `commit`).

Optimistic locking is simple and effective. For our example application, suppose we have a server-side project with the name `'initial project name'` and users perform the following sequence of operations:

1. User 1 calls `describe` and retrieves the descriptor (`descriptor.szName = 'initial project name'` and `descriptor.ocaversion = 1`).
2. User 1 (locally, on the client side) modifies the descriptor and changes `descriptor.szName` to `'my new project name'`.
3. User 2 also calls `describe` and receives the still valid descriptor (`descriptor.szName = 'initial project name'`).
4. User 2 modifies its local copy of the descriptor and sets `descriptor.szName` to `'my very new project name'`.
5. User 1 commits the change using `saveUpdate` with `descriptor.szName = 'my new project name'` and `descriptor.ocaversion = 1`.
6. The commit is successful because the `ocaversion` in the descriptor and on the server-side are still identical. This increases `ocaversion` on the server by one and the new server-side values are `name = 'initial project name'` and `ocaversion = 2`.
7. User 2 now wants to commit the modification and calls `saveUpdate` using a descriptor that contains `descriptor.szName = 'my very new project name'` and `descriptor.ocaDate = 1`.
8. The commit fails because the descriptor's `ocaversion` differs from the server-side `ocaversion`.
9. The server raises a `SaveFailedException` that can be propagated to the user via a meaningful error message.

One drawback of optimistic locking is that it covers only single-object modifications because, if we have several interconnected objects that are modified as a set, it can happen that only some of these objects can be saved successfully because other objects in the same set have changed in the meantime.

There are various approaches to deal with this problem: we can introduce additional transaction handling (as described in the next section) or we can modify the data model such that objects that may be modified as a set are stored within a single database table. Another option is to have objects within the same set share a single OCA attribute. We can achieve this by interconnecting the OCA attributes of the corresponding set of tables: each time one of the OCA attributes of an object is increased, we also increase the OCA attributes of the remaining objects in the set. (We can use triggers or additional server-side logic to implement this.)

Pessimistic Locking

For applications with a high rate of data modification and, therefore, high probability of conflicts, we need to resort to pessimistic locking: data that will be modified by a user is explicitly locked before starting the update process and is freed when the user commits or rolls back the changes; during that time, other users can only gain read access to the corresponding data set.

The interface for this scheme looks like this:

```
// Slice
exception LockFailedException;
exception CommitFailedException;
exception RollbackFailedException;

interface CProject
{
    idempotent CProjectDesc describe();
    CProjectDesc describeForUpdate()
        throws LockFailedException;
    void saveUpdate(CProjectDesc desc)
        throws CommitFailedException;
    void rollback()
        throws RollbackFailedException;
};
```

As with optimistic locking, we have a `describe` operation. In addition, a `describeForUpdate` operation has the same functionality, but also explicitly locks the corresponding data set.

As for optimistic locking, clients call `saveUpdate` to commit an update. In addition, we need to provide a `rollback` operation that closes the transaction without committing updated data.

Here is the preceding concurrent update scenario once more, this time using pessimistic locking:

1. User 1 calls `describeForUpdate` and receives the initial descriptor for further modification.
2. User 2 calls `describeForUpdate` and receives a `LockFailedException`.
3. User 1 modifies the object.
4. User 1 calls `saveUpdate`, which releases the lock on the data set.
5. User 2 calls `describeForUpdate` again and receives the new descriptor, which also locks the data set for further modification by user 2.

At first glance, this use case is simpler than the use case with optimistic locking. Unfortunately, it hides a number of constraints that must be taken into account when implementing the application. For one, we cannot share database connections among multiple requests because we need to guarantee that transaction boundaries are maintained; if the same database connection is used by more than one transaction, it is no longer clear to which transaction a particular update belongs. Therefore, we need a connection manager that maps transactions to database connections. (I will introduce such a mechanism later in this article because it is not only a good tool to manage transactions but also overcomes general multithreading issues for databases.)

Second, in a distributed environment, we cannot guarantee that a particular client will not suddenly disappear in the middle of an operation. If this happens during a transaction, the transaction must be rolled back to reclaim its locks. Doing this requires a separate routine that monitors all open database connections for request activity and eventually rolls back the transaction if a connection remains idle for too long.

The implementation can no longer use auto-commit (which was possible with optimistic locking) because we need transactions that span multiple requests. Therefore, a transaction's underlying connection cannot automatically be closed once a request completes but must be held open until the client performs an explicit commit or rollback.

In SQL, we can lock a database row using the `SELECT FOR UPDATE` command. Without additional parameters, this call blocks until the caller either commits or rolls back the requested row (that is, `describeForUpdate` would not return until the row gets unlocked). In a distributed environment, this is not the desired behavior. Databases such as Oracle and PostgreSQL support an additional `NOWAIT` parameter that guarantees non-blocking behavior: if a lock cannot be acquired, `SELECT FOR UPDATE` returns an error code that the application can propagate as a `LockFailedException`.

Referring to the previous example, the following happen beneath the covers:

1. User 1 calls `describeForUpdate` and receives the initial descriptor for further modification. Internally, this locks the corresponding database row for user 1's connection by executing `SELECT * FROM Project WHERE id = x FOR UPDATE NOWAIT`. The connection stays open and assigned to user 1 once the request completes.
2. User 2 calls `describeForUpdate` and receives a `LockFailedException`. Internally, this makes an attempt to acquire a lock by executing `SELECT * FROM Project WHERE id = x FOR UPDATE NOWAIT`. However, the `SELECT` returns an error code that is passed to the user as the `LockFailedException`.
3. User 1 modifies the object.
4. User 1 calls `saveUpdate`, which releases the lock on the data set. Internally, this calls `COMMIT` on the transaction associated with user 1's connection.
5. User 2 calls `describeForUpdate` again and receives the new descriptor, which also locks the data set for further modification by user 2. Internally, this locks the database row again using another `SELECT * FROM Project WHERE id = x FOR UPDATE NOWAIT`, but this time using a new connection object that is then assigned to user 2 (and not freed).

The preceding does not address modification of multiple objects within one transaction. Doing so would require creating a transaction management component, which makes applying the pessimistic locking scheme even more difficult.

In summary, I generally suggest to use optimistic locking because this requires less logic in the back end and thus is less error prone.

Using an O/R Mapper

Object-relational mappers—known as *O/R mappers*—are becoming the de-facto standard for accessing databases from modern programming languages such as Java and C#. Among others, Hibernate (for Java) and NHibernate (for C#) are two popular mappers that I will use as an example.

Common to all mapping services is that they hide SQL queries behind an object-oriented facade that provides convenient access to the underlying database. Furthermore, database rows are automatically mapped to objects, so-called *POJO*s and *POCO*s (*Plain Old Java Objects* and *Plain Old C# Objects*, respectively). O/R mappers even resolve foreign-key relationships and map them to object references. To map a data model to a POCO, NHibernate uses an XML definition:

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping
 xmlns="urn:nhibernate-mapping-2.0">
  <class name="MyApplication.Persistence.Project,
   MyApplication.Persistence" table="project">
    <id name="id" column="id" type="Int32"
     unsaved-value="0">
      <generator class="native">
        <param name="sequence">
           project_id_seq
        </param>
      </generator>
    </id>
    <version name="ocaVersion" column="ocaversion"
     type="Int32"/>
    <property name="szName" column="name"
     type="String"/>
    <property name="creatorId" column="creatorid"
     type="Int32"/>
    <property name="creationDate"
     column="creationdate" type="DateTime"/>
  </class>
</hibernate-mapping>
```

The `<class>` tag defines the mapping class and table, and the `<id>` tag defines how the primary key (the row ID) is generated. For this example, we'll use the database sequence generator. (NHibernate provides other strategies as well.)

Next, we define the aforementioned OCA field; for this example, we use a serial number that NHibernate increases automatically on each update. Finally, we map the database fields to their corresponding class member variables. (Please note that NHibernate supports foreign-key relationships, which I omitted for the `creatorId` field to keep things simple.)

Accessing the database through NHibernate in your program code looks like as follows. (I assume the code has previously initialized the `sessionFactory` object).

```
// C#
NHibernate.ISession session =
    sessionFactory.OpenSession();
NHibernate.ICriteria crit =
    session.CreateCriteria(
        typeof(Persistence.Project));
crit.Add(new NHibernate.Expression.
EqExpression("id", id));
if (crit.List().Count == 1)
{
    Persistence.Project project =
        (Persistence.Project)crit.UniqueResult();
}
else
{
    // throw error
}
session.close();
```

I suggest that you avoid mapping tables directly to descriptor classes, even though this initially looks attractive. The reason is that not all SQL and NHibernate types have direct Slice equivalents. (The time stamps I mentioned in part 1 of this article are an example). Thus, it is better to use a well-defined mapping layer that is exclusively responsible for mapping between descriptors and POCOs/ POJOs.

Connection Management

Ice has the advantage of being inherently multithreaded. Usually, the presence of multiple threads within a server is transparent to the application. However, database connections are often sensitive to multi-threading. In pure PostgreSQL for instance, a single database connection can be used by only one thread at a time. The server-side application code can deal with this limitation by using mutexes that guard the connection objects; however, doing so often causes a performance bottleneck.

Another strategy is to use multiple connections and store them in thread-local storage. That way, each application thread uses its own connection. (This method is used in the Oracle demos that are part of Ice 3.2.0—see the `demo/oracle` directory.) However, with large numbers of threads in an application, this can cause problems due to limits on the number of connections supported by the database.

A more adaptive approach is to create pool of connections: for each request that involves database activity, the application picks a connection from the pool and locks it for that particular request; once the request completes, the connection is returned to the pool for further use. This is more efficient than using a dedicated connection for each thread. (This pattern is, for example, used by the Java Connector API, which provides good inspiration for implementing custom connection pools in C++. Persistence mappers such as Hibernate also make use of such pools within their session factory object.)

The design goals for a connection pool are:

- Simplify connection handling by providing a single factory.
- Ensure thread-safety of connection handles.
- Reduce the number of open database sessions by reusing unused connections whenever possible.
- For pessimistic locking, isolate transactions by ensuring that each request receives a distinct connection handle when a transaction is started, thereby dedicating a separate connection to each transaction at a time.

No common implementation of database connection pools exists for C++, so I'll provide a rough sketch of how such a library could look using Qt:

```
// C++
class Connection
{
public:
    QSqlDatabase* pDatabase;
    void close();
    void beginTransaction();
    void commit();
    void rollback();
};

class ConnectionFactory
{
public:
    Connection* getConnection();
};
```

Using this library is similar to the preceding NHibernate example:

```
// C++
void
CProjectI::saveUpdate(const CProjectDesc& desc)
{
    Connection* pConn = pFactory->getConnection();

    QString szQuery   =
    QSqlQuery query("update ... from Project where
id = x and ocaversion = " +
        QString::number(desc.ocaVersion),
        pConn->pDatabase, /* ... */);
    // ...
    pConn->close();
}
```

The implementation retrieves a connection handle from the connection factory and, once the query completes, returns it to the pool by calling `close`.

This scheme is sufficient for implementing optimistic locking. For pessimistic locking, we need additional logic. First of all, the connection manager must support transactions; whenever a transaction is started, that is, a client calls `beginTransaction`, the corresponding connection must be bound to the session in which the transaction is used. The implementation does not return this connection to the pool of connections until the client calls either `rollback` or `commit`. An important issue is that the session will break in the middle of a transaction if the corresponding client suddenly disappears. For safe rollback and unlocking of the transaction, we must use a timer that automatically returns locked connections to the pool, rolling back any active transaction after a certain period of inactivity.

Another issue is that, if a database uses row locking, deadlocks can occur. This can happen, for example, if a client starts a transaction, locks some rows, and then disappears. Some database management systems, such as Oracle, have advanced features for recovering automatically from that state: as soon as the database discovers dead sessions, it performs automatic rollback. Of course, this means that rollback can happen to a transaction without the application or library being aware of it.

Zillions of Objects

Databases are the solution of choice for handling large data sets. However, it is not a good idea to create millions of servant instances in a server, each reflecting a particular row in the database—the server will run out of memory for far fewer servants than this. Ice provides *servant locators* to deal with this scenario. First, I will show a simple implementation and then discuss advanced mechanisms that use servant locators, the evictor pattern and the `IceUtil::Cache` class.

Using a Servant Locator

An Ice servant locator provides a mechanism to map a particular request to a specific servant for the duration of the request. For our example application, I will incorporate a servant locator to create this ad-hoc association. As I mentioned in part 1 of this article, an Ice object identity contains both an object's type (in the `category` attribute) as well as an object ID (in the `name` attribute). The locator uses these attributes to determine which servant to instantiate:

```
// C++
Ice::ObjectPtr
CProjectLocator::locate(
    const Ice::Current& current,
    Ice::LocalObjectPtr& cookie)
{
    IceUtil::Mutex::Lock sync(m_syncMutex);
    assert(current.id.category == "project");

    // retrieve the database row id
    std::stringstream ssStream(current.id.name);
    int nId;
    ssStream >> nId;

    CPersistentProject projectData =
        CPersistenceLayer::instance()->
            selectProject(nId);

    CProjectDesc projectDesc =
        CProjectDescMapper::map(projectData);
```

```
    if (projectDesc.m_nId == 0)
    {
        return 0;
    }
    CProjectIPtr retval = new CProjectI();
    retval->m_data = projectDesc;
    return retval;
}
```

The code retrieves the database row ID from the context that is passed to the `locate` function, loads the appropriate data from the database, creates the new servant (which is passed the data), and, finally, returns the servant. The locator is integrated into the program as follows:

```
// C++
CProjectLocatorPtr pProjectLocator =
    new CProjectLocator();
objectAdapter->addServantLocator(
    pProjectLocator, "project");
```

This simple scheme guarantees that database and Ice objects are synchronized as, for each request, a new servant object is created that holds the data. However, this default implementation will be quite slow because, for *every* request, it creates a new object and contacts the database. To improve on this, we can use an evictor or cache class.

Using an Evictor

The evictor pattern retains a fixed number of servants in memory, with fast random access to each servant. In addition, it maintains servants in least-recently-used (LRU) order; if no servant is in memory for a particular request, the evictor evicts the least-recently used servant before allocating a new servant for the request. The implementation still uses a servant locator but, instead of blindly creating a servant for each request, it checks whether a servant is already in memory before instantiating a new one (and possibly evicting another servant).

A sample implementation of the evictor pattern is part of the Ice distribution, in the `demo/book/evictor` directory. The `EvictorBase` class provided by Ice has a default `locate` implementation that we will use. Our locator class derives from `EvictorBase` and implements `add` and `evict` methods that the base class calls to notify the application that it needs a servant and is about to evict a servant.

```
// C++
class CProjectLocator : virtual public EvictorBase
{
public:
    CProjectLocator(
        const Ice::ObjectAdapterPtr& adapter);
protected:
    virtual Ice::ObjectPtr
      add(const Ice::Current& current,
        Ice::LocalObjectPtr& localobject);
    virtual void
      evict(const Ice::ObjectPtr& object,
          const Ice::LocalObjectPtr& localobject);
};
```

We can implement `evict` as an empty function because we do not need take special actions before evicting a servant; `add` is implemented as follows:

```
// C++
Ice::ObjectPtr
CProjectLocator::add(
    const Ice::Current& current,
    Ice::LocalObjectPtr& localobject)
{
    CProjectIPtr retval = 0;
    int nId = atoi(current.id.name.c_str());

    CPersistentProject projectData =
        CPersistenceLayer::instance()->
            selectProject(nId);

    CProjectDesc projectDesc =
        CProjectDescMapper::map(projectData);
    if (projectDesc.m_nId != 0)
    {
        retval = new CProjectI();
        retval->m_data = projectDesc;
    }
    return retval;
}
```

Using an evictor, we create a one-to-one mapping between database entities and Ice servants, at least for those objects that are currently cached. Those cached Ice servants each reflect a single database row. Performance using an evictor is significantly better because servants live longer and we do not access the database for each request.

Unfortunately, the evictor cache is not necessarily consistent with the database: the (logical) deletion of objects and, in turn, of database rows is not handled by the standard evictor implementation. If your application needs to implement object deletion, it must also clean the evictor cache!

One strategy for deletion that works fine if objects can delete themselves is to use a delete flag within the servant, for example:

```
// Slice
interface CProject
{
    void deleteProject();
};
```

The `deleteProject` operation is implemented as:

```
// C++
void
CProjectI::deleteProject(
    const Ice::Current& current)
{
  m_deleted = true;
}
```

When such a logically deleted servant is accessed through the locator, `locate` throws an `ObjectNotExistsException`. Note that this requires a small modification of the `EvictorBase` class to take care of checking the `m_deleted` flag.

Another way of implementing object deletion and keeping the database and evictor cache in sync is to derive from `EvictorBase` and implement a `removeProjectById` function:

```
// C++
void
CProjectEvictor::removeProjectById(int projectId)
{
    Ice.Identity id = new Ice.Identity();
    id.category = "project";
    id.name = projectId.ToString();

    lock (this)
    {
        if (!_map.Contains(id))
        {
            return true;
        }
        _map.Remove(id);
        if (_map.Contains(id))
        {
            return false;
        }
        return true;
    }
}
```

We have to make the `_map` member variable of `EvictorBase` protected to make this work.

Using a small mediator class such as

```
// C++
void
CProjectMediator::removeProject(int id)
{
    m_pProjectLocator->removeProjectById(id);
}
```

prevents your business logic from accessing the Ice-specific code.

All in all, using an evictor for implementing a servant locator has a number of advantages:

- It removes the performance bottleneck of the default servant locator implementation.
- The code is still quite simple and easy to understand and maintain; using the `EvictorBase` class is even easier than creating a standard servant locator implementation.

However, there is a slight drawback to this approach that is hidden in the `EvictorBase` class: the implementation uses a mutex to guard the LRU map. Thus, lookups and database access are serialized within `EvictorBase`. No two servants can be loaded from the database concurrently.

Using the IceUtil::Cache Class

This potential performance bottleneck can be solved by using the `IceUtil::Cache` class. This cache class is thread-safe and does not use a mutex to serialize lookups. The Ice 3.2.1 distribution includes a demo called `CustomEvictor` that shows how to combine the `Cache` class with the standard evictor. If you write your code based on this example, the code will not differ from the standard evictor approach but automatically be fully multi-threaded. Even the `remove` implementation can easily be adapted to this approach. (Unfortunately, this class is only available for C++.)

Complex Database Operations

So far, we have discussed how to retrieve and update descriptors using the corresponding interface: The `CProjectDesc` descriptor was retrieved and modified using the `Project` interface. This works for a lot of use cases but there might be some more complex tasks where this basic scheme does not apply. For example, in our project management application, we might want to find all projects that have a particular member. Depending on the database schema, this might require more complex database operations such as joins, sub-selects, or dynamic selections with run-time construction of the query.

Such operations should not be implemented in the servant itself. Instead, a separate servant can handle such complex tasks in a service-like manner:

```
// Slice
sequence<CProjectDesc> CProjectDescSequence;

interface CProjectInformationService
{
    CProjectDescSequence
     findAllProjectsWithMember(string membername);
};

interface CProject
{
    CProjectDesc describe();
};
```

To keep the function style consistent, we keep using project descriptors as the ultimate structures for passing data. As in part 1 of this article, the mapping between a database row and the corresponding descriptor is implemented by the `describe` operation. Because we now have a second function that also creates such descriptors, it makes sense to extract the mapping functionality into separate classes as mentioned in part 1. Such classes take care of mapping between data structures as retrieved from the database and the structures handled within the application.

For our example, we introduce the class `CProjectDescProvider` which can be retrieved from a global `DescProvider` factory:

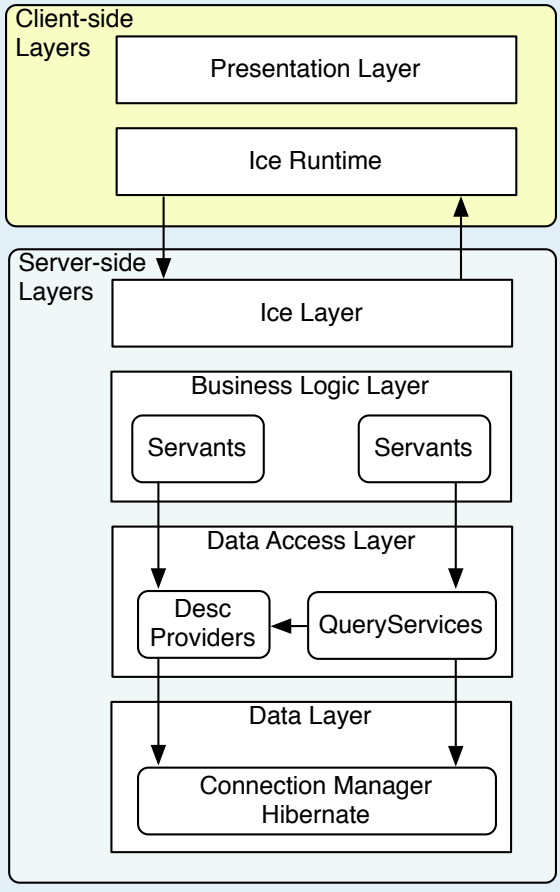
```
// C++
class CProjectDescProvider : DescProviderBase
{
public:
    CProjectDesc get(int id) const;
    bool set(const CProjectDesc& desc);
};
```

This new class is responsible for generating new descriptors based on database rows and for storing modified descriptors back into the database.

To support complex query operations, the implementations of `CProjectI` and `CProjectInformationService` access the same `CProjectDescProvider` instance, which in turn accesses the database to provide descriptors. The servant implementation no longer requires a descriptor member, which makes the corresponding implementation more lightweight.

A further advantage of this single-point data access is that we can easily ensure data consistency, as well as implement a caching layer for descriptors.

Figure 1: Application Layers



Final Words

Based on the generic layered application architecture I introduced in part 1, the integration of all concepts introduced yields Figure 1.

Depending on the complexity of use cases that you need to handle in your application, you may not need all of the features introduced here.

As the two parts of this article show, using the correct implementation techniques for persistence makes it quite easy to use databases and thus to implement 3+-tier applications that use Ice as the communication infrastructure. Ice features such as servant locators let you easily build scalable, high-performance applications.

I hope you will find the implementation techniques I presented here useful for your own development. Please feel free to contact me if you have any questions.

Who's Counting?

Michi Henning, Chief Scientist

Introduction

When developers face a programming task that is either close to the hardware or is critical to performance, they usually reach for C++. C++ permits manipulation of raw memory and, because it supports pointers and pointer arithmetic, enables many algorithms to be implemented more efficiently than a language such as Java that tightly controls access to memory and often requires data to be copied that, with C++, can be manipulated in place. This makes C++ the system programming language of choice in many situations.

One characteristic of C++ is that it encourages allocation of objects on the heap. In particular, whenever the need arises to manipulate data whose size is known only at run time but not at compile time, heap allocation is usually the only option. (And many programming patterns, such as the factory pattern, intrinsically require heap allocation.) However, therein lies a catch: we must carefully balance every call to `new` with a corresponding call to `delete`, and we must balance every call to `new[]` with a corresponding call to `delete[]`. Failure to do this has fatal consequences: too few calls to `delete` or `delete[]` will cause the program to eventually run out of memory, and too many calls to `delete` or `delete[]` will cause the program to die an untimely death. And, as a C++ programmer, I am sure that you will be aware of the numerous other ways for a C++ program to shoot itself in the foot, such as making mismatched calls to `new` and `delete[]` (or `new[]` and `delete`), or failing to deallocate memory when the program's execution path takes an unexpected turn, such as when code takes an early return out of a function or calls a function that unexpectedly throws an exception.

There are other spanners in the memory-management works. One of the biggest problems of explicit memory management is that, as the programmer, you are made responsible for tracking the life time of allocated memory: if one part of the program calls `delete` while another part of the program still holds a pointer to that region of memory and later dereferences that pointer, all hell breaks loose. To make matters worse, the problem usually only shows up far from the place where the damage was done (namely, where the incorrect call to `delete` was made) and instead manifests itself at the point where the no-longer valid memory is used (namely, where the pointer is dereferenced). This can make it extremely difficult to track down the cause of a memory management problem. If your program is threaded and the threads share pointers to allocated memory, the problem gets more complex by at least an order of magnitude, often because it can become very difficult to reproduce a memory management problem from run to run due to timing differences in the execution of threads.

The problems caused by memory management errors are serious. In the past, I have consulted on many projects that were plagued by memory management problems (some to the point where the project ended up being cancelled), and a number of companies make a healthy living from memory management debugging tools, such as IBM Rational Purify. These tools can be invaluable when it comes to tracking down memory management problems. However, the tools only help fix a problem *after* it arises, and only if you happen to have a test case that actually exposes the problem. This is not at all certain for large and complex programs; often, memory management errors remain dormant for years before they are discovered.

A much better way to deal with memory management errors is to make sure that they don't happen in the first place. Doing this not only saves a lot of licensing fees for debugging tools, but also saves your—and usually your manager's—sanity.

As a user of the Ice C++ mapping, you will appreciate its automatic memory management: you never need to call `delete`; instead, the mapping ensures that memory is deallocated at just the right time, namely, as soon as it is no longer needed. (If you are a past user of the CORBA C++ mapping, I am sure that you are doubly appreciative of this—the CORBA C++ mapping makes it ludicrously easy to corrupt or leak memory.) Much of this convenience of the Ice C++ mapping is based on a single concept, namely, smart pointers. Smart pointers provide all the necessary magic that makes it impossible to leak or prematurely deallocate memory, by tracking the life time of allocated objects with reference counts. Leaks are not completely impossible but, to cause them, your code has to deliberately go out of its way.

What many programmers do not realize is that smart pointers are not limited to Ice-related classes and APIs—you can use them just as easily for memory management of your own, application specific classes that have nothing to do with Ice. In this article, I will show you:

- How Ice smart pointers work
- How to use Ice smart pointers with your own classes
- How Ice smart pointers deal with cyclic dependencies in Slice classes

I suggest that you take a good look at these techniques. Doing so will not only simplify your code a great deal, but will almost certainly reduce the number of memory management defects in your project. And, believe me, reference counting is much more fun than bug counting, especially if the counting is being done by your manager…

Smart Pointers: An Overview

Why Bother with Smart Pointers?

Ice smart pointers are successful because they provide a number of features:

- Once you have initialized a smart pointer with the pointer returned by a call to `new`, you never need to call `delete`. Smart pointers automatically track the life time of the allocated object and delete it as soon as the program lets the last smart pointer go out of scope. You can freely copy and assign smart pointers without violating this guarantee.
- Smart pointers are strongly-typed. Just as with ordinary C++ pointers, you cannot accidentally assign a smart pointer to, say, a chair to the smart pointer for an automobile (unless the chair derives from the automobile, which is unlikely).
- Smart pointers are polymorphic. Just as with ordinary C++ pointers, you can pass a smart pointer to a derived instance where a smart pointer to a base instance is expected. Similarly, if you have a smart pointer to a base class that, at run time, points at a derived instance, invoking member functions via the smart pointer uses the usual semantics for virtual function calls and invokes the most-derived implementation of the member function. (In other words, smart pointers behave like normal C++ pointers.)
- Smart pointers are exception-safe. Even in the presence of exceptions that change the flow of control in unexpected ways, smart pointers correctly track allocated memory.
- Smart pointers are thread-safe. Internally, assignment and copying of smart pointers is interlocked such that you need not use any additional synchronization when using smart pointers from different threads.
- Smart pointers are efficient. The size of a smart pointer is the same as that of a raw C++ pointer (four bytes on a 32-bit platform), so smart pointers have no space overhead. Per class instance, the reference counting implementation adds only 12-bytes (on 32-bit Linux and Windows), which is insignificant even for classes containing little data. Run-time performance of smart pointers is excellent as well, with negligible impact on execution speed. (Ice uses smart pointers all throughout its implementation, with no negative performance impact, even for speed-critical code.)

These features make smart pointer an immensely useful tool: they make it much easier to write your code and eliminate many common mistakes that become manifest only at run time. In turn, this means that your software will be faster to develop and will contain fewer defects. Simply put, smart pointers reduce development and maintenance cost.

Implementation Overview

The basic mechanism of smart pointers is simple: each class instance holds a reference count that starts out with the value zero when the instance is created. When the instance's address (that is, its C++ pointer) is used to initialize a smart pointer, the reference count in the class instance is set to one. Thereafter, every time a smart pointer to the object is assigned or copied to another smart pointer, the reference count in the instance goes up by one and, every time a smart pointer to an instance goes out of scope (or has its value changed by assignment), the reference count is decremented by one. When the reference count reaches zero, which happens when the last smart pointer to the instance goes out of scope, the instance deletes itself by calling `delete this`.

You can find an overview of this idea in the client-side C++ mapping chapter in the Ice Manual and, if you are not familiar with the idea of reference counting and smart pointers, I suggest you read the section on smart pointers before proceeding. For this article, we are more interested in the implementation of smart pointers, so I will proceed to that.

Here is the basic implementation of smart pointers, which consists of two parts. The first part adds a reference count to every class that is to work with smart pointers:

```
// C++
namespace IceInternal
{
    namespace GC
    {
        extern IceUtil::RecMutex gcRecMutex;
    }

    class GCShared
    {
    private:
        int _ref;

    public:
        GCShared() : _ref(0) {}
        GCShared(const GCShared&) : _ref(0) {}
        virtual ~GCShared() {}
        GCShared& operator=(const GCShared&}
        {
            return *this;
        }
        virtual void __incRef()
        {
             Mutex::Lock sync(GC::gcRecMutex);
             ++_ref;
        }
        virtual void __decRef()
        {
            Mutex::Lock sync(GC::gcRecMutex);
            if(--_ref == 0)
            {

                delete this;
            }
        }
    };
}
```

Note that I have taken some liberties with this code—if you look at the actual implementation of `IceInternal::GCShared` in the Ice source code, you will find that it looks somewhat different: for simplicity, I have inlined some member functions that are out of line in the actual implementation, and I have omitted a number of details that deal with platform-specific and efficiency issues. The

code I show is sufficient to understand how reference counting and smart pointers work without getting lost in too much implementation detail.

The purpose of `GCShared` is to act as a base class from which reference-counted classes derive. This style of reference counting is known as *intrusive reference counting* because, for a class to be reference-counted, its definition must be modified. There is also *non-intrusive reference* counting, which keeps the reference count in a class instance that is separate to the one being reference-counted. The advantage of non-intrusive reference counting is that it allows you to reference-count classes without modifying their definition; the disadvantage is that non-intrusive reference counting requires more memory (which is why Ice uses intrusive reference counting).

If you define a class in Slice, the `slice2cpp` compiler arranges for the corresponding generated C++ class to derive from `GCShared`. For example, consider the following minimal Slice class:

```
// Slice
class Simple {};
```

That's as simple a class as we can get. For this class, the `slice2cpp` compiler generates the following C++ definition:

```
// C++
class Simple : public virtual Ice::Object
{
public:
    Simple() {}
    // ...

protected:
    virtual ~Simple() {}
};
```

Again, I have omitted some irrelevant detail here. The important point is that the generated class `Simple` derives from `Ice::Object`. In turn, if you look through the Ice header files, you will find that `Ice::Object` derives from `IceInternal::GCShared`:

```
// C++
namespace Ice
{
    class Object : public IceInternal::GCShared
    {
        // ...
    };
}
```

In other words, every Slice-generated class provides a reference count. You will also notice that the generated `Simple` class has a protected destructor. Reference-counted classes *must* be allocated on the heap, and making the destructor protected enforces this: with a protected destructor, `Simple` instances can be allocated only on the heap, but not on the stack or in a static variable; incorrect allocation causes a compile-time error.

The second part of the implementation is the smart pointer class itself, which uses the `__incRef` and `__decRef` member functions in `GCShared` to adjust the reference count. If you find the code that follows somewhat intimidating, don't despair—despite looking a little complex, it is actually very simple. Again, the actual implementation of this template class in the Ice source code differs, in order to reduce the size of the generated code; the version I show here is functionally equivalent and more readable.

```
// C++
namespace IceInternal
{
    template<typename T>
    class Handle {
    private:
        T* _p;

    public:
        Handle(T* p = 0) : _p(p) {
            if(_p)
                _p->__incRef();
        }
        Handle(const Handle& h) {
            if(_p = h._p)
                _p->__incRef();
        }
        ~Handle() {
            if(_p)
                _p->__decRef();
        }
        T* operator->() {
            if(!_p)
                throw NullHandleException(
                            __FILE__, __LINE__);
            return _p;
        }
        T& operator*() {
            if(!_p)
                throw NullHandleException(
                            __FILE__, __LINE__);
            return *_p;
        }
        operator bool() const {
            return _p;
        }
        Handle& operator=(const Handle& h) {
            if(_p != h->_p) {
                if(_p)
                    _p->__incRef();
                T* ptr = _p;
                _p = h->_p;
                if(ptr)
                    ptr->__decRef();
            }
            return *this;
        }
        Handle& operator=(T* p) {
            if(_p != h->_p) {
                if(_p)
                    _p->__incRef();
                T* ptr = _p;
```

```
                _p = h->p;
                if(ptr)
                    ptr->__decRef();
            }
            return *this;
        }
    };
}
```

Before we look at how this machinery works, let's look at what the `slice2cpp` compiler generates for our `Simple` class:

```
// C++
typedef IceInternal::Handle<Simple> SimplePtr;
```

This says that `SimplePtr` is a template instance of `IceInternal::Handle` for `Simple` class instances. If you look at the `Handle` class, you will find that it contains a private member `_p` of type `T*`. For the template instantiation with `Simple` as the template parameter, this means that `SimplePtr` contains a private member of type `Simple*`, that is, a raw C++ pointer to a `Simple` instance that lives on the heap.

To use the magic of smart pointers, we can write:

```
// C++
{
    SimplePtr s = new Simple;
    // ...
} // Instance is deleted automagically
```

Immediately after instantiation with `new`, the reference count of the `Simple` instance is zero. As soon as the constructor of `s` runs, it increases the reference count to one. Once `s` goes out of scope, its destructor runs. The destructor calls `__decRef` on the underlying `Simple` instance, which in turn decrements the reference count to zero and, as a result, calls `delete this`.

Similar considerations apply if you step through scenarios that involve copying and assignment of smart pointers: the `Handle` class takes care of maintaining the reference count as appropriate, and the `GCShared` base class provides the `__decRef` member function that causes the instance to delete itself once the final smart pointer to it goes out of scope.

Similarly, if the code leaves the enclosing scope of a smart pointer because of an exception (or takes an early return), the destructor of the smart pointer still runs, so it is impossible to leak memory if the execution path takes an unexpected turn.

As an added bonus, if you dereference a null smart pointer, the `Handle` template throws a `NullPointerException`, which gives your code a chance to recover. (In contrast, if you dereference a null C++ pointer, your program typically dies with a core dump.)

The `operator bool` of the `Handle` template allows you to test whether a smart pointer is null:

```
// C++
SimplePtr s = ...;
if(s)
{
    // s is non-null.
}
```

Playing by the Rules

Smart pointers require you to play by two simple rules. The first is that you *must* allocate reference-counted classes on the heap. This is necessary because the `GCShared` destructor calls `delete` and, of course, `delete` works only for heap-allocated instances.

The second rule is that, if you use smart pointers, you should not mix them with ordinary pointers. This avoids the potential problem of having a raw pointer point at an instance beyond its life time and the code doing undefined things if the raw pointer is dereferenced thereafter.

A corollary to the second rule is that, if you pass a class instance to and from functions, you should pass its *smart pointer* by value (or, for input parameters, by constant reference), instead of passing a reference or pointer to the instance itself. To illustrate why, let us consider a few examples:

```
// C++
SimplePtr getSimple();
```

`getSimple` returns a smart pointer *by value*. Doing this has the advantage that it is safe to ignore the return value. For example, you can call `getSimple` as follows without incurring a memory leak:

```
// C++
getSimple(); // No leak here.
```

`getSimple` returns a `SimplePtr` instance that, because it is not used by caller, is a temporary instance that is destroyed by the compiler as soon as `getSimple` returns. In turn, the destructor of `SimplePtr` ensures that the instance is deallocated as soon as the statement completes (assuming there are no other smart pointers to the same instance in the code). So, even though the returned smart pointer refers to a heap-allocated instance, this code does not cause a memory leak.

Similarly, in the implementation of `getSimple`, we can write:

```
// C++
SimplePtr
getSimple()
{
    return new Simple; // No leak here.
}
```

Because the return type is `SimplePtr`, the compiler constructs a temporary `SimplePtr` instance and initializes it with the return value from `new` (which sets the reference count of the instance to one). This ensures that the heap-allocated `Simple` instance is

destroyed as soon as the caller discards its last smart pointer to that instance (which, as we saw in the preceding example, may happen immediately if the caller ignores the return value).

Most compilers will avoid an additional copy of the return value:

```
// C++
SimplePtr s = getSimple();
// No extra copy here with most compilers.
```

A naïve compiler would create a `SimplePtr` instance inside the implementation of `getSimple` and construct the caller's variable `s` by invoking its copy constructor. However, compilers that implement the *return value optimization* (RVO) avoid this additional copy and instead initialize the caller's variable `s` directly instead of copying a temporary. (Most compilers implement RVO these days.) Many modern compilers also implement the *named return value optimization* (NRVO), which avoids a temporary even if `getSimple` uses a named variable as the return value:

```
// C++
SimplePtr
getSimple()
{
    SimplePtr r = new Simple();
    // ...
    return r; // No copy here with NRVO.
}
```

Input parameters should be passed by constant reference, for example:

```
// C++
void
setSimple(const SimplePtr& s)
{
    // ...
}
```

Note that `setSimple` accepts a *constant reference* to a `SimplePtr` instead of accepting the smart pointer by value. Either way is correct, but passing smart pointers by constant reference is slightly more efficient because it avoids making a copy of the smart pointer on the stack. In turn, this avoids unnecessarily locking, incrementing, and unlocking the instance's reference count when the function is called, and then locking, decrementing, and unlocking the reference count again when the function completes.

Another advantage of passing input smart pointers by constant reference (or value) is that the following code does not cause a memory leak:

```
// C++
setSimple(new Simple);
```

This code does not contain a leak because the compiler uses the constructor of `SimplePtr` to create a temporary smart pointer, which it passes to `setSimple`. That way, the newly allocated instance is immediately taken care of by a smart pointer and therefore cannot leak.

For output parameters, you should pass class instances by (non-constant) reference to their smart pointer:

```
// C++
bool findSimple(KeyType key, SimplePtr& s)
{
    // ...
}
```

Similar arguments apply to output parameters as for input parameters and return values, so I won't provide separate examples for this case.

Using Smart Pointers with Non-Slice Classes

If you have classes that are not generated from Slice classes, you can still use smart pointers with them. For this purpose, Ice provides the `IceUtil::Shared` class and the `IceUtil::Handle` template. `IceUtil::Shared` serves the same purpose as `IceInternal::GCShared`, but omits a few things that are internal to the Ice C++ mapping implementation. (In addition, it cannot deal with cyclic references, which I will discuss shortly.) Similarly, the `IceUtil::Handle` template is functionally equivalent to `IceInternal::Handle`, but omits a few things that are internal to the mapping and not relevant to application code.

So, suppose you have an arbitrary class, and you would like to automate memory management for instances of that class. As an example, suppose you have the following `Person` class (not generated from any Slice definition):

```
// C++
class Person
{
public:
    // Constructor, destructor, etc. here...

    // Various member functions here...

private:
    // Private members here...
};
```

To make this class suitable for use with smart pointers, all you need to do is publicly derive the class from `IceUtil::Shared`, and define a smart pointer for it:

```
// C++
class Person : public IceUtil::Shared
{
public:
    // Constructor, destructor, etc. here...

    // Various member functions here...

private:
    // Private members here...
};

typedef IceUtil::Handle<Person> PersonPtr;
```

Having done this, you can use heap-allocated `Person` instances just like any other Slice-generated class and enjoy the benefits of automatic deallocation:

```
// C++
{
    PersonPtr p = new Person;
    // ...
} // Person is deleted automagically.
```

The benefits of smart pointers become particularly apparent when you consider member variables. For example, suppose your `Person` class also contains two pointers to `Person` instances representing the mother and father of the person. Further, let's also assume that these instances must be dynamically allocated (for example, because the father and mother are not always known at construction time of a `Person` instance). Without smart pointers, your `Person` class would look, in outline, something like this:

```
// C++
class Person
{
public:
    Person(): _mother(0), _father(0) {}
    virtual ~Person() {
        if(_mother)
        {
            delete _mother;
        }
        if(_father)
        {
            delete _father;
        }
    }

    // Copy constructor and assignment operator
here...

    // Various member functions here...

private:
    Person* _mother;
    Person* _father;
};
```

This code initializes the two raw pointer members to null in the constructor and deletes the corresponding `Person` instances in the destructor if the pointers are non-null at the time. Code such as this is typical in many C++ programs. However, the above only shows the tip of the iceberg. For real-world classes, you also have to worry about copy construction and assignment because the compiler-generated default versions perform memberwise copy and assignment, which are almost always wrong for classes that contain pointers to heap-allocated instances. This means that you either need to disable copy construction and assignment, or implement them to perform the appropriate memory management activities. Doing this is not only tedious but also error-prone, particularly for threaded programs that need to deal with race conditions.

Now compare this with the code using smart pointers:

```
// C++
class Person;
typedef IceUtil::Handle<Person> PersonPtr;

class Person : public IceUtil::Shared
{
public:
    Person() {}
    virtual ~Person() {}

    // No need to define explicit copy constructor
    // and assignment operator—the defaults are
    // fine.

    // Various member functions here...

private:
    PersonPtr _mother;
    PersonPtr _father;
};
```

This implementation is free from memory-management artifacts. Because the member variables are smart pointers, when a `Person` instance is destroyed, so are the person's mother and father instances (provided these instances are not pointed at by smart pointers elsewhere in the program). In addition, copying and assignment are exception-safe and thread-safe.

In effect, the second implementation takes care of all thread- and memory-management responsibilities of the first implementation, at essentially zero cost in terms of development effort and run-time overhead. I hope that this example is sufficient to convince you of the value of smart pointers. They truly make code development far safer and easier. And, if you use smart pointers for a while, you will realize how much "development friction" they remove: things that used to be complicated and error-prone suddenly seem to happen by themselves, leaving you to focus more on the application logic, instead of having to worry about memory management and threading issues.

As a final note, if you use multiple inheritance with your classes, be sure to use virtual inheritance from `IceUtil::Shared`. Otherwise, you end up with two separate instances of the base class, and two separate reference counts, which does not work.

Cyclic Dependencies

A general problem with reference counting is that cyclic references cause leaks. In practice, this can occur, for example, with the factory pattern, where the factory creates a class and stores its smart pointer, and the class stores a smart pointer back the factory:

```
// C++
class Factory;
typedef IceUtil::Handle<Factory> FactoryPtr;

class Item : public IceUtil::Shared
{
public:
    Item(int id, const FactoryPtr& f) :
         _myId(id), _myFactory(f) {
        // ...
    }

    void destroy() {
        _myFactory->removeMe(_myId);
    }

private:
    int _myId;
    FactoryPtr _myFactory;
};
typedef IceUtil::Handle<Item> ItemPtr;

class Factory : public IceUtil::Shared
{
public:
    ItemPtr create(int id) {
        ItemPtr i = new Item(id, this);
        _items[id] = i;
        return i;
    }

    void removeMe(_myId) {
        _items.erase(_myId);
    }

    void destroy() {
        _items.clear();
    }

private:
    map<int, ItemPtr> _items;
};
```

Here, the factory and its items point at each other so we cannot simply let the factory's smart pointer go out of scope because that would leave all the cycles intact and leak memory for the factory and all its items. (See the following example for an illustration of how cycles prevent the reference count from ever reaching zero.) Instead, we have to break the cycle somehow. In this case, the `destroy` member function on the factory clears the factory's map, thereby breaking the cycle. Before letting the factory's smart pointer go out of scope, the caller must call the factory's `destroy` member function:

```
// C++
{
    FactoryPtr f = ...;

    // Use factory...

    f->destroy(); // Break cycles.
} // No leak here.
```

The cyclic dependency problem can also pop up in more general form, especially if you manipulate graphs of objects. For example:

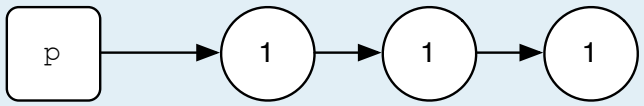
```
// C++
class MyClass;
typedef IceUtil::Handle<MyClass> MyClassPtr;


class MyClass : public IceUtil::Shared
{
public:
    MyClassPtr link;
    // ...
};
```

Each `MyClass` instance contains a pointer to another instance, so instances can form graphs. As long as the graph does not contain a cycle, everything works fine. For example, the following code fragment creates a linked list of instances:

```
// C++
{
    MyClassPtr p = new MyClass;
    p->link = new MyClass;
    p->link->link = new MyClass;
} // No leak here.
```

This creates the three-node graph shown in Figure 1. (Arrows indicate the smart pointers, and the number inside each node indicates the node's reference count.)

Figure 1: A Graph of Three Nodes



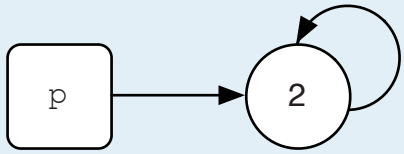
When execution of the preceding code reaches the end of the enclosing block, the destructor of `p` runs, which decrements the first node's reference count to zero. This causes the first node to self-destruct by calling `delete this`, which decrements the second node's reference count and causes that node to self-destruct as well, which in turn causes the final node to self-destruct.

However, if the graph of nodes contains a cycle, we have a more interesting situation. Any cycle at all will do (even a cycle involving only a single node):

```
// C++
{
    MyClassPtr p = new MyClass;
    p->link = p;
} // Leak!
```

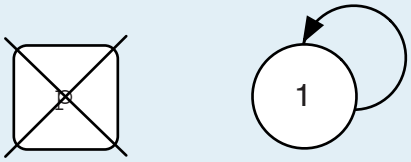
This code creates the situation shown in Figure 2, where a node points at itself.

Figure 2: A Single-Node Cycle



Ordinary reference counting cannot take care of this situation. Consider what happens when `p` goes out of scope. The destructor of `p` calls `__decRef` on the node, which decrements the reference count to one, as shown in Figure 3.

Figure 3: Reference Count after Destruction of `p`



The problem here is that node's reference count never drops back to zero so, unless we do something else, the node is never deallocated. Note that the same thing can happen if a node does not point at itself immediately, but points at itself via a number of intermediate nodes. In this case, we end up with all the nodes in the cycle with their reference count at one.

As for the factory example, the usual way to deal this problem is to add a `destroy` method to the class that the caller must call before allowing the final smart pointer to the graph to go out of scope. The implementation of `destroy` traverses the graph, looking for cycles and, if it detects a cycle, assigns zero to the smart pointer causing the cycle. For the preceding `MyClass` example, `destroy` could be implemented as follows:

```
// C++
void
MyClass::destroy()
{
    set<MyClass*> visited;
    while(link)
    {
       link = visited.insert(
            link.get()).second ? link->link : 0;
    }
}
```

The `destroy` implementation maintains a `visited` set of addresses of `MyClass` instances. The `get` member function of a smart pointer returns the address of the pointed-at instance, so `link.get` returns that address. `destroy` follows the chain of links, inserting each one into the `visited` set. If the insert returns true, `destroy` proceeds to the next link. If the insert returns false, the address of the current link is already in the set and, therefore, the current link causes a cycle. In this case, the code assigns zero to the smart pointer causing the cycle and then returns.

The caller must remember to call `destroy` before letting the last smart pointer to the graph go out of scope:

```
// C++
{
    MyClassPtr p = new MyClass;
    p->link = p;
    // ...
    p->destroy();
} // No more leaks, even with a cycle.
```

Exactly how to write the `destroy` function varies with each class and the actions `destroy` must take vary depending on the structure of the graph and how many smart pointers are contained in each node. But the basic approach is the same: `destroy` performs a traversal of the graph that visits every node and remembers each node visited so far; if it finds that a particular smart pointer points at a previously-visited node, it clears that smart pointer. The terminating condition of the traversal depends on whether the graph can contain more than one cycle. If so, `destroy` must continue to visit nodes until it has examined all nodes; otherwise, if the graph can contain only a single cycle (as in the preceding example), `destroy` can terminate as soon as it has broken that cycle.

Slice Classes with Cyclic Dependencies

Consider the following Slice definition:

```
// Slice
class Node
{
    Node n;
};
```

The generated code for this looks something like the following:

```
// C++
class Node;
typedef IceInternal::Handle<Node> NodePtr;

class Node : public Ice::Object
{
public:
    NodePtr n;
    // ...
};
```

This looks suspiciously like the `MyClass` example we saw earlier: instances of type `Node` can form cycles. However, if a graph of *Slice-generated* classes contains a cycle, Ice does *not* leak memory.

Ice includes a garbage collector that finds and deallocates all nodes that are no longer reachable from the program's code and, therefore, are part of a cycle that was left behind at some point. To see why such a garbage collector is necessary, consider the following interface:

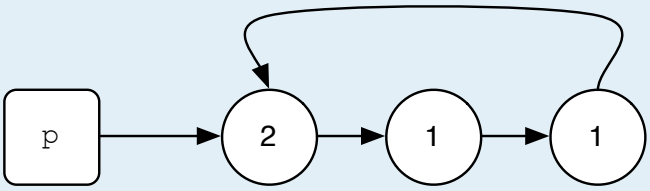
```
// Slice
interface Graph
{
    Node makeGraph();
};
```

The `makeGraph` operation returns a graph of nodes that can contain cycles. For example, the implementation of `makeGraph` in the server might be as follows:

```
// C++
class GraphI : public Graph
{
    virtual NodePtr makeGraph(const Ice::Current&)
    {
        NodePtr p = new Node;
        p->link = new Node;
        p->link->link = new Node;
        p->link->link->link = p;
        return p;
    }
};
```

This returns the cyclic graph shown in Figure 4.

Figure 4: Graph Returned by `makeGraph`



Obviously, the implementation of `makeGraph` cannot call a `destroy` method to break the cycle because it intentionally returns a cyclic graph to the client. On the other hand, the Ice run time, after marshaling the graph back to the client cannot call `destroy` either because other parts of the server might still hold smart pointers to nodes in the graph and may not wish it to be destroyed just yet.

The Ice garbage collector deals with this problem nicely because it only deallocates cycles once they are no longer reachable from other parts of the program, that is, once they have become fully disconnected. For the preceding example, the collector will reclaim the cycle only if no other smart pointer in the server still points at a node in the graph; otherwise, the cycle will be reclaimed once the last smart pointer to a node in the cycle goes out of scope, at which point no part of the cycle can still be reached via smart pointers in the application code.

By default, the garbage collector makes a collection pass whenever you destroy a communicator:

```
// C++
CommunicatorPtr comm = initialize(argc, argv);

// Lots of cyclic graphs left behind here...

comm->destroy(); // All garbage reclaimed here.
```

Collecting garbage during destruction of a communicator is useful mainly if you use a memory management tool such as Purify because this prevents the tool from reporting lots of memory leaks on program exit. Of course, if your code repeatedly leaves cycles of instances behind but does not destroy a communicator for some time, chances are that the program will run out of memory. To deal with this situation, Ice provides two additional ways for you to run the garbage collector:

- You can set the property `Ice.GC.Interval` to a (positive) time interval *n* in seconds. Doing so causes the Ice run time to make a collection pass once every *n* seconds.
- You can explicitly call the static function `Ice::collectGarbage`. Doing so instructs the garbage collector to immediately collect all garbage that has accumulated up to this point. The call is synchronous, that is, `collectGarbage` does not return until the collection pass is complete.

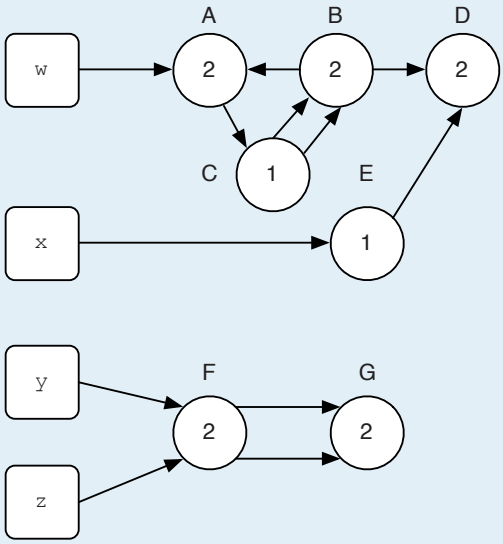
Of the two options, the first one is less intrusive because it requires no code changes at all. In contrast, the second option requires you to insert calls to `collectGarbage` at appropriate points in your program. Finding these appropriate points can be difficult because, depending how the code is structured and how graphs are used, it may be difficult or impossible to know whether garbage exists at a particular point along the execution path. However, explicit calls to `collectGarbage` are still useful in situations where, at a specific point in the code, you know that garbage is likely to exist.

In general, however, it is easiest to simply set `Ice.GC.Interval` to make a collection pass once very few seconds or minutes (depending on how quickly garbage accumulates in your program). You can tune the time interval by observing the collector's statistics, as described below.

Collection Algorithm

The algorithm for how to implement garbage collection in conjunction with reference-counting smart pointers is not well known and not widely published, so I am presenting it here. I will not go through all the details of the implementation, but give you an illustration of the algorithm instead. (Note that you do not need to understand how the algorithm works to use the Ice garbage collector.) To give us a concrete example to work with, consider the graph of nodes in Figure 5.

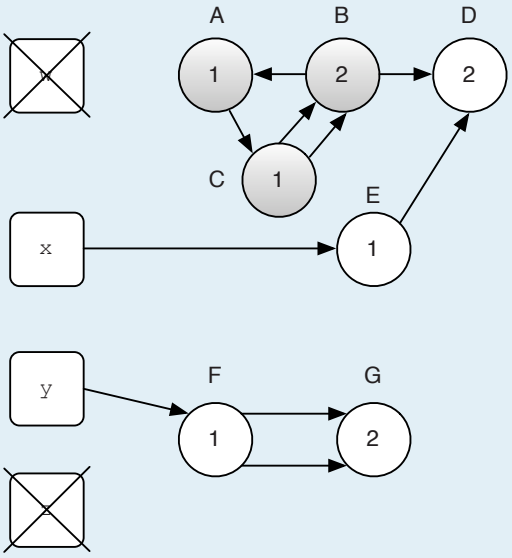
Figure 5: A Graph of Nodes



Such a graph could arise from a class that contains two smart pointers in each instance (as was the case for the `Person` class I discussed earlier). As before, boxes represent smart pointer variables in the application code, circles represent nodes, and the reference count is shown inside each node. The labels outside the nodes denote their identity.

Suppose the application lets the variables `w` and `z` go out of scope. After the destructors of `w` and `z` have adjusted the reference counts, we end up with the situation shown in Figure 6.

Figure 6: The Graph from Figure 5 after Adjusting Reference Counts



The shaded nodes are garbage because they can no longer be reached from the program. Note that node D is not garbage because it can still be reached via `x` and the intermediate node E. Also note that the garbage collector cannot simply detect the cycle involving A, B, and C and delete these three nodes because doing so would leave D's reference count too high: removing the cycle also involves adjusting D's reference count.

Here is an overview of the garbage collection algorithm that is used by Ice. The algorithm relies on a set `S` that, at any point in time, contains the addresses of all instantiated classes. (Whenever a class instance is created, its address is added to `S`; whenever a class instance is deleted, its address is removed from `S`.) On each run, the garbage collector goes through the following steps:

- Initialize a map `M` of node–count pairs by adding to `M` the address of each node in `S` together with the node's current reference count.
- Initialize a map `R` of node–count pairs to contain the address of all nodes that are directly (but not recursively) reachable from nodes in `S` via data members that can point at a node. Set the count of each entry in `R` to the number of times the node is pointed at by entries in `S`.
- For each entry in `R`, subtract that entry's count from the corresponding reference count in `M`.
- Remove all entries from `M` that have a reference count greater than zero, as well as all entries that are (recursively) reachable from entries with a reference count greater than zero.
- Any entries that remain in `M` cannot be reached from the application. Deallocate all nodes in `M`, together with all nodes that are recursively reachable from these nodes.
- For each node that is still live and was pointed at by a deallocated node, adjust that node's reference count by decrementing it once for each time it was pointed at by a deallocated node in `M`.

The preceding is a fancy way of expressing a simple idea. Intuitively, the algorithm works by counting how many times each node is pointed at by other nodes. For each time a node is pointed at by another node, the algorithm decrements a copy of the reference count of the node. If, after going through all existing nodes, a node has a reference count of zero, the node can be reached only from other nodes and therefore is part of a cycle; such a node can be deleted. On the other hand, if after going through all the existing nodes, a node has a reference count greater than zero, it is pointed at (directly or indirectly) by something other than another node (namely, a smart pointer instance in the application); such a node is still in use and cannot be deleted.

To make this more concrete, let's run through the graph in Figure 6 and see how the collector deals with this graph.

- Initially, S contains A, B, C, D, E, F, and G.
- The collector initializes M to contain the following pairs: <A,1>, <B,2>, <C,1>, <D,2>, <E,1>, <F,1>, <G,2>.
- The collector initializes R to contain: <A,1>, <B,2>, <C,1>, <D,2>, <G,2>. (E and F do not appear in R because they are not pointed at by any node in S.)
- The collector reduces the reference counts in M by the corresponding entry in R, so M now contains <A,0>, <B,0>, <C,0>, <D,0>, <E,1>, <F,1>, and <G,0>.
- We now remove all entries with a reference count greater than zero from M, as well as all entries that can (recursively) be reached from these entries. This leaves <A,0>, <B,0>, and <C,0> in M.
- The three instances that remain in M are garbage and can be deallocated.

If you are interested in how all this is implemented, take a look at src/Ice/GC.cpp in the Ice for C++ source distribution. It contains the code that implements the preceding algorithm in the CollectGarbage function.

Smart Pointer Implementation for Garbage Collection

For the garbage collector to do its job, it needs to know which class instances exist at the time it runs. A look at GCShared shows us how this works:

```
// C++
class GCShared
{
public:
    virtual void __incRef() { ... }
    virtual void __decRef() { ... }
    // ...
protected:
    void __gcIncRef();
    void __gcIncRef();
};
```

Note that __incRef and __decRef member functions of GCShared are virtual member functions, so derived classes can override their implementation. As we saw earlier, the default implementations of these functions simply increment and decrement the reference count, respectively. If a Slice class does not contain any data member that is a Slice class, the default implementations of these functions are sufficient. However, if a Slice class (recursively) contains a data member that is a Slice class, the slice2cpp compiler overrides these implementations. For example, for our Node class, the compiler generates the following (I have inlined the code for simplicity):

```
class Node
{
public:
    NodePtr n;
    virtual void __incRef() {
        __gcIncRef();
    }

    virtual void __decRef() {
        __gcDecRef();
    }
};
```

In other words, if a class contains a class member, its __incRef and __decRef member functions behave differently. Here is the (somewhat simplified) implementation of these functions:

```
// C++
void
IceInternal::GCShared::__gcIncRef()
{
    RecMutex::Lock lock(GC::gcRecMutex);
    if(_ref == 0)
    {
        gcObjects.insert(this);
    }
    ++_ref;
}

void
IceInternal::GCShared::__gcDecRef()
{
    RecMutex::Lock lock(GC::gcRecMutex);
    if(--_ref == 0)
    {
        gcObjects.erase(this);
        delete this;
    }
}
```

Looking at __gcIncRef, you can see that, the first time a class instance's reference count is incremented, the address of the instance is added to a gcObjects set. This set corresponds to the set S we saw in the description of the collector algorithm. Similarly, __gcDecRef removes the address of a class instance from this set once the reference count drops to zero. In this way, the collector's gcObjects set always contains the address of all existing class instances that (recursively) have at least one class member.

The reason why slice2cpp overrides the __incRef and __decRef member functions is that only classes that (recursively) contain at least one class member can participate in a cycle: a class that does not contain a class member cannot point at any other class instance and, therefore, not be part of a cycle (other than being pointed at by an instance of a different type that *is* part of a cycle). By only adding those class instances to gcObjects that potentially can form cycles, we reduce the amount of work that the collector does during each collection pass because, that way, there are fewer instances to examine.

To do its job, the garbage collector needs to know which class instances are reachable from a particular class instance. Again, slice2cpp contributes the relevant functionality by overriding the implementation of a member function of GCShared:

```
// C++
class GCShared
{
public:
    virtual void __gcReachable(
                  IceInternal::GCCountMap&) const;
};
```

The job of `__gcReachable` is to add all class instances and how many times these instances are pointed at to the map that is passed to the function. The default implementation of `__gcReachable` does nothing. However, if a class contains class members, `slice2cpp` overrides the default implementation. For example, for our `Node` class, `slice2cpp` generates code equivalent to the following:

```
// C++
void
Node::__gcReachable(IceInternal::GCCountMap& cm)
{
    if(n)
    {
        IceInternal::GCCountMap::iterator pos
            = cm.find(n.get());
        if(pos == cm.end())
            cm[n.get()] = 1;
        else
            ++pos->second;
    }
}
```

When the collector calls `__gcReachable` on a particular `Node` instance, the instance reachable from that node is whatever its member smart pointer `n` currently points at. The `get` member function of `NodePtr` returns the address of its underlying instance, so the preceding code either inserts the address of the node that is reachable from the current node with a count of one (if the reachable node is not already in the map) or increments the count stored in the map for that node (if the reachable node is already in the map).

This is how the collector builds the map `R` of node–count pairs that we saw earlier in the description of the algorithm: it simply calls `__gcReachable` on every instance in the `gcObjects` set of all instances; each invocation either adds one or more reachable nodes to the map, or if one or more nodes are already in the map, increments the count of how many times they are pointed at by other nodes.

For our simple `Node` class, the code generated by `slice2cpp` is quite simple because `Node` only contains a single smart pointer to other nodes, and contains that smart pointer directly. For more complex classes, where sub-members of the class are smart pointers, the generated code is a little more complex because it has to descend into those sub-members to the appropriate level to add the addresses of reachable instances to the map. As an exercise, you may want to examine the code that is generated for the following Slice definition:

```
// Slice
class Person;
sequence<Person> PersonSeq;

struct Lineage
{
    PersonSeq maternalLine;
    PersonSeq paternalLine;
};

class Person {
    string name;
    Lineage line;
};
```

It is instructive to trace through the various levels of virtual functions to see how the garbage collector maintains an accurate picture of the graph of instances.

A final piece of the puzzle is the question of how the collector manages to adjust the reference count of a node that is pointed at by a garbage node (such as node D in Figure 6). In that case, it needs to both clear the smart pointer inside node B (which points at node D) and decrement D's reference count. `slice2cpp` again generates helper code that the garbage collector can call to achieve this:

```
// C++
class GCShared
{
public:
    virtual void __gcClear() {}
};
```

The default implementation of `__gcClear` does nothing. However, for classes that (recursively) contain class members, `slice2cpp` overrides the default implementation. For example, for our `Node` class, it generates code equivalent to the following:

```
// C++
void
Node::__gcClear()
{
    if(!n)
        return;
    if(n.get()->__usesClasses())
    {
        n.get()->__decRefUnsafe();
        n.__clearHandleUnsafe();
    }
    else
    {
        n = 0;
    }
}
```

The `__decRefUnsafe` member function of `GCShared` decrements the reference count of an instance without first acquiring a lock. Similarly, the `__clearHandleUnsafe` member function of `IceInternal::Handle` sets the smart pointer to

null without first acquiring a lock (for efficiency reasons). The `__usesClasses` member function is generated by `slice2cpp` and returns true if a class (recursively) contains class members. If so, the collector has to "reach inside" the instances and adjust the smart pointer and reference count using `__decRefUnsafe` and `__clearHandleUnsafe`; otherwise, the pointed-at instance is a leaf node without class members, and the collector can reclaim it by simply assigning zero to its smart pointer, which causes the leaf node's destructor to run immediately.

Performance

Now that we know how the garbage collection machinery works, let's have a look at how well it performs.

In terms of memory footprint, the cost of the collector is small. The code for the collector fits into around 10kB of code on a 32-bit machine (not counting the generated code, which is also small, on the order of a few dozen bytes per class type). For its data structures, the main overhead of the collector is the memory required for the `gcObjects` set, which is 24 bytes per instance using the GNU STL implementation. (Sizes for other implementations will be similar). During a collection pass, the collector builds the map of node–count pairs, which temporarily adds an additional 48 bytes per instance. (This memory is reclaimed at the end of each pass.)

In terms of execution speed, the collector also has minimal impact. The first observation is that all smart pointer operations that increment or decrement the reference count are interlocked on a single lock, `gcRecMutex`. So, if different threads concurrently manipulate smart pointers (not necessarily for the same class instance), they are serialized on this lock: only one increment or decrement of a reference count can take place at a time.

In practice, serializing these operations does not cause problems because the critical region inside `__incRef` and `__decRef` is very small. It is very rare for a thread to find the mutex already locked and stall; almost always, the mutex is available and the cost of locking and unlocking it is negligible.

Another observation is that the garbage collector locks `gcRecMutex` during each collection pass. Doing this is necessary because, while the collector builds the map of node–count pairs, it must ensure that reference counts cannot change. Of course, this means that, while the collector performs a collection run, all threads that concurrently manipulate smart pointers stall until the collection run completes. This is the inevitable price of using the collector.

Fortunately, the collector is very efficient, so it finishes each collection pass very quickly. The exact run-time cost of performing a collection pass depends on a number of factors, including how many class instances exist, how many of these instances are actually part of a cycle, and how highly connected the instances are.

The collection algorithm itself performs in $O(n \log n)$ time, where n is the total number of instances in the `gcObjects` set, independent of the degree of connectivity of the nodes. However, the total time required also depends on node connectivity because the collector, for each live node, needs to compute the set of recursively reachable instances.

The worst-case time for computing the reachable set arises for a fully connected graph, where each node points at all other nodes (including itself), and all nodes can also be reached from the program. Such a graph contains $O(n^2)$ smart pointers, and computing the recursively reachable set requires $O(n \log n)$ time. The algorithm computes the recursively reachable set once for each live node, so the overall worst-case time bound of the collector is $O(n^2 \log n)$. (Note, however, that deallocating the same graph without a garbage collector would still incur $O(n^2)$ run-time cost.)

For graphs with lower connectivity, such that the total number of smart pointers is proportional to n, the overall cost of the collector is $O(n \log n)$. (Because such a graph contains $O(n)$ smart pointers, deallocating it without a garbage collector would still incur $O(n)$ run-time cost.)

Overall, the collector is most expensive if it is invoked on graphs with high degree of connectivity that do not contain garbage. For example, for a graph with 3,000 nodes, each containing 3,000 smart pointers to every node, and with all nodes reachable from the program, a collection run requires 10 seconds on a 2.1GHz Pentium 4 with 1GB of memory, even though it does not deallocate anything. (This graph contains 9 million edges, so the collector examines around 300 nodes and 900,000 edges per second). Once the entire graph becomes garbage, it takes 3.5 seconds to collect it. (This includes the cost of calling `delete` and invoking destructors.) However, such a graph represents a pathological case: realistic data structures are not as highly connected.

A more realistic example involves 300,000 instances that form 150,000 two-instance cycles. While all 300,000 nodes are still reachable from the program, each pass of the collector requires 1.5 seconds. (The graph contains 600,000 edges, so the collector examines 200,000 nodes and 400,000 edges per second.) Once all 150,000 cycles are garbage, it takes 1.3 seconds to deallocate them all, including the cost of calling `delete` and invoking destructors. (Only one third of that time is spent inside the collector, the remainder of the time is spent invoking destructors and inside `delete`, so that cost would also be incurred without a garbage collector.)

Keep in mind also that the preceding figures were obtained with rather substandard hardware; with a faster machine and a faster memory bus, execution times decrease accordingly. The upshot of all this is that, in practice, the collector is simply too fast to notice, except in pathological cases. However, even in such cases, much of the cost is the cost of calling `delete` and running destructors, and that cost is incurred whether you use a collector or not; by reducing the frequency at which the collector runs, you can usually amortize the execution time of each run to an acceptable level.

If you want to watch how the collector performs, you can set the property `Ice.Trace.GC`. With a setting of 1, when the last communicator is destroyed, the Ice run time prints the number of collection runs, the total number of instances examined and reclaimed, and the number of milliseconds spent in the collector. With a setting of 2, the run time also prints statistics for each individual collection run.

Last Words

Having read this far, you may wonder why, instead of building our own garbage collector, we didn't simply bundle an available collector (such as the Boehm collector) with Ice. There are a number of reasons:

- The Boehm collector replaces the global `::new` operator and requires application code to replace the default allocator of STL containers. This makes its use non-transparent to application code.
- If an application uses `malloc` and `free` or uses a custom allocator (often inadvertently, by linking against a third-party library), it must ensure that data allocated by `malloc` or the custom allocator does not contain pointers to data that was allocated on the garbage collector's heap. Depending on the application, this can be difficult to enforce.
- The Boehm collector examines raw memory to perform liveness analysis. The code to do this requires adjustment for each CPU architecture and C++ compiler, and the effort to port the Boehm collector to a new hardware platform or compiler is non-trivial. For Ice, which must run on a variety of architectures, including embedded platforms such as mobile phones, porting the Boehm collector would incur substantial additional cost for each new platform that we want to support.

Apart from these points, the most significant disadvantage of the Boehm collector is that it affects the entire process. Ice is provided as a library that your applications link against; if we were to include the Boehm collector with Ice, this would force garbage collection on all of your application, for all its data. However, doing so would be a significant change in policy that would be unacceptable to many companies: ZeroC cannot dictate its customers that they all must use a general-purpose garbage collector (which, of course, collects *everything*, not just Slice class instances).

In contrast, the Ice collector is completely portable, transparent to application code, and only deals with Slice classes, so it suffers none of these disadvantages.

I suggest that you give smart pointers a good look. They ease development considerably and make it much less likely for memory management bugs to cause grief, either during testing or in the field. This not only saves you money, but also saves your sanity and allows you to put more brain cycles towards the application logic, instead of worrying about memory management. So, lean back, let Ice do the counting, and enjoy your leak-free program.

FAQ Corner

In each issue of our newsletter, we present a few frequently-asked questions about Ice. The questions and answers are taken from our support forum at http://www.zeroc.com/forums/ and deal with specific problems that developers tend to encounter, and for which the answer may not be readily apparent from reading the documentation. We hope that you will find the hints and explanations in this section useful.

Q: What are Ice Clients and Servers?

From a user's perspective, a client is usually some interactive application program that connects to a server, such as a web browser or email client. Similarly, a server is usually seen as some program that runs on a remote machine and serves one or more clients, such as a web or email server. While this view is valid for simple request–response systems, in the context of Ice, it is too loose.

An *Ice client* is any program that invokes an operation on an Ice object. In other words, clients are active entities that issue requests for service to servers. Conversely, an *Ice server* is any program that implements an operation. In other words, servers are passive entities that provide service in response to operation invocations from clients.

On occasion, this definition of Ice client and server does not neatly line up with the more loose view. For example, IceStorm is generally thought of as a server. However, when IceStorm delivers an event to a subscriber, it is IceStorm that invokes an operation on the object registered for a topic, and it is the subscriber that implements the operation. In other words, a "client" program, such as a stock ticker that subscribes to stock updates distributed by IceStorm, is really an Ice server, and IceStorm is an Ice client.

The preceding example illustrates that the terms client and server really refer to the role that is played by each. Moreover, that role applies only within the context of a particular operation invocation. The client role is played by the invoking end, and the server role is played by the responding end.

A *pure* Ice client is a program that only ever invokes operations and never responds to them, and a *pure* Ice server is a program that only ever responds to operation invocations and never makes invocations of its own. (Any program that does not create an object adapter is guaranteed to be a pure client.) However, many Ice applications are not pure clients or servers. In particular, a server that makes an invocation on an Ice proxy in order to implement some operation (typically, on a helper object in some other server) is no longer a pure server. Instead, it acts as a server to the original client, but also acts as a client to the server implementing the helper object.

Of course, because Ice is location transparent, the server has no idea where the helper object is implemented. In fact, the helper object may be implemented by the server itself, in which case the server ends up being both its own client and server. Alternatively, it is possible for the helper object to be implemented by the original client. If so, the helper object is known as a *callback object*. Callback objects are no different from ordinary objects; it just so happens that callback objects are implemented by clients and that the server often (but not always) invokes an operation on a callback object as part of executing an operation invoked by the client that provided the callback object; such invocations are known as *nested callbacks*.

From an Ice perspective, there is nothing special about callbacks. They are simply operation invocations like any other invocation. (However, for nested callbacks, the threading model is important. How deeply callbacks can be nested depends on the size of the client- and server-side thread pools. You can look at the demo in `demo/Ice/callback` if you are interested in experimenting with nested callbacks.)

In summary, an Ice client is any program that invokes an operation, and an Ice server is any program that responds to an operation invocation. If a program both invokes operations and responds to them, it is both a client and a server.