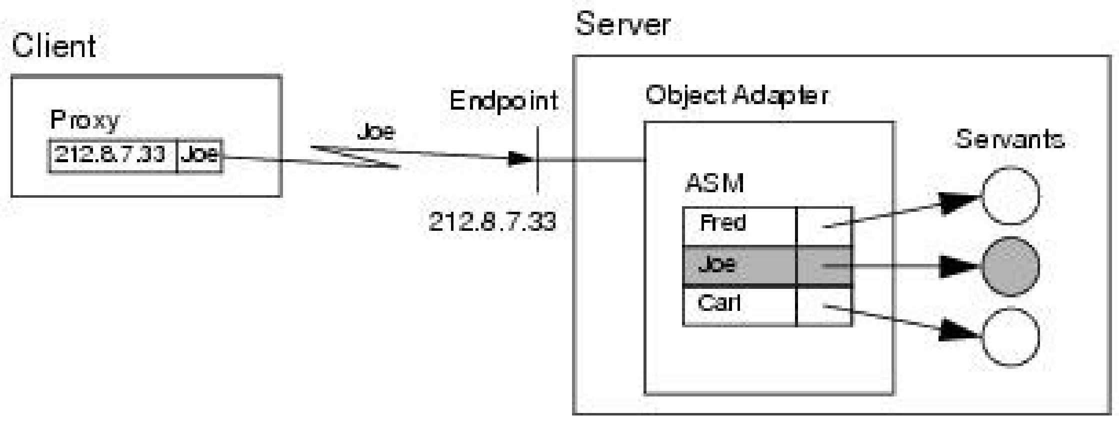
Advanced Server

Active Servant Map

- Each object adapter maintains a data structure known as the active servant map (ASM).
  - Lookup table that maps object identities to servants:
    - In C++ - smart pointer to the corresponding servant
- The process of associating a request via a proxy to the correct servant is known as binding.



Servant Activation and Deactivation

- Servant activation - making the presence of a servant for a particular Ice object known to the Ice run time.
  - Activating a servant adds an entry to the active servant map.
- The inverse operation is known as servant deactivation. Deactivating a servant removes an entry for a particular identity from the ASM.

```
module Ice {
    local interface ObjectAdapter {
        // ...
        Object* add(Object servant, Identity id);
        Object* addWithUUID(Object servant);
        Object  remove(Identity id);
        Object  find(Identity id);
        Object  findByProxy(Object* proxy);
        // ...
    };
};
```

Adapter States

- An object adapter has a number of processing states:
  - holding - incoming requests are not dispatched to servants
  - active - the adapter accepts incoming requests and dispatches them to servants
  - inactive - in this state, the adapter has conceptually been destroyed

```
module Ice {
 local interface ObjectAdapter {
     // ...
     void activate();
     void hold();
     void waitForHold();
     void deactivate();
     void waitForDeactivate();
     void isDeactivated();
     void destroy();
     // ...
 };
};
```

Using Multiple Object Adapters

- Needed when:
  - You need fine-grained control over which objects are accessible.
    - You could have an object adapter with only secure endpoints to restrict access to some administrative objects, and another object adapter with non-secure endpoints for other objects.
    - You can firewall a particular port, so objects associated with the corresponding endpoint cannot be reached unless the firewall rules are satisfied.
  - You need control over the number of threads in the pools for different sets of objects in your application.
  - You want to be able to temporarily disable processing new requests for a set of objects. This can be accomplished by placing an object adapter in the holding state.

The Ice::Current Object

```
module Ice {
    local dictionary<string, string> Context;

    enum OperationMode { Normal, \Nonmutating, \Idempotent };

    local struct Current {
        ObjectAdapter   adapter;
        Connection      con;
        Identity        id;
        string          facet;
        string          operation;
        OperationMode   mode;
        Context         ctx;
        int             requestId;
    };
};
```

Servant Locators

- Using an adapter’s ASM to map Ice objects to servants has a number of design implications:
  - Each Ice object is represented by a different servant.
  - All servants for all Ice objects are permanently in memory.
- Requirements:
  - The server has sufficient memory available to keep a separate servant instanti-ated for each Ice object at all times.
  - The time required to initialize all the servants on startup is acceptable.
- Ice offers two APIs that help you scale servers to larger numbers of objects:
  - Servant locators
  - Default servants

Servant Locator Interface

```
module Ice {
    local interface ServantLocator {
        ["UserException"]
        Object locate(Current curr,
                      out LocalObject cookie);

        ["UserException"]
        void finished(Current curr,
                      Object servant,
                      LocalObject cookie);

        void deactivate(string category);
    };
};
```

Threading Guarantees for Servant Locators

- The Ice run time guarantees that every operation invocation that involves a servant locator is bracketed by calls to locate and finished, that is, every call to locate is balanced by a corresponding call to finished.
- In addition, the Ice run time guarantees that locate, the operation, and finished are called by the same thread.
  - It allows you to use locate and finished to implement thread-specific pre- and post-processing around operation invocations.
- Beyond this, the Ice run time provides no threading guarantees for servant locators. In particular:
  - It is possible for invocations of locate to proceed concurrently (for the same object identity or for different object identities)
  - It is possible for invocations of finished to proceed concurrently
  - It is possible for invocations of locate and finished to proceed concurrently

Servant Locator Registration

```
module Ice {
    local interface ObjectAdapter {
        // ...
     void addServantLocator(ServantLocator locator,
                            string category);
     ServantLocator removeServantLocator(string category);
     ServantLocator findServantLocator(string category);
        // ...
    };
};
```

- You can register exactly one servant locator for a specific category. Attempts to call addServantLocator for the same category more than once raise an **`AlreadyRegisteredException`**.
- You can register different servant locators for different categories, or you can register the same single servant locator multiple times (each time for a different category).
- It is legal to register a servant locator for the empty category. Such a servant locator is known as a default servant locator.

Call Dispatch Semantics

1. Look for the identity in the ASM. If the ASM contains an entry, dispatch the request to the corresponding servant.
2. If the category of the incoming object identity is non-empty, look for a default servant that is registered for that category.
3. If the category of the incoming object identity is empty, or no default servant could be found for the category in step 2, look for a default servant that is registered for the empty category.
4. If the category of the incoming object identity is non-empty and no servant could be found in the preceding steps, look for a servant locator that is regis-tered for that category.
5. If the category of the incoming object identity is empty, or no servant locator could be found for the category in step 4, look for a default servant locator (that is, a servant locator that is registered for the empty category).
6. Raise **`ObjectNotExistException`** or **`FacetNotExistException`** in the client.

Implementing a Simple Servant Locator

```
struct Details {
    // Lots of details about the entry here...
};

interface PhoneEntry {
    idempotent Details getDetails();
    idempotent void updateDetails(Details d);
    // ...
};

struct SearchCriteria {
    // Fields to permit searching...
};

interface PhoneBook {
    idempotent PhoneEntry* search(SearchCriteria c);
    // ...
};
```

Implementing a Simple Servant Locator (cont.)

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

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

    virtual void finished(const Ice::Current&        c,
                          const Ice::ObjectPtr&      servant,
                          const Ice::LocalObjectPtr& cookie);

    virtual void deactivate(const std::string& category);
};
```

Implementing a Simple Servant Locator (cont.)

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

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

    // We have the state, instantiate a servant and return it.
    //
    return new PhoneEntryI(d);
}
```

Using the `category` Member of the Object Identity

```
Ice::ObjectPtr
MySeravntLocator::locate(const Ice::Current&  c,
                         Ice::LocalObjectPtr& cookie)
{
    // Get the object identity. (We use the name member
    // as the database key.)
    //
    std::string name = c.id.name;
    std::string realId = c.id.name.substr(1);
    try {
        if (name[0] == 'd') {
            // The request is for a directory.
            //
            DirectoryDetails d = DB_lookup(realId);
            return new DirectoryI(d);
        } else {
            // The request is for a file.
            //
            FileDetails d = DB_lookup(realId);
            return new FileI(d);
        }
    } catch (DatabaseNotFoundException&) {
        return 0;
    }
}
```

Using the `category` Member of the Object Identity

```
class DirectoryLocator : public virtual Ice::ServantLocator {
public:
    virtual Ice::ObjectPtr locate(const Ice::Current&  c,
                                  Ice::LocalObjectPtr& cookie)
    { // Code to locate and instantiate a directory here... }
    virtual void finished(const Ice::Current&        c,
                          const Ice::ObjectPtr&      servant,
                          const Ice::LocalObjectPtr& cookie){}
    virtual void deactivate(const std::string& category){}
};

class FileLocator : public virtual Ice::ServantLocator {
public:
    virtual Ice::ObjectPtr locate(const Ice::Current&  c,
                                  Ice::LocalObjectPtr& cookie)
    { // Code to locate and instantiate a file here... }
    virtual void finished(const Ice::Current&        c,
                          const Ice::ObjectPtr&      servant,
                          const Ice::LocalObjectPtr& cookie){}
    virtual void deactivate(const std::string& category){}
};
// Register two locators, one for directories and
// one for files.
adapter>addServantLocator(new DirectoryLocator(), "d");
adapter>addServantLocator(new FileLocator(), "f");
```

Using Cookies

- Occasionally, it can be useful to be able to pass information between locate and finished.
  - For example, the implementation of locate could choose among a number of alternative database backends, depending on load or availability and, to properly finalize state, the implementation of finish might need to know which database was used by locate.
  - To support such scenarios, you can create a cookie in your locate implementation; the Ice run time passes the value of the cookie to finished after the operation invocation has completed.

Using Cookies

```
class MyCookie : public virtual Ice::LocalObject {
public:
    // Whatever is useful here...
};

typedef IceUtil::Handle<MyCookie> MyCookiePtr;
class MyServantLocator : public virtual Ice::ServantLocator {
public:
    virtual Ice::ObjectPtr locate(const Ice::Current&  c,
                                  Ice::LocalObjectPtr& cookie)
    {
        cookie = new MyCookie(...);
        return new PhoneEntryI;
    }

    virtual void finished(const Ice::Current&         c,
                          const Ice::ObjectPtr&       servant,
                          const Ice::LocalObjectPtr& cookie)
    {
        MyCookiePtr mc = MyCookiePtr::dynamicCast(cookie);
        // Use information in cookie to clean up...
    }

    virtual void deactivate(const std::string& category);
};
```

Default Servants

```
module Ice {
    local interface ObjectAdapter {
        void addDefaultServant(Object servant,
                               string category);
        Object removeDefaultServant(string category);
        Object findDefaultServant(string category);
        // ...
    };
};
```

Default Servants - Guidelines

- Object identity is the Key
  - When an incoming request is dispatched to the default servant, the target object identity is provided in the Current argument. The name field of the identity typically supplies everything the default servant requires in order to satisfy the request. For instance, it may serve as the key in a database query, or even hold an encoded structure in some proprietary format that your application uses to convey more than just a string.
  - Naturally, the client can also pass arguments to the operation that assist the default servant in retrieving whatever state it requires. However, this approach can easily introduce implementation artifacts into your Slice interfaces, and in most cases the client should not need to know that the server is implemented with a default servant. If at all possible, use only the object identity.

Default Servants - Guidelines

- Minimize Contention
  - For better scalability, the default servant’s implementation should strive to elimi-nate contention among the dispatch threads. As an example, when a database holds the default servant’s state, each of the servant’s operations usually begins with a query. Assuming that the database API is thread-safe, the servant needs to perform no explicit locking of its own. With a copy of the state in hand, the implementation can work with function-local data to satisfy the request.

Default Servants - Guidelines

- Combine Strategies
  - The ASM still plays a useful role even in applications that are ideally suited for default servants. For example, there is no need to implement a singleton object as a default servant: if there can only be one instance of the object, implementing it as a default servant does nothing to improve your application’s scalability.
  - Applications often install a handful of servants in the ASM while servicing the majority of requests in a default servant. For example, a database application might install a singleton query object in the ASM while using a default servant to process all invocations on the database records.

Default Servants - Guidelines

- Categories Denote Interfaces
  - In general, all of the objects serviced by a default servant must have the same interface. If you only need a default servant for one interface, you can register the default servant with an empty category string. However, to implement several interfaces, you will need a default servant implementation for each one. Further-more, you must take steps to ensure that the object adapter dispatches an incoming request to the appropriate default servant. The category field of the object identity is intended to serve this purpose.
  - For example, a process control system might have interfaces named Sensor and Switch. To direct requests to the proper default servant, the application uses the symbol Sensor or Switch as the category of each object’s identity, and registers corresponding default servants having those same categories with the object adapter.

Default Servants - Guidelines

- Plan for the Future
  - If you suspect that you might eventually need to implement more than one inter-face with default servants, we recommend using a non-empty category even if you start out having only one default servant. Adding another default servant later becomes much easier if the application is already designed to operate correctly with categories.
- Throw exceptions
  - If a request arrives for an object that no longer exists, it is the default servant’s responsibility to raise **`ObjectNotExistException`**.

Default Servants - Guidelines

- Handle **`ice_ping`**
  - The default implementation of **`ice_ping`** that the servant inherits from its skeleton class always succeeds. For servants that are registered with the ASM, this is exactly what we want; however, for default servants, **`ice_ping`** must fail if a client uses a proxy to a no-longer existent Ice object. To avoid getting successful **`ice_ping`** invocations for non-existent Ice objects, you must override **`ice_ping`** in the default servant. The implementation must check whether the object identity for the request denotes a still-existing Ice object and, if not, raise **`ObjectNotExistException`**.
- Throw exceptions
  - If a request arrives for an object that no longer exists, it is the default servant's responsibility to raise **`ObjectNotExistException`**.

Incremental Initialization

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

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

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

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

    return servant;
}
```

Incremental Initialization

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

Incremental Initialization

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

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

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

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

Incremental Initialization

```
Ice::ObjectPtr
MyServantLocator::locate(const Ice::Current&  c,
                         Ice::LocalObjectPtr&)
{
    IceUtil::Mutex::Lock lock(_m);
    // Check if we have instantiated a servant already.
    Ice::ObjectPtr servant = c.adapter.find(c.id);

    if (!servant) {      // We don't have a servant already
        // Instantiate a servant.
        ServantDetails d;
        try {
            d = DB_lookup(c.id.name);
        } catch (const DB_error&) {
            return 0;
        }
        servant = new PhoneEntryI(d);

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

    return servant;
}
```

Default Servants

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

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

    return ns;
}
```

Overriding `ice_ping`

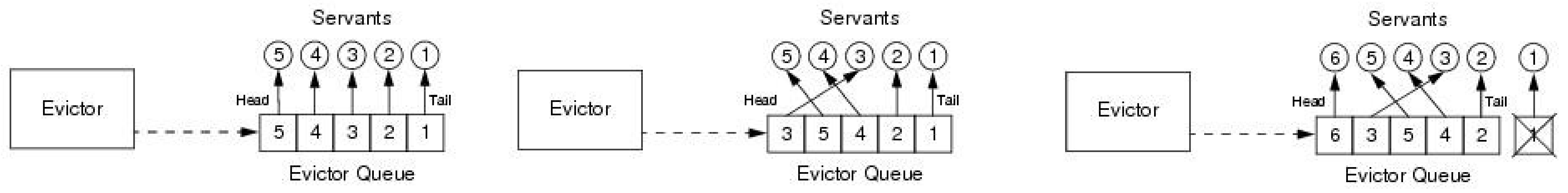
```
void
Filesystem::DirectoryI::ice_ping(const Ice::Current& c) const
{
    try {
        d = DB_lookup(c.id.name);
    } catch (const DB_error&) {
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);
    }
}
```

Servant Evictors

- A variation on the previous theme and particularly interesting use of a servant locator is as an evictor. An evictor is a servant locator that maintains a cache of servants:
  - Whenever a request arrives (that is, locate is called by the Ice run time), the evictor checks to see whether it can find a servant for the request in its cache. If so, it returns the servant that is already instantiated in the cache; otherwise, it instantiates a servant and adds it to the cache.
  - The cache is a queue that is maintained in least-recently used (LRU) order.: the least-recently used servant is at the tail of the queue, and the most-recently used servant is at the head of the queue. Whenever a servant is returned from or added to the cache, it is moved from its current queue position to the head of the queue, that is, the “newest” servant is always at the head, and the “oldest” servant is always at the tail.

Servant Evictors

- The queue has a configurable length that corresponds to how many servants will be held in the cache; if a request arrives for an Ice object that does not have a servant in memory and the cache is full, the evictor removes the least-recently used servant at the tail of the queue from the cache in order to make room for the servant about to be instantiated at the head of the queue.



Evictor Implementation in C++

```
class EvictorBase : public Ice::ServantLocator {
public:
    EvictorBase(int size = 1000);

    virtual Ice::ObjectPtr locate(const Ice::Current& c,
                                  Ice::LocalObjectPtr& cookie);
    virtual void finished(const Ice::Current& c,
                          const Ice::ObjectPtr&,
                          const Ice::LocalObjectPtr& cookie);
    virtual void deactivate(const std::string&);

protected:
    virtual Ice::ObjectPtr add(const Ice::Current&,
                               Ice::LocalObjectPtr&) = 0;
    virtual void evict(const Ice::ObjectPtr&,
                       const Ice::LocalObjectPtr&) = 0;

private:
    // ...
};

typedef IceUtil::Handle<EvictorBase> EvictorBasePtr;
```

Evictor Implementation in C++

- Two main data structures are needed for implementation:
  1. A map that maps object identities to servants, so we can efficiently decide whether we have a servant for an incoming request in memory or not.
  2. A list that implements the evictor queue. The list is kept in LRU order at all times.
- The evictor map does not only store servants but also keeps track of some administrative information:
  1. The map stores the cookie that is returned from add, so we can pass that same cookie to evict.
  2. The map stores an iterator into the evictor queue that marks the position of the servant in the queue. Storing the queue position is not strictly necessary—we store the position for efficiency reasons because it allows us to locate a servant's position in the queue in constant time instead of having to search through the queue in order to maintain its LRU property.
  3. The map stores a use count that is incremented whenever an operation is dispatched into a servant, and decremented whenever an operation completes.

Evictor Implementation in C++

- The need for the use count deserves some extra explanation:
  - Suppose a client invokes a long-running operation on an Ice object with identity I.
  - In response, the evictor adds a servant for I to the evictor queue.
  - While the original invocation is still executing, other clients invoke operations on various Ice objects, which leads to more servants for other object identities being added to the queue.
  - As a result, the servant for identity I gradually migrates toward the tail of the queue.
  - If enough client requests for other Ice objects arrive while the operation on object I is still executing, the servant for I could be evicted while it is still executing the original request.

Evictor Implementation in C++

- By itself, this will not do any harm.
- However, if the servant is evicted and a client then invokes another request on object I, the evictor would have no idea that a servant for I is still around and would add a second servant for I.
- However, having two servants for the same Ice object in memory is likely to cause problems, especially if the servant's operation implementations write to a database.
- The use count allows us to avoid this problem: we keep track of how many requests are currently executing inside each servant and, while a servant is busy, avoid evicting that servant.
    - As a result, the queue size is not a hard upper limit: long-running operations can temporarily cause more servants than the limit to appear in the queue.
    - However, as soon as excess servants become idle, they are evicted as usual.

Evictor Implementation in C++

- The evictor queue does not store the identity of the servant.
  - Instead, the entries on the queue are iterators into the evictor map.
  - This is useful when the time comes to evict a servant: instead of having to search the map for the identity of the servant to be evicted, we can simply delete the map entry that is pointed at by the iterator at the tail of the queue.
  - We can get away with storing an iterator into the evictor queue as part of the map, and storing an iterator into the evictor map as part of the queue because both **`std::list`** and **`std::map`** do not invalidate forward iterators when we add or delete entries (except for invalidating iterators that point at a deleted entry, of course).
- Finally, our locate and finished implementations will need to exchange a cookie that contains a smart pointer to the entry in the evictor map.
  - This is necessary so that finished can decrement the servant’s use count.

Evictor Implementation in C++

```
class EvictorBase : public Ice::ServantLocator {
    // ...
private:
    struct EvictorEntry;
    typedef IceUtil::Handle<EvictorEntry> EvictorEntryPtr;

    typedef std::map<Ice::Identity, EvictorEntryPtr> EvictorMap;
    typedef std::list<EvictorMap::iterator> EvictorQueue;

    struct EvictorEntry : public Ice::LocalObject
    {
        Ice::ObjectPtr servant;
        Ice::LocalObjectPtr userCookie;
        EvictorQueue::iterator queuePos;
        int useCount;
    };

    EvictorMap _map;
    EvictorQueue _queue;
    Ice::Int _size;

    IceUtil::Mutex _mutex;

    void evictServants();
};
```

Evictor Implementation in C++

```
EvictorBase::EvictorBase(Ice::Int size) : _size(size)
{
    if (_size < 0)
        _size = 1000;
}
```

Evictor Implementation in C++

```
Ice::ObjectPtr
EvictorBase::locate(const Ice::Current& c,
                    Ice::LocalObjectPtr& cookie)
{
    IceUtil::Mutex::Lock lock(_mutex);

    //
    // Check if we have a servant in the map already.
    //
    EvictorEntryPtr entry;
    EvictorMap::iterator i = _map.find(c.id);
    if (i != _map.end()) {
        //
        // Got an entry already, dequeue the entry from
        // its current position.
        //
        entry = i->second;
        _queue.erase(entry->queuePos);
    } else
```

Evictor Implementation in C++

```
    {
        //
        // We do not have an entry. Ask the derived class to
        // instantiate a servant and add a new entry to the map.
        //
        entry = new EvictorEntry;
        entry->servant = add(c, entry->userCookie); // Down-call
        if (!entry->servant) {
            return 0;
        }
        entry->useCount = 0;
        i = _map.insert(std::make_pair(c.id, entry)).first;
    }

    //
    // Increment the use count of the servant and enqueue
    // the entry at the front, so we get LRU order.
    //
    ++(entry->useCount);
    entry->queuePos = _queue.insert(_queue.begin(), i);

    cookie = entry;

    return entry->servant;
}
```

Evictor Implementation in C++

```
void
EvictorBase::finished(const Ice::Current&,
                      const Ice::ObjectPtr&,
                      const Ice::LocalObjectPtr& cookie)
{
    IceUtil::Mutex::Lock lock(_mutex);

    EvictorCookiePtr ec = EvictorCookiePtr::dynamicCast(cookie);

    // Decrement use count and check if
    // there is something to evict.
    //
    --(ec->entry->useCount);
    evictServants();
}
```

Evictor Implementation in C++

```
void
EvictorBase::evictServants()
{
    //
    // If the evictor queue has grown larger than the limit,
    // look at the excess elements to see whether any of them
    // can be evicted.
    //
    EvictorQueue::reverse_iterator p = _queue.rbegin();
    int excessEntries = static_cast<int>(_map.size() - _size);

    for (int i = 0; i < excessEntries; ++i) {
        EvictorMap::iterator mapPos = *p;
        if (mapPos->second->useCount == 0) {
            evict(mapPos->second->servant,
                  mapPos->second->userCookie);
            p = EvictorQueue::reverse_iterator(
                    _queue.erase(mapPos->second->queuePos));
            _map.erase(mapPos);
        } else
            ++p;
    }
}
```

Evictor Implementation in C++

```
void
EvictorBase::deactivate(const std::string& category)
{
    IceUtil::Mutex::Lock lock(_mutex);

    _size = 0;
    evictServants();
}
```

Evictor Implementation in C++

```
void
EvictorBase::evictServants()
{
    // More aggressive version
    // If the evictor queue has grown larger than the limit,
    // try to evict servants until the length drops
    // below the limit.
    //
    EvictorQueue::reverse_iterator p = _queue.rbegin();
    int numEntries = static_cast<int>_map.size();

    for (int i = 0; i < numEntries && _map.size() > _size; ++i) {
        EvictorMap::iterator mapPos = *p;
        if (mapPos->second->useCount == 0) {
            evict(mapPos->second->servant,
                  mapPos->second->userCookie);
            p = EvictorQueue::reverse_iterator(
                    _queue.erase(mapPos->second->queuePos));
            _map.erase(mapPos);
        } else
            ++p;
    }
}
```

Using Servant Evictors

- Using a servant evictor is simply a matter of deriving a class from **`EvictorBase`** and implementing the add and evict methods.
  - You can turn a servant locator into an evictor by simply taking the code that you wrote for locate and placing it into **`add`** - **`EvictorBase`** then takes care of maintaining the cache in least-recently used order and evicting servants as necessary.
  - Unless you have clean-up requirements for your servants (such as closing network connections or database handles), the implementation of evict can be left empty.

Using Servant Evictors

- One of the nice aspects of evictors is that you do not need to change anything in your servant implementation: the servants are ignorant of the fact that an evictor is in use. This makes it very easy to add an evictor to an already existing code base with little disturbance of the source code.
- Evictors can provide substantial performance improvements over default servants: especially if initialization of servants is expensive (for example, because servant state must be initialized by reading from a network), an evictor performs much better than a default servant, while keeping memory requirements low.

The Ice Threading Model

- Ice is inherently a multi-threaded platform.
- There is no such thing as a single-threaded server in Ice.
  - As a result, you must concern yourself with concurrency issues: if a thread reads a data structure while another thread updates the same data structure, havoc will ensue unless you protect the data structure with appro-priate locks.
  - In order to build Ice applications that behave correctly, it is important that you understand the threading semantics of the Ice run time.

Introduction to Thread Pools

- A thread pool is a collection of threads that the Ice run time draws upon to perform specific tasks. Each communicator creates two thread pools:
  - The client thread pool services outgoing connections, which primarily involves handling the replies to outgoing requests and includes notifying AMI callback objects.
    - If a connection is used in bidirectional mode, the client thread pool also dispatches incoming callback requests.
  - The server thread pool services incoming connections.
    - It dispatches incoming requests and, for bidirectional connections, processes replies to outgoing requests.

Introduction to Thread Pools

- By default, these two thread pools are shared by all of the communicator’s object adapters. If necessary, you can configure individual object adapters to use a private thread pool instead.
- If a thread pool is exhausted because all threads are currently dispatching a request, additional incoming requests are transparently delayed until a request completes and relinquishes its thread; that thread is then used to dispatch the next pending request. Ice minimizes thread context switches in a thread pool by using a leader-follower implementation.

Configuring Thread Pools

- Each thread pool has a unique name that serves as the prefix for its configuration properties:
  - **`name.Size`**
    - This property specifies the initial size of the thread pool. If not defined, the default value is one.
  - **`name.SizeMax`**
    - This property specifies the maximum size of the thread pool. If not defined, the default value is one. If the value of this property is less than that of name.Size, this property is adjusted to be equal to **`name.Size`**.
  - **`name.SizeWarn`**
    - This property sets a high water mark; when the number of threads in a pool reaches this value, the Ice run time logs a warning message. If you see this warning message frequently, it could indicate that you need to increase the value of name.SizeMax. The default value is zero, which disables the warning.

Configuring Thread Pools

- **`name.StackSize`**
  - This property specifies the number of bytes to use as the stack size of threads in the thread pool. The operating system’s default is used if this property is not defined or is set to zero.
- **`name.Serialize`**
  - Setting this property to a value greater than zero forces the thread pool to serialize all messages received over a connection. It is unnecessary to enable serialization for a thread pool whose maximum size is one because such a thread pool is already limited to processing one message at a time. For thread pools with more than one thread, serialization has a negative impact on latency and throughput. If not defined, the default value is zero.
- **`name.ThreadIdleTime`**
  - This property specifies the number of seconds that a thread in the thread pool must be idle before it terminates. The default value is 60 seconds if this property is not defined. Setting it to zero disables the termination of idle threads.

Configuring Thread Pools

- For configuration purposes, the names of the client and server thread pools are **`Ice.ThreadPool.Client`** and **`Ice.ThreadPool.Server`**, respectively. As an example, the following properties establish the initial and maximum sizes for these thread pools:
  - **`Ice.ThreadPool.Client.Size=1`**
  - **`Ice.ThreadPool.Client.SizeMax=10`**
  - **`Ice.ThreadPool.Server.Size=1`**
  - **`Ice.ThreadPool.Server.SizeMax=10`**
- To monitor the thread pool activities of the Ice run time, you can enable the **`Ice.Trace.ThreadPool`** property.
  - Setting this property to a non-zero value causes the Ice run time to log a message when it creates a thread pool, as well as each time the size of a thread pool increases or decreases.

Configuring Thread Pools

- A dynamic thread pool can grow and shrink when necessary in response to changes in an application’s work load.
  - All thread pools have at least one thread, but a dynamic thread pool can grow as the demand for threads increases, up to the pool’s maximum size.
  - Threads may also be terminated automatically when they have been idle for some time.
- The dynamic nature of a thread pool is determined by the configuration properties **`name.Size`**, **`name.SizeMax`**, and **`name.ThreadIdleTime`**.
  - A thread pool is not dynamic in its default configuration because **`name.Size`** and **`name.SizeMax`** are both set to one, meaning the pool can never grow to contain more than a single thread.
  - To configure a dynamic thread pool, you must set at least one of **`name.Size`** or **`name.SizeMax`** to a value greater than one.

Adapter Thread Pools

- The default behavior of an object adapter is to share the thread pools of its communicator and, for many applications, this behavior is entirely sufficient. However, the ability to configure an object adapter with its own thread pool is useful in certain situations:
  - When the concurrency requirements of an object adapter does not match those of its communicator.
    - In a server with multiple object adapters, the configuration of the communicator’s client and server thread pools may be a good match for some object adapters, but others may have different requirements.
      - For example, the servants hosted by one object adapter may not support concurrent access, in which case limiting that object adapter to a single-threaded pool eliminates the need for synchronization in those servants. On the other hand, another object adapter might need a multi-threaded pool for better performance.
  - To ensure that a minimum number of threads is available for dispatching requests to an adapter’s servants. This is especially important for eliminating the possibility of deadlocks when using nested invocations

Adapter Thread Pools

- An object adapter’s thread pool supports all of the properties described previously.
  - For configuration purposes, the name of an adapter’s thread pool is **`adapter.ThreadPool`**, where adapter is the name of the adapter.
- An adapter creates its own thread pool when at least one of the following properties has a value greater than zero:
  - **`adapter.ThreadPool.Size`**
  - **`adapter.ThreadPool.SizeMax`**
  - These properties have the same semantics as those described earlier except they both have a default value of zero, meaning that an adapter uses the communicator’s thread pools by default.
  - As an example, the properties shown below configure a thread pool for the object adapter named PrinterAdapter:
    - **`PrinterAdapter.ThreadPool.Size=3`**
    - **`PrinterAdapter.ThreadPool.SizeMax=15`**
    - **`PrinterAdapter.ThreadPool.SizeWarn=14`**

Single-Threaded Pool

- There are several implications of using a thread pool with a maximum size of one thread:
  - Only one message can be dispatched at a time.
    - This can be convenient because it lets you avoid (or postpone) dealing with thread-safety issues in your application.
      - However, it also eliminates the possibility of dispatching requests concurrently, which can be a bottleneck for applications running on multi-CPU systems or that perform blocking operations. Another option is to enable serialization in a multi-threaded pool.
  - Only one AMI reply can be processed at a time.
    - An application must increase the size of the client thread pool in order to process multiple AMI callbacks in parallel.
  - Nested twoway invocations are limited.
    - At most one level of nested twoway invocations is possible.
- It is important to remember that a communicator’s client and server thread pools have a default maximum size of one thread, therefore these limitations also apply to any object adapter that shares the communicator’s thread pools.

Multi-Threaded Pool

- Configuring a thread pool to support multiple threads implies that the application is prepared for the Ice run time to dispatch operation invocations or AMI call-backs concurrently.
  - Although greater effort is required to design a thread-safe application, you are rewarded with the ability to improve the application's scalability and throughput.

Multi-Threaded Pool

- Choosing an appropriate maximum size for a thread pool requires careful analysis of your application.
  - For example, in compute-bound applications it is best to limit the number of threads to the number of physical processors in the host machine; adding any more threads only increases context switches and reduces performance.
  - Increasing the size of the pool beyond the number of processors can improve responsiveness when threads can become blocked while waiting for the operating system to complete a task, such as a network or file operation.
  - On the other hand, a thread pool configured with too many threads can have the opposite effect and negatively impact performance.
  - Testing your application in a realistic environment is the recommended way of determining the optimum size for a thread pool.

Multi-Threaded Pool

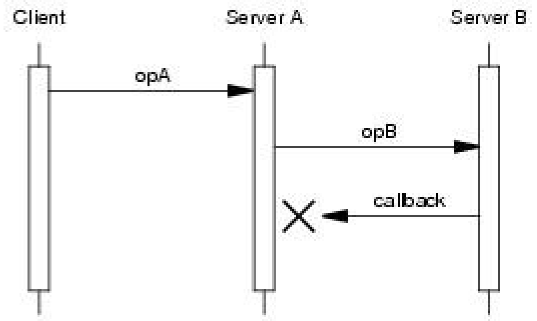
- If your application uses nested invocations, it is very important that you evaluate whether it is possible for thread starvation to cause a deadlock.
  - Increasing the size of a thread pool can lessen the chance of a deadlock, but other design solutions are usually preferred.

Nested Invocations

- A nested invocation is one that is made within the context of another Ice operation.
  - For instance, the implementation of an operation in a servant might need to make a nested invocation on some other object, or an AMI callback object might invoke an operation in the course of processing a reply to an asynchronous request.
  - It is also possible for one of these invocations to result in a nested call-back to the originating process.
  - The maximum depth of such invocations is determined by the size of the thread pools used by the communicating parties.

Deadlocks

- Applications that use nested invocations must be carefully designed to avoid the potential for deadlock, which can easily occur when invocations take a circular path.
- The implementation of opA makes a nested twoway invocation of opB, but the implementation of opB causes a deadlock when it tries to make a nested callback.
- The communicator’s thread pools have a maximum size of one thread unless explicitly configured otherwise.
- In Server A, the only thread in the server thread pool is busy waiting for its invocation of opB to complete, and therefore no threads remain to handle the callback from Server B.
- The client is now blocked because Server A is blocked, and they remain blocked indefinitely unless timeouts are used.



Deadlocks

- There are several ways to avoid a deadlock in this scenario:
  - Increase the maximum size of the server thread pool in Server A.
    - Configuring the server thread pool in Server A to support more than one thread allows the nested callback to proceed.
      - This is the simplest solution, but it requires that you know in advance how deeply nested the invocations may occur, or that you set the maximum size to a sufficiently large value that exhausting the pool becomes unlikely.
      - For example, setting the maximum size to two avoids a deadlock when a single client is involved, but a deadlock could easily occur again if multiple clients invoke opA simultaneously.
      - Furthermore, setting the maximum size too large can cause its own set of problems.

Deadlocks

- Use a oneway invocation.
  - If Server A called opB using a oneway invocation, it would no longer need to wait for a response and therefore opA could complete, making a thread available to handle the callback from Server B.
  - However, we have made a significant change in the semantics of opA because now there is no guarantee that opB has completed before opA returns, and it is still possible for the oneway invocation of opB to block.
- Create another object adapter for the callbacks.
  - No deadlock occurs if the callback from Server B is directed to a different object adapter that is configured with its own thread pool.
- Implement opA using asynchronous dispatch and invocation.
  - By declaring opA as an AMD operation and invoking opB using AMI, Server A can avoid blocking the thread pool’s thread while it waits for opB to complete.
  - This technique, known as asynchronous request chaining, is used extensively in Ice services such as IceGrid and Glacier2 to eliminate the possi-bility of deadlocks.