Connections

ZeroC's Newsletter for the Ice Community

Issue Number 24, June 2007

Diversity

In this issue, *Connections* presents another article written by one of ZeroC's customers. In a two-part series, Stephan Stapel from S2 Industries shows you how to integrate your Ice applications with relational database management systems. (In this issue, Stephan discusses the basics of object-relational mapping; in the next issue, he will discuss more advanced topics, such as threading, connection management, and scalability.)

I'm thrilled to see our customers contribute to *Connections*. For one, it means that we don't have to fill every issue ourselves (and, believe me, a lot of work goes into each article). But, more importantly, it means that real customers write about real applications they are creating with Ice. Stephan's articles draw on two such applications, a document management system and a project management system. Off hand, if someone asked me to list essential software components for these applications, Ice is not something that would immediately spring to my mind. Yet, as our customers integrate more and more computing tasks across diverse environments such as the Web, intranets, the mobile phone system, and special-purpose embedded devices, they keep coming up with new and innovative uses for Ice. Increasingly, Ice is used in situations we never thought of; currently, these include document retrieval, telediagnosis, real-time investment trading, video conferencing and online collaboration, online gaming, biometrics, real-time process control, remote sensing, internet telephony, command-and-control, and road traffic monitoring (among many others).

This diversity is not only interesting, but also provides inspiration for us to extend Ice in ways that, without the experience of our customers, we would never have thought of. Because each situation presents its own set of problems, there is always something worthwhile to learn about how customers have dealt with various trade-offs. And chances are that an approach used to solve a problem in one situation will transfer to a (sometimes surprisingly) different situation and therefore be interesting to a wider audience. So, I want to encourage you to write for *Connections* and share your experiences, not just for ZeroC's sake, but also for the sake of the ever-growing Ice community. Don't be shy about things in Ice you found didn't work so well: things that were difficult to do are more interesting to read about than things that were easy. And, as a customer-driven company, we need your criticism because it provides the impetus for ongoing improvements and new features (and, besides, our egos can take it).

As applications using Ice grow more diverse, so does Ice itself. Initially, Ice supported C++ and Java on mainstream operating systems and, from there, has grown to support seven languages (a "baker's half-dozen") and platforms such as Gumstix Linux, Windows Mobile, Java Micro Edition, plus a whole raft of general-purpose OSs. In turn, this expanded the diversity of hardware: Ice can now run on hardware as limited as a remote control, mobile phone, or PDA, and as powerful as a big-iron server cluster. All this diversity is driven by your feedback, so please keep it coming. And, if you've somehow managed to make Ice run on an abacus, we'd love to hear about it!

Michi

Michi Henning
Chief Scientist,

Issue Features

Beyond Freeze—Persistence with Ice
Part 1: The Basics

Stephen Stapel discusses how to integrate Ice applications with relational database management systems.

Connection Management in Ice

Matthew Newhook provides detailed look at how the Ice run time establishes and tears down connections, and how it chooses among connections if an Ice object can be reached via more than one endpoint.

Contents

Beyond Freeze—Persistence with IcePart 1: The Basics 2

Connection Management in Ice 9

FAQ Corner 15

Beyond Freeze—Persistence with Ice Part 1: The Basics

Stephan Stapel, Freelance Systems Architect
S2 Industries

Motivation

As an object-oriented middleware framework, Ice is designed to implement an object-oriented representation of both information and functionality for remote access. This works perfectly for functional business concerns that are implemented with Ice: servants are accessed remotely via proxies and the Ice run time. Almost the same is true for information: information structures can be designed with Slice and can easily be transferred using the Ice run time, but the question is how to store and retrieve the underlying data sets. Traditionally, business applications store data within relational database management systems (RDBMS). These systems are easy to set up, use, and maintain, are rock-solid, provide fast access, and—depending on the actual product that is used—can be very cost-effective.

Unfortunately, relational databases store data in a different style than objects. For example, an RDBMS core concept, namely foreign key relations, does not apply to objects. Similarly, an object-oriented core concept, namely inheritance, is not supported by a (classical) RDBMS. (There also are other differences, such as the presence or absence of null values.) These differences are known as the object-relational impedance mismatch; to overcome the mismatch, we require a mapping between the two styles of data representation, known as an object-relational mapping.

This article is based on our experience with two real-world applications that use both Ice and relational databases. The first application is a large-scale document management system, currently managing around 100,000 scanned documents. The documents are versioned and are described by metadata as well as user annotations. Document images are stored in the file system; however, the entire document information is managed using PostgreSQL. Ice is used to provide a consistent interface for data manipulation.

The second application is a planned multi-project management system that includes milestone tracking and management of project members and tasks. This application is also based on PostgreSQL and Ice.

I will use examples from the second application to introduce the concepts and to make recommendations on how to successfully integrate database and middleware technologies.

Scenario

Ice currently provides Freeze as its only built-in persistence mechanism. Freeze allows an application to easily load and store objects in a database without having to care about the underlying database representation. Freeze uses Oracle Berkeley DB as its database.

Freeze uses simple maps of objects that guarantee fast access to object data. By using indices, querying arbitrary fields is possible. However, fancy database actions such as resolving foreign key relationships are not possible with this approach. Instead, the application must use multiple maps and resolve the referenced objects explicitly. Complex queries on multiple attributes or sub-selections are not possible either. Another constraint with Freeze is that a database can be accessed by only one server at a time.

For more complex use cases, it is better to store the data in a relational database that supports the aforementioned complex queries, joins, and so on. Using an RDBMS is also necessary if an existing application is extended to use Ice, in which case Freeze simply cannot be used.

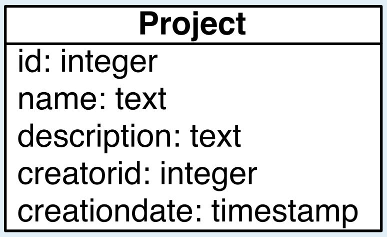
Among the important available relational database systems are Oracle, Microsoft SQL Server, MySQL, and PostgreSQL. I will use the latter for the examples throughout this article.

To avoid making things too complicated, I assume in the following examples that we start from the beginning and do not rely on pre-existing database schemas.

Object-Relational Mapping

To illustrate object-relational (OR) mapping, we begin with a simple example. Figure 1 shows a simple entity, called `Project`. This entity will be used in the project management system.

Figure 1: The `Project` Entity



The entity has a number of attributes, each providing a particular datum. The most important attribute is the `id` attribute; it uniquely identifies a project and is known as the primary key of the entity. Furthermore, a project has a distinct `name`. The `description` contains text of arbitrary length that provides further information about the project.

The `creatorid` attribute is an implicit reference to another element, namely the creator of the project. The creator also has an `id` attribute that serves as the creator's primary key. Such a reference is called a foreign key relationship because the `creatorid` is a reference to the key of an entity of different type. (We will use this attribute shortly.) The `creatorid` attribute is complemented by the `creationdate` attribute; this attribute is set whenever a new project is created and indicates the date and time of the creation of the element.

To store data of this kind into the database, we use the data definition language (DDL) that is part of the SQL language to create a database table with the aforementioned attributes:

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

We can now access the table, and insert, query, and delete data easily using SQL. The question is how to access this table via Ice.

Approaches for Representing Data within Ice Objects

Before discussing how to represent complex entities using Ice, we should first look at the building blocks of entities: fields and their attributes. Let's examine the most common SQL (SQL92/ SQL99) standard types and their possible Slice representation:

SQL Type	Slice type
binary	bool or byte
bit string	sequence<bool> or sequence<byte>
blob	sequence<byte>
boolean	bool
character	string
clob	string
nchar	string
nvchar	string
vchar	string
double	double
float	float
integer	int
real	float
smallint	int
Time	DataTypes::Time
Date	DataTypes::Date
Time with timezone	DataTypes::Time
timestamp	DataTypes::DateTime
timestamp with timezone	DataTypes::DateTime

Please note that the string mapping requires additional checks as string lengths in databases are usually limited, while Slice strings have no length limitation.

Standard SQL provides built-in types such as `boolean`, `integer`, `float`, `double` that map nicely to Slice and the corresponding implementation language types.

Some of the standard types have no native representation in either Slice or some of the implementation languages supported by Ice. For example, C++ does not have equivalents for the `time`, `date`, `datetime`, and the `bit string` types. The following Slice definition illustrates how these types can be represented in Slice:

```
// Slice
module DataTypes
{
    struct Date
    {
        int nYear;
        byte nMonth;
        byte nDay;
        byte nHour;
    };

    struct Time
    {
        byte nHour;
        byte nMinute;
        byte nSecond;
        int nMilisecond;
        int nUtcOffset;
    };

    struct DateTime
    {
        int nYear;
        byte nMonth;
        byte nDay;
        byte nHour;
        byte nMinute;
        byte nSecond;
        int nMilisecond;
        int nUtcOffset;
    };
};
```

Using these structures, both date and time variables can be used with Ice. Please note that other timestamp mappings would also be perfectly valid (such as using an `int` value to represent the number of seconds that have elapsed since 1st of January 1970).

Code that converts between the Slice representation and the corresponding implementation language type (if any) such as `System.DateTime` for C#, `datetime.datetime` for Python, or `QDateTime` for Qt can be placed into a helper library for easy access.

Representing Entities as Objects

When mapping database entities to Ice objects, there are basically two approaches that can be used: fine-grained interfaces or coarse-grained interfaces.

Fine-grained interfaces provide a get and a set function for each attribute to read and write the value. (If an attribute is read-only, the set function is absent.) For our project example, the interface would look like this:

```
// Slice
interface CProject
{
    idempotent string getName();
    void setName(string newname);
    idempotent string getDescription();
    void setDescription(string newdescription);
    // no setters for this field
    idempotent CUser* getCreatorId();
    // no setters for this field
    idempotent DateTime getCreationTime();
};
```

The advantage of this approach is that we have a fully object-oriented representation of the data. This style of data access is easy to read and also widely used to manipulate properties in languages such as Java and C#. Also, data is mostly kept on the server and only single attributes are exchanged in an atomic fashion. Thus, there are few concerns regarding synchronization of concurrent accesses.

Unfortunately, when modifying multiple attributes, this approach causes a huge number of tiny data transfers between client and server, each of which incurs some latency (not to mention the additional transactions that are necessary for each set function implementation).

Frequently, a key performance factor is not the amount of data that is exchanged but the latency, so this approach is unsuited for many common use cases.

To avoid these small and expensive data transfers, coarse-grained interfaces can be used. Coarse-grained interfaces provide access to all the data of an entity via a single read function and a single write function. The coarse-grained interface for the project therefore looks like this:

```
// Slice
struct CProjectDesc
{
    CProject* projectproxy;
    string szName;
    string szDescription;
    CUser* creatorproxy;
    DataTypes::DateTime creationTime;
};

interface CProject
{
    idempotent CProjectDesc describe();
    void saveUpdate(CProjectDesc desc);
};
```

The Slice defines a structure that contains all attributes that should be accessible to the user. In order to access one of these attributes, the user calls the `describe` function to retrieve the entire attribute set. Contrary to the fine-grained approach, a single remote invocation is used to receive or to update all of the object's data. In the style of the well-known POJO (Plain Old Java Objects) approach, a nice name for this approach is POSO (Plain Old Slice Objects).

I prefer implementing data access using the coarse-grained approach. While it does not have the pure object-oriented charm of the fine-grained approach, it makes up for this with numerous other advantages.

One drawback of the coarse-grained approach is that transmitting object data to the client can result in synchronization problems because different versions of the data can exist in different clients simultaneously. Thus, correct handling of concurrency issues is important.

An open issue with the coarse-grained approach is how to distinguish between read-only attributes and read-write attributes. In the preceding example, any descriptor's member variable can be modified by the user and passed to `saveUpdate`. It is up to the servant's implementation to decide which attribute to save and which member modification to ignore. One option to distinguish read-only from read-write attributes is to create sub-structures as follows:

```
// Slice
struct CProjectRODesc
{
  CProject* projectproxy;
  CUser* creatorproxy;
  DataTypes::DateTime creationTime;
};

struct CProjectRWDesc
{
    string szName;
    string szDescription;
};

struct CProjectDesc
{
    CProjectRODesc ro;
    CProjectRWDesc rw;
};

interface CProject
{
    idempotent CProjectDesc describe();
    void saveUpdate(CProjectRWDesc desc);
};
```

This splitting of attributes makes the definition clearer and also optimizes the `saveUpdate` function call because only the read-write attributes are transmitted over the wire.

Proxy Creation

Database row identification and proxy identification should be as consistent as possible to ease their mapping. The `Ice::Identity` structure allows us to easily create the association here: we can use the structure's two members `name` and `category` for the mapping:

```
// C++
std::stringstream ssStream;
ssStream << nId;

Ice::Identity id;
id.name = ssStream.str();
id.category = "project";
Ice::ObjectPrx prx = adapter->createProxy(id);
```

As you might have noticed in the descriptor definition above, there is no `id` field to map the object to the corresponding database row. Instead, the proxy is created using the database primary key `nId` as the object identity. The key can later be retrieved with `Current.id` on the server side or from the proxy with the `ice_getIdentity` proxy function on the client side; the key then can be used to access the corresponding database record.

Relations among Objects

So far I have discussed how to implement an isolated entity as an object. However, this is insufficient for applications that use interdependent objects, that is, objects that refer to each other. For our project management example, retrieving the creator and members of an arbitrary project requires use of such a relation. The UML model is shown in Figure 2.

The corresponding SQL table definitions look like this:

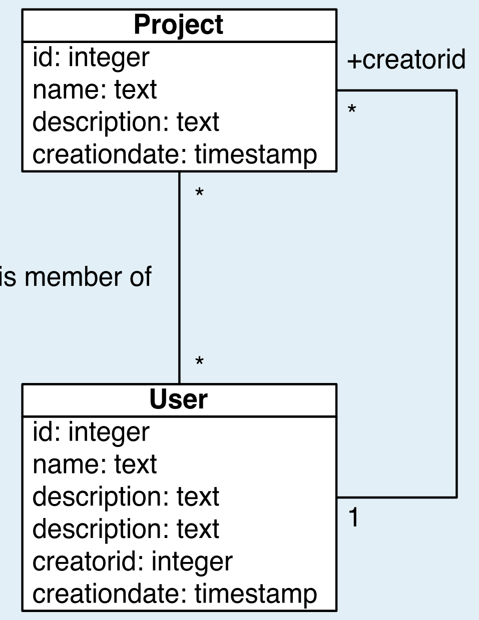
```
-- PostgreSQL
CREATE TABLE User (
    id integer,
    username text,
    creatorid integer,
    creationdate timestamp without time zone
);

CREATE TABLE Project (
    id integer,
    name text,
    description text,
    creatorid integer REFERENCES user,
    creationdate timestamp without time zone
);

CREATE TABLE UserProject (
    id integer,
    userid integer REFERENCES User,
    projectid integer REFERENCES Project
);
```

Here is the corresponding Slice interface:

```
// Slice
struct CUserDesc
{
    CUser* userproxy;
    string szUsername;
    CUser* creatorproxy;
    DataTypes::DateTime creationdate;
};

interface CUser
{
    idempotent CUserDesc describe();
};

sequence<CUser*> CUserPrxSequence;

interface CProject
{
    // returns the basic fields
    idempotent CProjectDesc describe();
    idempotent CUser* getCreator();
    idempotent CUserPrxSequence
    getProjectMembers();
};
```

When invoking the `getCreator` function, the creator proxy is created on demand by the `CProject` servant using the creator's primary key as the object identity. By installing a servant locator for the `CUser` servants, any call via a `CUser` proxy is automatically routed through this locator to the underlying servant implementation. A variant of this access pattern is to return the `CUserDesc`

Figure 2: The `Project` Entity and its Related `User` Entity



structure instead. In this case, the `CProject` servant implementation needs to know how to fill that structure in order to return it. In part 2 of this article, I will introduce a way to do this easily without the need for the `CProject` servant to know anything about the `CUserDesc` structure.

When modeling relations, we must decide whether relations are always retrieved from the server (as in the preceding example, using `getCreator` and `getProjectMembers`) or whether relations are also transferred to the client using the descriptor approach.

Depending on how the business logic is implemented, either approach can be valid: it might be a good choice to use a separate `addMember` operation if additional data needs to be passed to add members to a project, such as their project-specific role or the date of their entry into the project. Alternatively, members can be added to a project by simply adding a new item to a member list (the `memberProxies` attribute) of the project descriptor and invoking `saveUpdate`.

For the latter case, the definition looks like this:

```
// Slice
sequence<CUser*> CUserPrxSequence;

struct CProjectDesc
{
    CProject* projectProxy;
    CUserPrxSequence memberProxies;
    string szName;
    string szDescription;
    CUser* userProxy;
    DataTypes::DateTime creationTime;
};

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

The two approaches should not be mixed—I suggest to either use the `memberProxies` list in the descriptor or to define an `addMember` function. Mixing the two approaches is not only confusing to the caller, but also can raise concurrency issues. For example consider the following scenario:

1. User 1 retrieves the descriptor with `describe` to add a new member to the `memberproxies` list.
2. User 2 adds a member to the project using `addMember`.
3. User 1 tries to save the modified descriptor using `saveUpdate`. At this point the update fails because parts of the descriptor (namely the member list) have changed. (Allowing the update to succeed would violate the ACID guarantees and is therefore not an option.)

Inheritance

There are various strategies for modeling inheritance with entity relationships (ERs). Two important ones are:

- Table per class hierarchy. Data for the base class and all children is stored in the same table. Using a discriminator column, the corresponding type for a particular row can be retrieved, which in turn defines the valid columns for that row.
- Table per subclass. A base table exists for the base class. This table defines the primary key and type of each object. For each subclass, a separate table holds all subclass attributes.

I will use the table per subclass strategy to model a hierarchy of user types:

```
-- PostgreSQL
CREATE TABLE User (
    type integer,
    id integer,
    username text,
    creatorid integer,
    creationdate timestamp without time zone
);

CREATE TABLE Companyuser (
    userid integer references user,
    employeenumber string
);

CREATE TABLE Consultantuser (
    userid integer references user,
    consultingfirmname string
);
```

In this example, two additional user types are defined: one type for users that are working for a specific company (`Companyuser`) and one type for users that are working as external consultants for that company (`Consultantuser`).

When mapping ER-modeled class hierarchies to Slice, one has to decide between data-only inheritance and data-and-functionality inheritance. Data-only inheritance only affects the object descriptors in that it uses different attribute sets for each type. In contrast, data-and-functionality inheritance also affects the interfaces that are used: for each subclass, a separate interface defines the additional functionality for that subclass.

For data-only inheritance we will create a hierarchy of descriptors and therefore use a Slice `class` instead of a Slice `struct` for each class.

Because functionality does not differ among subclasses with this definition, we can still use a single interface to access both data and functionality of a user:

```
// Slice
class CUserDesc
{
    CUserTypeDiscriminator typeDiscriminator;
    CUser* userproxy;
    string szUsername;
    CUser* creatorproxy;
    DataTypes::DateTime creationdate;
};

class CCompanyUserDesc extends CUserDesc
{
    string szEmployeeNumber;
};

class CConsultantUserDesc extends CUserDesc
{
    string szConsultingFirmName;
};

interface CUser
{
    idempotent CUserDesc describe();
};
```

When retrieving such a user descriptor object from a `CUser` proxy, a type-safe down-cast can be used to safely cast the descriptor to the correct derived type.

If the functionality of derived classes differs significantly, we must create one interface for each subclass:

```
// Slice

// Descriptors defined as before...
interface CUser
{
    idempotent CUserDesc describe();
};

interface CCompanyUser extends CUser
{
    // Provide additional functionality here...
};

interface CConsultantUser extends CUser
{
    // Provide additional functionality here...
};
```

This second approach is slightly more complex. For example, we now use additional proxy types, and the server must implement the corresponding servant hierarchy.

Inheritance Limitations

A danger of using inheritance to model business concepts is its inflexibility. An alternative to inheritance is to use roles and composition. Instead of defining strict hierarchical subclasses for types, only a base class exists (`CUserDesc`, in this example). Specific types that were previously modeled as subclasses are instead modeled as roles that are attached to base class instances. This permits assigning more than one role to a person whereas, with subclasses, a person can have only one role. For the preceding example, we need one role to represent a company user and another role to represent a consultant user. For each of these roles, we can introduce a well-defined set of attributes. Roles are assigned to users as follows:

```
// Slice
enum CUserRole
{
    CompanyUserRole,
    CconsultantUserRole
};

dictionary<string,DataTypes::Variant>
    AttributeMap;

class CUserRole
{
    CUserRole userRole;
    AttributeMap attributes;
};

sequence<CUserRole> CUserRoleSequence;
class CUserDesc
{
    CUser* userproxy;
    string szUsername;
    CUser* creatorproxy;
    DataTypes::DateTime creationdate;
    CUserRoleSequence rgRoles;
};
```

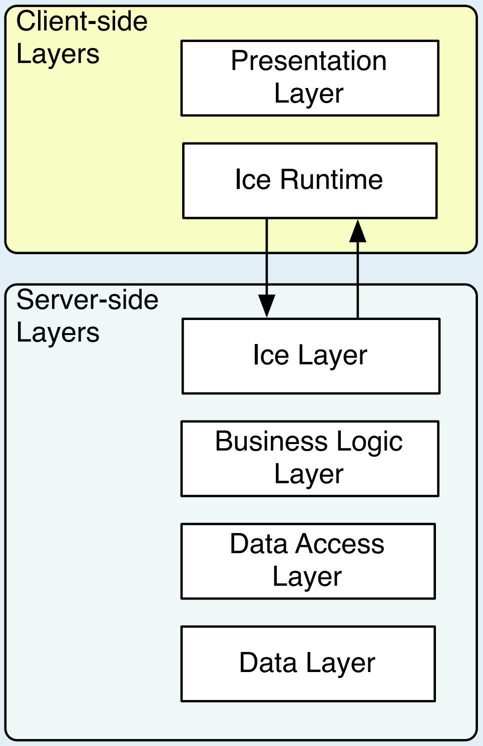
Implementation Best Practices

From my experience, creating explicit abstraction layers in applications minimizes interdependence between the various components of the application. By encapsulating the database access code into separate classes, data structures at higher layers are simplified as well. Figure 3 shows a common way to layer such abstractions.

The lowest layer in this case is the database. A data access layer physically connects to the database and moves data to and from the database. This shields the remainder of the application from the particulars of the database that is used. On top of the data access layer is a business object layer that contains all the business logic and uses the data access layer for persistence. Potentially, another layer on top of the business logic can be used to encapsulate Ice-related code, that is, the servant implementations; however, servant implementation and business logic are often combined into a single layer.

Despite this layering, Slice-defined types should be used for as many layers as possible. This avoids the need to convert data among different representations, which is error-prone and inefficient. (If you use different types to represent the same data in different layers, extensive unit testing should be used to ensure that conversion between different representations works correctly.)

Figure 3: Typical Application Layers



Ideally, the descriptor should be used to pass data between the persistence layer and the servant class. Further, the object data should be stored within the servant class using the same descriptor by adding a private member `m_desc` of type `CProjectDesc`.

A general recommendation is to encapsulate database transactions within single calls from the client's perspective to reduce the complexity of session management and transaction management. Transactions should not span across multiple client calls because this can cause deadlocks, for example, if a client disappears in the middle of a transaction. Avoiding operations that span transactions greatly simplifies application development and is usually possible by designing the application appropriately.

Additional logic may also be needed to support the chosen database. For example, some databases, such as PostgreSQL, limit access to a single thread for each connection. Unless care is taken, this can cause problems in multi-threaded environments such as Ice. (I will return to such concurrency issues in part 2 of this article.)

Another challenge is caching within the data access layer or the business logic layers of the application. While caching is a perfectly valid approach to increase the application's overall performance, it also increases the risk of losing data in case of crashes. Therefore, it is desirable to write modified data to the database immediately after receiving it from a client, that is, to restrict caching to operations that do not modify data. If the data is layered and managed by descriptor objects, read caches can be implemented easily, provided the application has exclusive access to the database. When this is not the case, it is usually easiest to simply disable read caching.

In the second part of this article, I will explore a number of advanced topics, including concurrency issues, connection management, and techniques for ensuring that the application scales well.

Connection Management in Ice

Matthew Newhook, Senior Software Engineer

Introduction

The Ice run transparently creates and closes connections on behalf of the application so, as an application developer, you can generally ignore how Ice manages connections. However, especially if servers provide multiple endpoints for Ice objects, it is useful to know how Ice deals with connections and chooses among them.

Client-side Connections

When clients contact a server via TCP or SSL, Ice needs to establish a connection between the two. Connections are always initiated by clients, and accepted by servers. Clients can obtain a `Connection` object from a proxy. This object describes the underlying connection for the proxy. (A connection object can be obtained even for datagram proxies, that is, proxies that contact the server via UDP.) The `Connection` object provides operations such as `close` and `createProxy`, as well as a number of other operations. (Please consult the Ice Manual for details.)

There are two methods that obtain the `Connection` object from a proxy:

- `ice_getConnection`. This proxy method returns the `Connection` object associated with the proxy. If no connection to the target exists yet, the Ice run time establishes a connection first and then returns the `Connection` object for the new connection. If the run time cannot establish a connection the operation raises an exception; if the Ice object to which the proxy refers is collocated, the method raises a `CollocationOptimizationException`.
- `ice_getCachedConnection`. This proxy method returns the `Connection` object associated with the proxy if a connection was previously established; if no connection is established, the method returns null.

Here is a simple example that illustrates how to obtain a `Connection` object:

```
// C++
CommunicatorPtr communicator = ...;
HelloPrx hello = HelloPrx::uncheckedCast(
    communicator->stringToProxy(
        "hello:tcp -h remote.host.com -p 10000"));
ConnectionPtr conn = hello->ice_getConnection();
```

The call to `ice_getConnection` establishes a connection to the `remote.host.com` at port 10000 and returns the associated `Connection` object. Contrast this with the following example:

```
// C++
CommunicatorPtr communicator = ...;
HelloPrx hello = HelloPrx::uncheckedCast(
    communicator->stringToProxy(
        "hello:tcp -h remote.host.com -p 10000"));
ConnectionPtr conn = hello->
    ice_getCachedConnection();
```

In this case, the call to `ice_getCachedConnection` returns null because no connection was established previously for the `hello` proxy.

As you might imagine, connections are not cheap. In particular, a connection consumes a file descriptor, memory, and—if you use the thread-per-connection concurrency model—a thread. Proxies use the communicator's default thread pool concurrency model or thread-per-connection if you set the property `Ice.ThreadPerConnection`. You can obtain a proxy that uses the thread-per-connection model (regardless of the default setting) by calling `ice_threadPerConnection(true)`. For example:

```
// C++
HelloPrx hello = ...;
HelloPrx htpc = HelloPrx::uncheckedCast(
    hello->ice_threadPerConnection(true));
```

Because connections are expensive, connection reuse is an integral part of the Ice run time. It is important to understand how the client side determines whether to establish a new connection or whether to re-use an existing connection.

Connection Life Cycle

The Ice run time maintains a pool of existing connections (on a per-communicator basis); the run time binds these connections to a proxy as a side-effect of the client making remote invocations. The run time creates new connections transparently as they are needed.

For example, consider the following code:

```
// C++
CommunicatorPtr communicator = ...;
HelloPrx h1 = HelloPrx::uncheckedCast(
    communicator->stringToProxy(
        "hello:tcp -h remote.host.com -p 10000"));
// Connection creation and binding occurs here
h1->sayHello();
```

When the client invokes `sayHello` via the proxy `h1`, the Ice run creates a connection to `remote.host.com` at port 10000 and binds this connection to the proxy. Note that the preceding example uses an `uncheckedCast` which does not make a remote invocation and, therefore, never establishes a connection. On the other hand, if the code were to use a `checkedCast` instead, then connection establishment would take place as part of the `checkedCast`, because a checked cast requires a remote call (ice_isA) to determine whether the target object supports the specified interface. (See my article “Proxies” in Issue 23 of *Connections* for more details.)

The life cycle of a connection is independent of the life cycle of a proxy. For example:

```
// C++
void
doit(const CommunicatorPtr& communicator)
{
    HelloPrx h1 = HelloPrx::uncheckedCast(
    communicator->stringToProxy(
        "hello:tcp -h remote.host.com -p 10000"));
    // Connection creation and binding occurs here
    h1->sayHello();
}
```

Once the doit function returns, the C++ run time destroys the proxy h1. However, the connection bound to that proxy object remains: the life cycle of a connection and the life cycle of the proxies that are bound to that connection are completely independent. This raises the question of how and when connections are closed and their associated resources released. The Ice run time closes and destroys connections in a variety of circumstances:

- Destroying a communicator closes and destroys that communicator’s connections.
- If active connection management (ACM) is enabled, it will close connections that have been idle for longer than a specified timeout.
- You can call close on a proxy’s Connection object to explicitly close a connection.
- If a connection has a timeout, the run time closes the connection when the timeout expires. (This is considered an unrecoverable exception.)
- If the run time encounters an unrecoverable error, such as a socket error, or receives data that violates the Ice protocol or encoding, it closes the corresponding connection.

The Ice Manual provides more detail on these scenarios.

A proxy may be bound to different connections during its life cycle. For example, a proxy may have a connection that remains idle for some time and is closed by ACM; the next time the proxy is used to make an invocation, the run time transparently establishes a new connection for the proxy. Similarly, a new connection may be established for a proxy because the previous connection was closed for any of the preceding reasons, or because connection caching is disabled. (I will return to this topic shortly.)

If you want to permanently bind a proxy to a specific connection, you can create a fixed connection proxy by calling Connection::createProxy. (The Ice Manual provides details on why you might want to do this.)

The Ice run time reuses existing connections when possible. For example, consider:

```
// C++
CommunicatorPtr communicator = ...;
HelloPrx h1 = HelloPrx::uncheckedCast(
    communicator->stringToProxy(
        "hello:tcp -h remote.host.com -p 10000"));
h1->sayHello();
HelloPrx h2 = HelloPrx::uncheckedCast(
    communicator->stringToProxy(
      "hello2:tcp -h remote.host.com -p 10000"));
h2->sayHello();
```

In this case, the Ice run time binds the two proxies h1 and h2 to the same connection because both proxies refer to an object at the same endpoint (remote.host.com at port 10000). In contrast, consider:

```
// C++
CommunicatorPtr communicator = ...;
HelloPrx h1 = HelloPrx::uncheckedCast(
    communicator->stringToProxy(
        "hello:tcp -h remote.host.com -p 10000"));
h1->sayHello();
HelloPrx h2 = HelloPrx::uncheckedCast(
    communicator->stringToProxy(
      "hello2:tcp -h remote2.host.com -p 8000"));
h2->sayHello();
```

In this example, the hello object resides on remote.host.com at port 10000, and the hello2 object resides on remote2.host.com at port 8000. Because the two proxies have different endpoints, the Ice run time establishes a separate connection for each proxy.

The situation becomes more complex if a proxy contains more than one endpoint. For example, consider:

```
// C++
CommunicatorPtr communicator = ...;
HelloPrx h1 = HelloPrx::uncheckedCast(
    communicator->stringToProxy(
        "hello:tcp -h remote.host.com -p 10000:tcp
-h remote2.host.com -p 8000"));
h1->sayHello();
HelloPrx h2 = HelloPrx::uncheckedCast(
    communicator->stringToProxy(
        "hello2:tcp -h remote.host.com -p 10000:tc
p -h remote2.host.com -p 8000"));
h2->sayHello();
```

In this case, both the hello and the hello2 objects can be reached on either remote.host.com at port 10000, or on remote2.host.com at port 8000. The question is whether the two proxies will share the same connection or may end up with separate connections. The answer is that the proxies share a single connection—to see why, we need to look in more depth at how Ice binds connections.

Endpoint Selection

During binding, the Ice run time looks at the endpoints of a proxy and, from that list of endpoints, produces an ordered list of candidate endpoints. The default algorithm for creating the list of candidate endpoints and binding a connection is as follows:

1. Remove any unusable or incompatible endpoints.
2. Shuffle the endpoints and, after shuffling, move secure endpoints to the end of the list. This establishes an endpoint preference order.
3. Check whether a compatible connection exists to any of the candidate endpoints. If so, reuse the connection.
4. Otherwise, no compatible connection exists. For each endpoint in the candidate list, attempt to establish a connection to that endpoint and use the connection if successful; otherwise, try the next endpoint on the candidate list until either a connection can be established, or no more candidate endpoints remain.

Depending on proxy settings, this algorithm may be modified—what follows are the nitty-gritty details of how endpoints are selected and how connections are established.

Removing Unusable and Incompatible Endpoints

The first step is to remove any endpoints that satisfy one of the following criteria:

- The endpoint is unknown. For example, if the IceSSL plug-in is not installed, an SSL endpoint is an unknown endpoint.
- The endpoint is incompatible. An endpoint is incompatible if it does not match the proxy's invocation mode. For example, for a datagram proxy, all endpoints that do not use UDP are removed.
- The endpoint is insecure, but the proxy is secure or `Ice.Override.Secure` is set.

If no endpoints remain once the run time has removed unusable and incompatible endpoints, the invocation raises a `NoEndpointException` because the proxy has no endpoints that could be used to make the invocation. Consider the examples that follow for illustration. (Note that the examples do not specify the `-h` option in the endpoints for brevity; doing so sets the host to `127.0.0.1` or the value of `Ice.Default.Host` if that property is set).

```
// C++
// Server: IceSSL plug-in installed.
ObjectPrx
SomeServantI::getObj(const Current& current)
{
    return current.adapter->getCommunicator()->
     stringToProxy(
      "obj:tcp -p 8000:udp -p 9000:ssl -p 10000");
}

// Client: IceSSL plug-in not installed.
ObjectPrx obj = someServant->getObj();
obj->ice_ping();
```

In this example, a client without the IceSSL plug-in receives a proxy containing TCP, SSL, and UDP endpoints over the wire. The Ice run time preserves the SSL endpoint in the client's proxy even though the client cannot use the endpoint. This allows the client to later send the proxy over the wire without losing the SSL endpoint. Also note that the client cannot directly create a proxy with an SSL endpoint by calling `stringToProxy` because, without the IceSSL plug-in, `stringToProxy` raises an `EndpointParseException` for SSL endpoints.

After the client-side run time has removed the unsuitable SSL and UDP endpoints, only the TCP endpoint remains and will be used. The SSL endpoint is removed because IceSSL is not installed in the client, and the UDP endpoint is removed because it can only be used to make datagram invocations.

```
// C++
// IceSSL plug-in installed.
ObjectPrx obj = communicator->stringToProxy(
    "obj:tcp -p 8000:udp -p 9000:ssl -p 10000");
obj->ice_ping();
```

With this example, the TCP and the SSL endpoint remain. If `Ice.Override.Secure` would be set, the TCP endpoint would be removed as well.

```
// C++
// IceSSL plug-in installed.
ObjectPrx obj = communicator->stringToProxy(
    "obj:tcp -p 8000:udp -p 9000:ssl -p 10000");
obj = obj->ice_datagram();
obj->ice_ping();
```

In this case, the only remaining endpoint is the UDP endpoint. If `Ice.Override.Secure` would be set, the UDP endpoint would be removed as well (because UDP cannot be used for secure invocations) and the call to `ice_ping` would raise a `NoEndpointException`.

```
// C++
// IceSSL plug-in installed.
ObjectPrx obj = communicator->stringToProxy(
    "obj:tcp -p 8000:udp -p 9000:ssl -p 10000");
obj = obj->ice_secure();
obj->ice_ping();
```

In this example, because the proxy is secure, only the SSL endpoint remains for connection establishment.

Endpoint Order

Once the Ice run time has removed unsuitable endpoints, it establishes the order in which endpoints will be used for connection attempts. Doing so proceeds in two steps:

1. The run time sorts the endpoint list based on the endpoint selection policy (which can be set with the `ice_endpointSelection` proxy method). By default, the endpoint selection policy is `Random`, meaning that the run time shuffles the endpoints into random order. Otherwise, the selection policy is `Ordered` and Ice preserves the order in which the endpoints are listed in the proxy.
2. If `PreferSecure` is false (the default value), the run time moves all secure endpoints to the end of the list. Conversely, if `PreferSecure` is true, the run times moves all secure endpoints to the beginning of the list. (You can set `PreferSecure` with the `ice_preferSecure` proxy method).

Consider the following examples:

```
// C++
// IceSSL plug-in is installed.
ObjectPrx obj = communicator->stringToProxy(
    "obj:tcp -p 8000:ssl -p 10000:tcp -p 9000");
obj->ice_ping();
```

In this case, the endpoint list is either <tcp -p 8000, tcp -p 9000, ssl -p 10000> or <tcp -p 9000, tcp -p 8000, ssl -p 10000>. The order of the TCP endpoints is random because the endpoint selection policy has the default value. However, the SSL endpoint is guaranteed to be at the end because `PreferSecure` is false.

```
// C++
// IceSSL plug-in is installed.
ObjectPrx obj = communicator->stringToProxy(
    "obj:tcp -p 8000:ssl -p 10000:tcp -p 9000");
obj = obj->ice_endpointSelection(Ordered);
obj->ice_ping();
```

In this case, the endpoint list is <tcp -p 8000, tcp -p 9000, ssl -p 10000>. This is because the selection policy is `Ordered`, so the two TCP endpoints retain the original order. The SSL endpoint appears at the end because `PreferSecure` is false.

```
// C++
// IceSSL plug-in is installed.
ObjectPrx obj = communicator->stringToProxy(
    "obj:tcp -p 8000:ssl -p 10000:tcp -p 9000");
obj = obj->ice_endpointSelection(Ordered);
obj = obj->ice_preferSecure(true);
obj->ice_ping();
```

In this case, the endpoint list is <ssl -p 10000, tcp -p 8000, tcp -p 9000>. Again, the endpoint selection policy is `Ordered`, so the two TCP endpoints retain the original order. However, because `PreferSecure` is true, the SSL endpoint appears first.

Connection Creation and Binding

If connection caching is enabled (which it is by default), the run time first checks all of the endpoints to see whether a connection to one of the endpoints already exists. If so, it reuses that connection; otherwise, it establishes a new one, that is, the run time establishes a new connection only if no compatible connection to any of the endpoints exists. If connection caching is disabled, the run time goes through the endpoint list in order and, for each endpoint, determines whether a compatible existing connection to that endpoint exists, in which case that connection is bound; otherwise, it attempts to establish a new connection to that endpoint. This means that, if connection caching is disabled, the run time may establish a new connection even if there is a compatible connection later in the endpoint list.

A connection can be reused if the connection's endpoint matches the proxy's endpoint and the connection matches the proxy's configuration. Specifically, the timeout setting of the connection must match the configured timeout for the proxy. (If the proxy has a configured connection ID, the connection ID must match—see the Ice Manual for details on connection IDs.)

Connection timeouts are a very important and often misunderstood aspect of Ice. In short, each connection has an associated timeout value. The timeout value is copied from the proxy that originally caused the connection to be established. If a request sent over that connection times out, all outstanding requests on that connection also time out and Ice forcefully closes the connection. Therefore, two proxies with different timeout values cannot share a connection. For example:

```
// C++
CommunicatorPtr communicator = ...;
ObjectPrx o = communicator->stringToProxy(
    "hello:tcp -h remote.host.com -p 10000");
HelloPrx h1 = HelloPrx::uncheckedCast(
    o->ice_timeout(1000));
h1->sayHello();
HelloPrx h2 = HelloPrx::uncheckedCast(
    o->ice_timeout(2000));
h2->sayHello();
```

In this case, `h1` and `h2` are bound to different connections because the timeout of the two proxies differs. Let me return to an earlier example again:

```
// C++
CommunicatorPtr communicator = ...;
HelloPrx h1 = HelloPrx::uncheckedCast(
    communicator->stringToProxy(
        "hello:tcp -h remote.host.com -p 10000:tcp
-h remote2.host.com -p 8000"));
h1->sayHello();
HelloPrx h2 = HelloPrx::uncheckedCast(
    communicator->stringToProxy(
        "hello2:tcp -h remote.host.com -p 10000:tc
p -h remote2.host.com -p 8000"));
h2->sayHello();
```

Consider the first invocation via `h1`. The endpoint list will be either <tcp -p 10000, tcp -p 8000> or <tcp -p 10000, tcp -p 8000> depending on how the endpoints are shuffled. (The endpoints are shuffled because the endpoint selection policy has the default value of `Random`.) Assuming a server runs at each endpoint, the client creates a connection to whatever endpoint happens to be first in the candidate list and binds that connection to the `h1` proxy. Now consider the second invocation via `h2`. In this case, as before, there are two possible endpoint lists. However, because connection caching is enabled, the Ice run time prefers to reuse the existing connection, and thus binds to whatever connection was established by the initial invocation via `h1`.

Connection Establishment Retries

If an attempt to establish a connection fails, the run time retries based on the value of `Ice.RetryIntervals`. The default value of this property is zero, which instructs the run time to retry connection establishment once for each endpoint. If no connection can be established via any of the endpoints, the run time raises an exception that indicates the reason for the final failed connection attempt.

Connection Caching

As previously stated, by default, a connection is bound to a proxy during the first remote invocation via that proxy; thereafter, the proxy continues to use this connection for as long as it remains open. In other words, the proxy caches the connection. If the connection is closed at some point, the next remote invocation via the proxy establishes a new connection using the algorithm I outlined earlier. For the majority of applications, this is the correct behavior because it minimizes the overhead of remote invocations.

For some applications, however, it is desirable to rebind a proxy's connection on each remote invocation. In particular, the default algorithm is unsuitable for per-request load balancing. In this scenario, the proxy contains an endpoint for each replica in a replica group. However, the default algorithm does exactly the wrong thing because, once a connection to any one of the replicas is established, all future requests are sent via that same connection, so only one replica is ever used:

```
// C++
CommunicatorPtr communicator = ...;
HelloPrx h1 = HelloPrx::uncheckedCast(
    communicator->stringToProxy(
        "hello:tcp -h remote.host.com -p 10000:tcp
-h remote2.host.com -p 8000"));
h1->sayHello();
h1->sayHello();
```

In this case, the second `sayHello` invocation is sent via whatever connection was established by the first invocation. To change this behavior, you must create a new proxy by calling `ice_connectionCached(false)`:

```
// C++
CommunicatorPtr communicator = ...;
ObjectPrx o = communicator->stringToProxy(
    "hello:tcp -h remote.host.com -p 10000:tcp -h
remote2.host.com -p 8000");
HelloPrx h1 = HelloPrx::uncheckedCast(
    o->ice_connectionCached(false));
h1->sayHello();
h1->sayHello();
```

Because connection caching is now disabled for `h1`, the second call to `sayHello` causes the binding algorithm to run a second time. There are two possible outcomes.

- During the first invocation, the endpoints are shuffled and the run time establishes a connection to one of the endpoints, for example, tcp -p 10000.
- During the second invocation, the selection algorithm runs a second time. If the endpoint shuffle results in the same order as for the first invocation, the request is sent over the already-existing connection. However, if the shuffle results in the opposite order, the second invocation causes a second connection to be opened, to tcp -p 8000.

Eventually, after a number of invocations, both connections will be established; selecting one of the two existing connections for every invocation made by the client is very efficient and results in per-request random load balancing.

Now assume we disable connection caching and set the selection policy to `Ordered`. Assuming a server actually runs at the first endpoint, all invocations made by the client will be bound separately, and the first endpoint will be tried first on each invocation. This behavior is useful for servers in a master–slave relationship: the master endpoint is listed first and will always be used unless the master is down, at which point the slaves identified by subsequent endpoints are tried. However, note that at present this is quite expensive because, while the master is down, the run time attempts to create a new connection to the first endpoint on every invocation, only to have every such attempt fail until the master comes back on line.

Active Connection Management

Active connection management (ACM) improves application scalability by closing idle connections. At regular intervals, the Ice run time checks each existing connection and, if a connection has been idle for more than `Ice.ACM.Client` (or `Ice.ACM.Server`) seconds, it gracefully closes the connection. (The default values of these properties are 60 seconds for the client and zero (i.e., disabled) for the server.) The next invocation made by a client via a proxy whose connection was closed causes the connection to be re-established, so ACM is transparent to application code.

Note that, on the server side, ACM is disabled by default because server-side ACM can cause oneway invocations to be silently discarded. (See this FAQ for more information.) Disabling ACM on the client side is necessary only if the client uses bi-directional connections. To disable ACM, set the corresponding property to zero.

In the context of ACM, “idle” means that, for the timeout period, no request has been sent over the connection, no invocations are in progress whose requests were sent over the connection, and no batch messages were added to a batch to be sent over the connection. For example:

```
// C++
CommunicatorPtr communicator = ...;
HelloPrx h1 = HelloPrx::uncheckedCast(
    communicator->stringToProxy(
        "hello:tcp -h remote.host.com -p 10000"));
h1->sayHello();
sleep(70);
h1->sayHello();
```

In this case, the connection that is established by the first call to `sayHello` is closed after 60 seconds (the default idle timeout for the client side). The second call creates a new connection to the same endpoint and binds that connection to the proxy. However:

```
// C++
CommunicatorPtr communicator = ...;
HelloPrx h1 = HelloPrx::uncheckedCast(
    communicator->stringToProxy(
        "hello:tcp -h remote.host.com -p 10000"));
h1->sayHello(); // Takes 70 seconds
h1->sayHello();
```

In this case, the connection is not closed because a reply is outstanding on the connection during the 70 seconds it takes the first call to complete.

Note that disabling ACM on the client side does not guarantee that the connection will not be closed because ACM may be active on the server side. If you want to be sure that connections remain open, you must disable ACM for both client and server.

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: I use Ice with Visual C++ 6.0/7.1 and also want to use Ice for Java/C#/VB/Python. What should I do?

The binary installers for Visual C++ 6.0 and Visual C++ 7.1 do not include the Slice compilers for Java, C#, Visual Basic, and Python. (These language mappings are only included in the installer for Visual C++ 8.0, aka Visual C++ 2005.)

If you want to use Ice with both Visual C++ 6.0/7.1 and one or more of the other languages, you need to download the installers for Visual C++ 6.0 or 7.1, plus the installer for Visual C++ 8.0. First run the installer for Visual C++ 6.0 or 7.1. The default installation directories for these installers carry a Visual C++ version number, such as `C:\Ice-3.2.0-VC60` and `C:\Ice-3.2.0-VC71`, respectively. Then run the installer for Visual C++ 8.0; the default installation directory only carries the Ice version number, but no version for Visual C++, for example, `C:\Ice-3.2.0`. When you run the Visual C++ 8.0 installer, choose custom setup. You can select the additional languages you want to use with Ice. For example, if you are only interested in development support for Java and C#, you can deselect everything else.

Having done this, you will end up with two Ice installations, one for Ice for Visual C++ 6.0/7.1, and a second one that provides the Slice compilers and run-time support for the additional languages you select. For example, if you select Java development, you will find the `slice2java` compiler in `C:\Ice-3.2.0\bin`, and JAR files in `C:\Ice-3.2.0\lib`.

To compile Ice for Java applications, you need the `slice2java` compiler provided by the Visual C++ 8.0 installation. This means that you must set your `PATH` correctly when you run `slice2java`: your `PATH` must point at `C:\Ice-3.2.0\bin`. Similarly, your Java `CLASSPATH` must include `C:\Ice-3.2.0\ant` if you want to compile the demo applications and, at run time, the `CLASSPATH` of your Ice for Java applications must include `C:\Ice-3.2.0\lib\Ice.jar` (and `C:\Ice-3.2.0\lib\db.jar` if your applications use Freeze).

Similar considerations apply for other language mappings: the Slice compilers are provided by the Visual C++ 8.0 installation so, when using the Slice compilers (`slice2cs`, `slice2vb`, or `slice2py`), you must ensure that your `PATH` includes `C:\Ice-3.2.0\bin` instead of `C:\Ice-3.2.0-VC60\bin` or `C:\Ice-3.2.0-VC71\bin`.

To run C# and Visual Basic applications, no further configuration is necessary; the Ice assemblies are installed in the global assembly cache (GAC), so there is no need to set environment variables to point at an installation directory.

To run Python applications, you need to direct the Python interpreter at the Ice extension for Python, that is, the application's `PYTHONPATH` must include `C:\Ice-3.2.0\python`.

Q: How does multi-threading work with Ice for Python?

Python supports multi-threaded programming, but the interpreter is inherently single threaded: a global interpreter lock (GIL) allows only one thread at a time to execute a Python opcode. Despite this limitation, careful management of the GIL can still provide performance improvements; for example, a thread that is about to block on I/O can release the GIL so a different thread can use the CPU in the meantime. Although the GIL is an implementation detail that is typically of interest only to developers of Python extensions, it is still important for Python programmers to understand the semantics of the GIL and how it affects their applications.

The Ice extension for Python is built as a run-time layer on top of Ice for C++, which is threaded. Depending on the concurrency model in use, the Ice for C++ run time may have several native threads running concurrently in the background, performing tasks such as accepting a new connection or reading an incoming request. The Python interpreter is unaware of these threads; furthermore, they are not constrained by the GIL. It is only when an Ice thread needs to call into the Python API, for example to dispatch an operation to a Python servant, that it must acquire the GIL.

The GIL also affects calls from Python code into the Ice API, such as when a program invokes a remote operation via a proxy. In this scenario, the GIL is already locked by the interpreter and the Ice extension releases the GIL to give another thread a chance to execute while the remote operation is in progress. If the GIL were to remain locked for the duration of the remote operation, not only would we unnecessarily restrict concurrency, but we would also introduce the possibility of a deadlock. For example, the operation might invoke a callback to a servant in the client process; if the GIL were to remain locked during the initial operation, the Ice extension would be unable to acquire the GIL before dispatching the callback. For these reasons, the Ice extension releases the GIL during any Ice API method that can block the calling thread, and acquires the GIL again just prior to returning control to the interpreter.

In practice, the limitations of the GIL mean that only one thread executes in the servant implementation code of an Ice server that is written in Python. Note that the single-threaded nature of the Python interpreter does not guarantee that all servant operations will execute atomically. In particular, if a Python server has more than one thread in its thread pool, it is possible for one thread to enter an operation on a servant, get suspended, and then for another thread to execute part or all of an operation on the same or a different servant. Proper use of synchronization is required in this case to protect shared resources. If you want true single-threaded execution, such that every operation runs to completion before a thread enters another operation, you must leave the `Ice.ThreadPool.Server.Size` property at its default setting of 1.

If you find that you cannot tolerate the performance limitations of serialized execution in your Ice server, currently the only option is to implement the server in a language other than Python, at least until Python provides true multi-threading support. The GIL will not be removed for Python 3.0 however, so this is not likely to happen any time soon.