Ice Programming with Java

1. Introduction to Ice

ZeroC

Lesson Overview

- This lesson covers:
  - the motivation for using Ice
  - the fundamentals of the Ice architecture
  - the Ice object model
- This lesson also provides an overview of the major Ice components (including some components not covered in this course).
- By the end of this lesson, you will have a basic understanding of the Ice architecture and how Ice helps you to develop distributed applications.

What is Ice?

- An object-oriented distributed middleware platform.

- Ice includes:

  - object-oriented RPC mechanism

  - language-neutral specification language (Slice)

  - language mappings for various languages: C++, Java, C#, Python, Objective-C, Ruby and PHP (Ruby and PHP for the client-side only)

  - support for different transports (TCP, SSL, UDP) with highly-efficient protocol

  - external services (server activation, firewall traversal, etc.)

  - integrated persistence (Freeze)

  - threading support

Clients and Servers

A client–server system is any software system in which different parts of the system cooperate on an overall task.

- A server is an entity that, on request, provides a service (such as a computation) to clients. Servers are passive.
- A client is an entity that requests services from servers. Clients are active.
- Client and server often run on separate machines, but might also run on the same machine or be linked into a single process.

Frequently, clients and servers are not "pure" clients and servers.

- A server might act as a client, and a client might act a server.
- Client and server are therefore roles that have a well-defined meaning only for the duration of a single request. The initiating side is, by definition, the client; the responding side is, by definition, the server.

Ice Objects

- An Ice object is a conceptual entity, that is, an abstraction.

- An Ice object:

  - can exist in the local or a remote address space

  - responds to operation invocations

  - can have multiple redundant instantiations

  - has one or more interfaces (facets), and has a single most-derived default interface (the default facet)

  - provides operations that can accept in-parameters, and can return out-parameters and/or a return value

  - has a unique object identity

Proxies

- Clients contact Ice objects via proxies.

- A proxy is a handle that uniquely denotes an Ice object.

- A proxy is the local ambassador for a (possibly remote) Ice object.

- When a client invokes an operation on a proxy, the Ice run time:

  1. Locates the Ice object

  2. Activates the object's implementation within the server

  3. Transmits in-parameters to the object

  4. Waits for the operation to complete

  5. Returns any out-parameters and the return value to the client (or an exception in case of an error)

Stringified Proxies

- Proxies can be converted to and from strings.

  `SimplePrinter:default -h host.xyz.com -p 10000`

- This is a proxy for an object with identity `SimplePrinter`.

- The object's server runs on `host.xyz.com` and listens on port 10000 for incoming requests.

- The server can be contacted using the configured default protocol. (If no default protocol is configured, the protocol defaults to TCP).

- Because such a proxy directly contains the endpoint at which the server can be found, it is known as a *direct* proxy. The general form of stringified direct proxies is:

  `<identity>:<endpoint>[:<endpoint>...]`

- Endpoints have the general form:

  `<protocol> [-h <host>] [-p <port>] [-t timeout] [-z]`

Servants

A servant is a server-side programming-language artifact that provides the concrete representation of an abstract Ice object.

Servants are said to *incarnate* Ice objects.

- Typically, servants are object instances with methods that correspond to the operations supported by an Ice object.
- Servants are written by you, the developer.
- When a client invokes an operation, the Ice run time takes care of invoking the corresponding method on the servant.
- The method bodies on a servant provide the behavior of the corresponding Ice object.
- A single servant can incarnate a single Ice object, or simultaneously incarnate several Ice objects.
- A single Ice object can have multiple servants (typically in different servers, for redundancy).

At-Most-Once Semantics

The Ice run time guarantees at-most-once semantics.

A single operation invocation by a client is guaranteed to:

- either invoke the operation exactly once
- or invoke the operation not at all

It is impossible for a single invocation of a client to result in the operation being invoked more than once.

At-most-once semantics are important if an operation is not idempotent.

You can mark individual operations as idempotent to relax the strict at-most-once semantics.

ZeroC

Method Invocation and Dispatch

Ice supports:

- Oneway and twoway synchronous method invocation
- Oneway and twoway asynchronous method invocation (AMI)
- Batched oneway invocation
- Datagram invocation
- Batched datagram invocation
- Synchronous method dispatch
- Asynchronous method dispatch (AMD)

Client and Server Structure
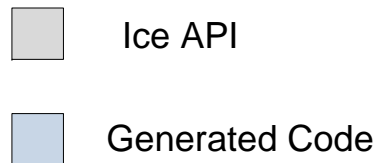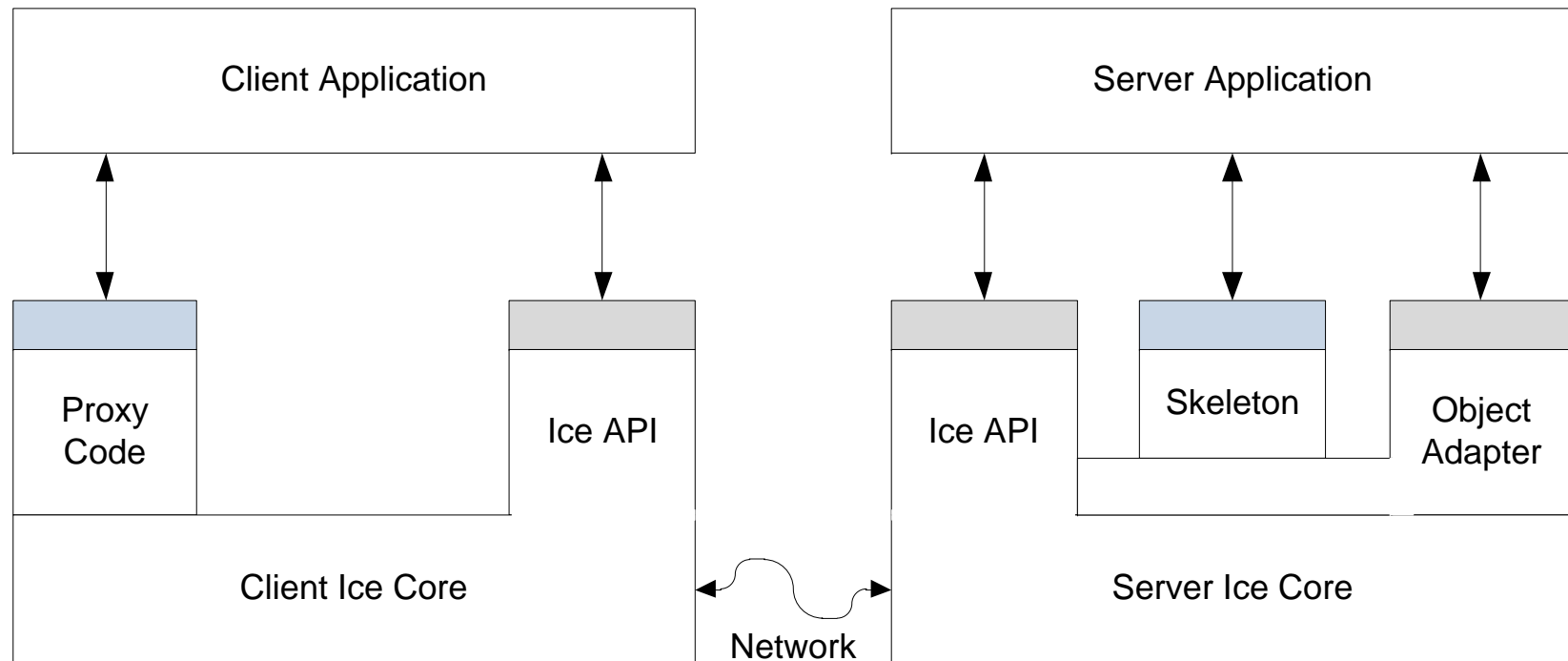
Ice Services

Ice provides a number of services:

- Persistence service (Freeze)

- Replication, load balancing, server activation service (IceGrid)

- Application server (IceBox)

- Publish–subscribe service (IceStorm)

- Software distribution and patching service (IcePatch2)

- Firewall traversal and session management (Glacier2)

Freeze is a library; the other services are implemented as stand-alone processes.

ZeroC

Ice Programming with Java

## 2. The Slice Interface Definition Language

Lesson Overview

- This lesson presents:

  - the syntax and semantics of the Slice interface definition language

- Slice is an acronym for Specification Language for Ice, but is pronounced as a single syllable, to rhyme with Ice.

- By the end of this lesson, you will be able to write interface definitions in Slice and to compile these definitions into Java stubs and skeletons.

What is Slice?

- Slice separates language-independent types from language-specific implementation.

- A compiler creates language-specific source code from Slice definitions.

- Slice is a declarative language that defines types. You cannot write executable statements in Slice.

- Slice establishes the client-server contract: data can be exchanged only if it is defined in Slice, via operations that are defined in Slice.

- Slice definitions are analogous to C++ header files: they ensure that client and server agree about the interfaces and data types they use to exchange data.

Single-Language Development
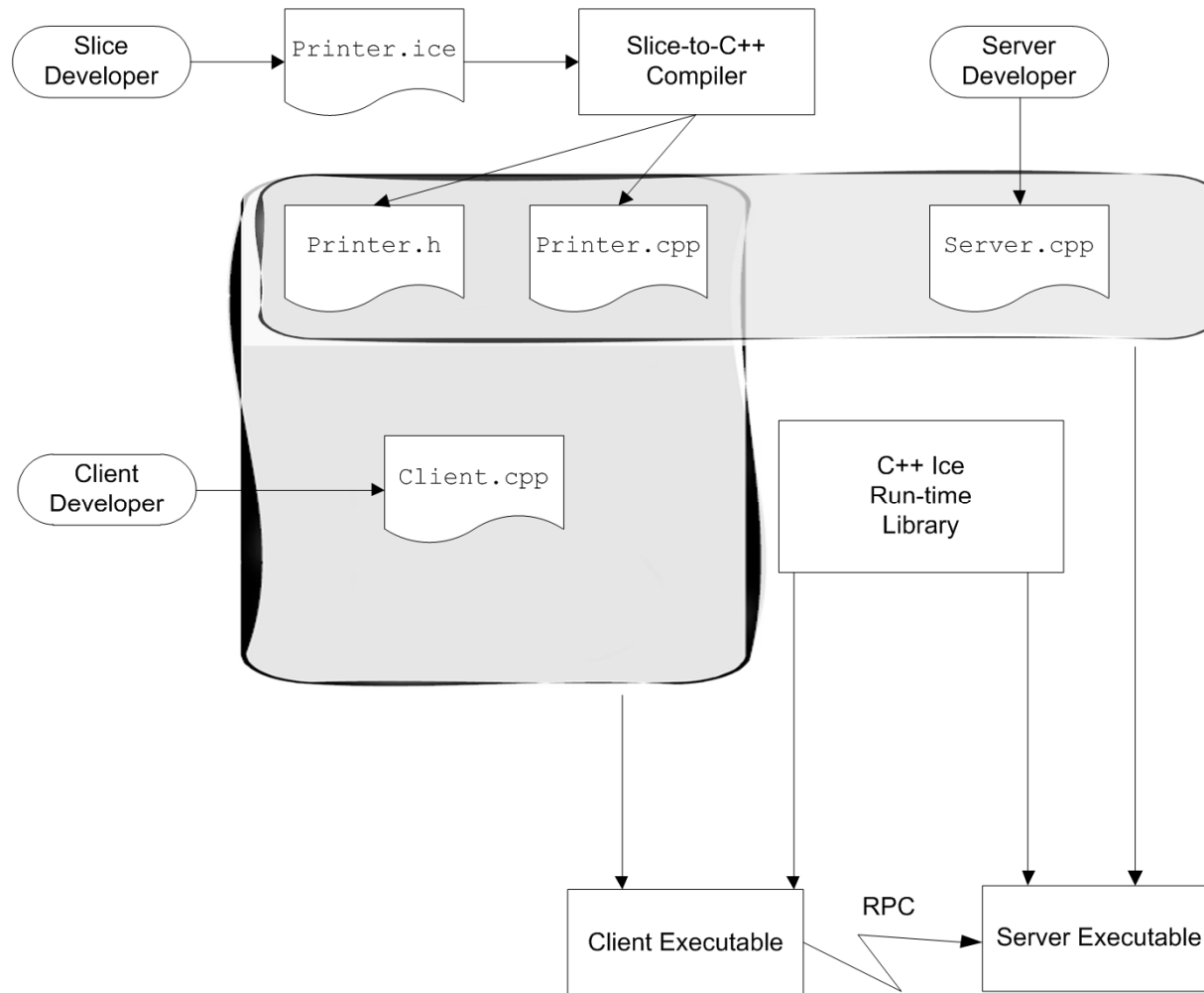
Cross-Language Development
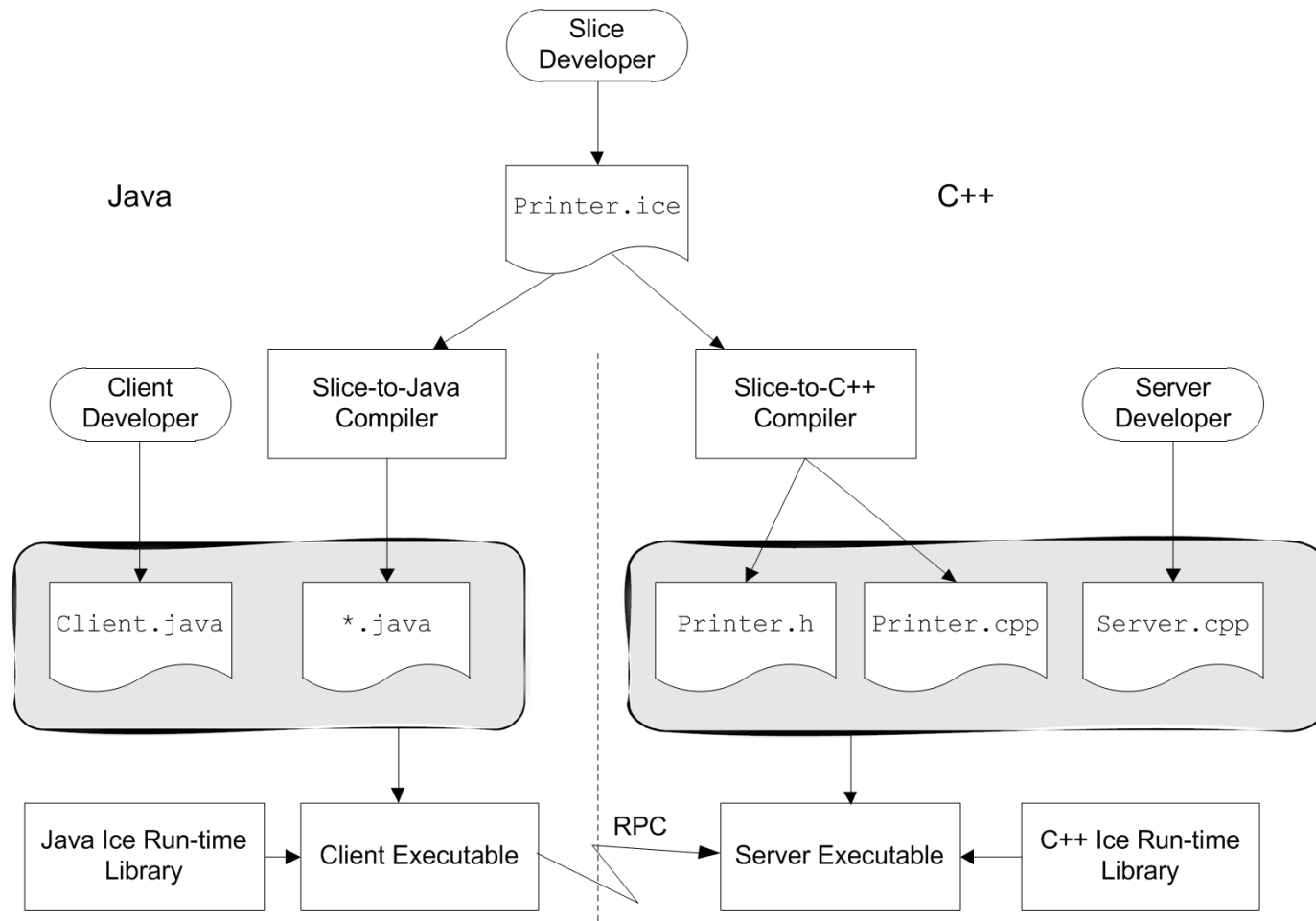
Slice Source Files

- Slice source files must end in a `.ice` extension.

- Slice source files are preprocessed by the C++ preprocessor, so you can use `#include`, `#define`, etc.

- If you `#include` another file, the compiler parses everything, but generates code only for the including file—the included file must be compiled separately.

- Slice is a free-form language, so indentation and white space are not lexically significant (other than as token separators).

- Definitions can appear in any order, but things must be defined before they are used (or forward declared).

Comments and Keywords

- Slice supports both C- and C++-style comments:

```
/*
 * This is a comment.
 */

// This comment extends to the end of this line.
```

- Slice keywords are written in lowercase (e.g. `class`), except for the keywords `Object` and `LocalObject`, which must be capitalized as shown.

ZeroC

Identifiers

- Identifiers consist of alphabetic characters, digits, and optionally underscores.
- Identifiers must start with an alphabetic character.
- Identifiers are case insensitive: `Foo` and `foo` cannot both be defined in the same naming scope.
- Identifiers must be capitalized consistently: once you have defined `Foo`, you must refer to it as `Foo` (not `foo` or `F00`).
- Slice identifiers cannot begin with `Ice`.
- You *can* define identifiers that are the same as a keyword, by escaping them:

```
\dictionary // Identifier, not keyword
```

  This mechanism exists as an escape hatch in case new keywords are added to the language over time.
- Avoid creating identifiers that are likely to be programming-language keywords, such as `function` or `new`.

ZeroC

Modules

A Slice file contains one or more module definitions.

```
module Example {
    // Definitions here...
};
```

The only definition that can appear at global scope (other than comments and preprocessor directives) is a module definition.

All other definitions must be nested inside modules.

Modules can be reopened and can be nested.

```
module Example {
    // Some definitions here.
};
module Example {
    // More definitions here...
    module Nested { /* … */};
};
```

ZeroC

The Ice Modules

Ice uses a number of top-level modules: `Ice`, `Freeze`, `Glacier2`, `IceBox`, `IcePatch2`, `IceStorm`, and `IceGrid`.

- The Ice module contains definitions for basic run time features.

- The remaining modules contain definitions for specific services.

Almost all of the Ice run time APIs are defined in Slice. This automatically defines the API for all implementation languages.

Only a few key functions (the initialization for the run time) and a few language-specific helper functions are defined natively.

Basic Slice Types

Slice provides a number of built-in basic types:

Type	Range of Mapped Type	Size of Mapped Type
bool	false or true	≥ 1 bits
byte	-128-127 or 0-255	≥ 8 bits
short	-2^{15} to 2^{15}-1	≥ 16 bits
int	-2^{31} to 2^{31}-1	≥ 32 bits
long	-2^{63} to 2^{63}-1	≥ 64 bits
float	IEEE single-precision	≥ 32 bits
double	IEEE double-precision	≥ 64 bits
string	All Unicode glyphs, excluding the character with all bits zero.	Variable-length

Enumerations

Enumerations are much like their Java counterpart:

```
enum Fruit { Apple, Pear, Orange };
```

You cannot specify the value of the enumerators:

```
enum Fruit { Apple=0, Pear=7, Orange=2 }; // Illegal!
```

As for C++ (and unlike Java), enumerators enter the namespace enclosing the enumeration:

```
enum Fruit { Apple, Pear, Orange };
enum ComputerBrands { Apple, IBM, Sun, HP }; // Error!
```

Empty enumerations are illegal.

Structures

Structures contain at least one member of arbitrary type:

```
struct TimeOfDay {
    short hour;         // 0-23
    short minute;       // 0-59
    short second;       // 0-59
};
```

The name of the structure, `TimeOfDay`, becomes a type name in its own right. (There are no typedefs in Slice.)

Structures form a namespace, the member names must be unique only within their enclosing structure.

Members may optionally declare a default value.

Sequences

Sequences are (possibly empty) variable-length collections:

```
sequence<Fruit> FruitPlatter;
```

The element type can be anything, including another sequence type:

```
sequence<FruitPlatter> FruitBanquet;
```

The order of elements is never changed during transmission; sequences are ordered collections.

Use sequences to model collections, such as sets, lists, arrays, bags, queues, and trees.

Use sequences to model optional values:

```
sequence<string> InitialOpt;

struct Person {
    string      firstName;
    InitialOpt  initial;
    string      lastname;

};
```

ZeroC

Dictionaries

A dictionary is a map of key-value pairs:

```
struct Employee {
    long    number;
    string firstName;
    string lastName;
};

dictionary<long, Employee> EmployeeMap;
```

Use dictionaries to model maps and sparse arrays.

Dictionaries map to efficient lookup data structures, such as STL maps or hash tables.

The key type of a dictionary must be one of:

- An integral type (`bool`, `byte`, `short`, `int`, `long`, `enum`) or `string`

- A structure containing only members of integral type or type `string`

ZeroC

Constants and Literals

Slice permits constants of type:

- `bool, byte, short, int, long`
- enumerated type
- `float` and `double`
- `string`

Examples:

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double    PI = 3.1416;

enum Fruit { Apple, Pear, Orange };
const Fruit FavoriteFruit = Pear;
```

Slice does not support constant expressions.

Interfaces

Interfaces define object types:

```
struct TimeOfDay { /* ... */ };

interface Clock {
    TimeOfDay getTime();
    void setTime(TimeOfDay time);
};
```

- Interfaces define the public interface of an object. There is no notion of a private part of an object in Slice.

- Interfaces only have operations, not data members. (Data members are implementation state, not interface.)

- Invoking an operation on an interface sends a (possibly remote) invocation (RPC) to the target object.

- Interfaces define the smallest and only granularity of distribution: if something does not have an interface (or Slice class, which is also an interface), it cannot be invoked remotely.

ZeroC

Operations and Parameters

An interface contains zero or more operation definitions.
Each operation definition has:

- – an operation name

- – a return type (or `void` if none)

- – zero or more parameters

- – an optional `idempotent` modifier

- – an optional exception specification

If an operation has `out`-parameters, they must follow in-parameters.

Operations cannot be overloaded.

```
interface Example {
    void op();
    int otherOp(string p1, out string p2);
};
```

ZeroC

idempotent Operations

An `idempotent` operation is an operation that, if invoked twice, has the same effect as if it is invoked once:

```
idempotent void setName(string name);
idempotent string getName();
```

The `idempotent` keyword affects the error-recovery behavior of the Ice run time: for normal operations, the run time has to be more conservative to preserve at-most-once semantics.

User Exceptions

Operations can throw exceptions:

```
exception Error {};  // Empty exceptions are legal

exception RangeError {
    TimeOfDay errorTime;
    TimeOfDay minTime;
    TimeOfDay maxTime;
};

interface Clock {
    idempotent TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time)
        throws RangeError, Error;
};
```

Operations *must* declare the exceptions they can throw in the exception specification.

Exceptions are *not* data types: they cannot be used as data members or parameters.

ZeroC

Exception Inheritance

Exceptions can form single-inheritance hierarchies:

```
exception ErrorBase {
    string reason;
};

enum RTError {
    DivideByZero, NegativeRoot, IllegalNull /* ... */
};

exception RuntimeError extends ErrorBase {
    RTError err;
};
```

An operation that specifies a base exception in its exception specification can throw the base exception and any exceptions derived from the base:

```
void op() throws ErrorBase; // Can throw RuntimeError
```

Derived exceptions cannot redefine data members defined in a base.

Ice Run-Time Exceptions

Any operation (whether it has an exception specification or not) can always throw Ice run-time exceptions.

Run-time exceptions capture common error conditions, such as out of memory, connect timeout, etc.

Three exceptions have special meaning:

- `UnknownException`

    The operation in the server has thrown a non-Ice exception (such as `java.lang.ClassCastException`).

- `UnknownUserException`

    The operation has thrown an exception that is not in its exception specification.

- `UnknownLocalException`

    The operation on the server-side has thrown a run-time exception that is not marshaled back to the client. (See next slide.)

Run-Time Exceptions Raised by the Server

There are three exceptions that can be received from the remote end:

- `ObjectNotExistException`

  The client has called an operation via a proxy that denotes a servant that does not exist. Most likely cause: the object existed in the past but has since been destroyed.

- `OperationNotExistException`

  The client has invoked an operation that the target object does not support. Most likely cause: client and server were compiled with mismatched Slice definitions.

- `FacetNotExistException`

  The client has called an operation via a proxy that denotes an existing object, but the specified facet does not exist. Most likely cause: the client specified an incorrect facet name, or the facet existed in the past but has since been destroyed.

Proxies

Proxies are the distributed equivalent of class pointers or references:

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

dictionary<string, Clock*> TimeMap;  // Time zones

exception BadZoneName { /* ... */ };

interface WorldTime {
    idempotent Clock* findZone(string zoneName) throws BadZoneName;
    idempotent TimeMap listZones();
};
```

The * operator is known as the *proxy operator*.

Interface Inheritance

Interfaces support inheritance:

```
interface AlarmClock extends Clock {
    TimeOfDay getAlarmTime();
    void setAlarmTime(TimeOfDay alarmTime)
                                throws BadTimeVal;
};
```

Multiple inheritance is legal as well:

```
interface Radio {
    void setFrequency(long hertz) throws GenericError;
    void setVolume(long dB) throws GenericError;
};

enum AlarmMode { RadioAlarm, BeepAlarm };

interface RadioClock extends Radio, AlarmClock {
    void        setMode(AlarmMode mode);
    AlarmMode getMode();
};
```

Interface Inheritance Limitations

An interface cannot inherit an operation with the same name
from more than one base interface:

```
interface Clock {
    void set(TimeOfDay time); // set time
};

interface Radio {
    void set(long hertz);      // set frequency
};

interface RadioClock extends Radio, Clock { // Illegal!
    // ...
};
```

There is no concept of overriding or overloading.

Implicit Inheritance from Object

All interfaces implicitly inherit from `Object`, which is the root of the inheritance hierarchy.

```
interface ProxyStore {
    void    putProxy(string name, Object* o);
    Object* getProxy(string name);
};
```

Because any proxy is assignment compatible with `Object`, `ProxyStore` can store and return proxies for any interface type.

Explicit inheritance from Object is illegal:

```
interface Wrong extends Object { // Error!
    // ...
};
```

ZeroC

Self-Referential Interfaces & Forward Declarations

Interfaces can be self-referential:

```
interface Node {
    int val();
    Node* next();
};
```

You can forward-declare an interface to create interfaces that mutually refer to each other:

```
interface Wife; // Forward declaration

interface Husband {
    Wife* getWife();
};

interface Wife {
    Husband* getHusband();
};
```

Classes

Classes can contain data members as well as operations.

Classes support single implementation and multiple interface inheritance. They implicitly derive from `Object` (just like interfaces).

One way to use classes is as structures that are extensible by inheritance:

```
class TimeOfDay {
    short hour;          // 0 - 23
    short minute;        // 0 - 59
    short second;        // 0 - 59
};
class DateTime extends TimeOfDay {
    short day;           // 1 - 31
    short month;         // 1 - 12
    short year;          // 1753 onwards
};
```

Empty classes are legal.

Data members may define default values, as with structures.

Classes as Parameters and Slicing

Classes are passed *by value*, just like structures.

You can pass a derived class where a base is expected:

```
interface Clock {
    void setTime(TimeOfDay t);
};
```

You can pass a `TimeOfDay` instance or a `DateTime` instance to `setTime`.

The receiver gets the most-derived type that it has static type knowledge of:

- If the server was linked with the stubs for both `TimeOfDay` and `DateTime`, the server receives a `DateTime` instance (as the static type `TimeOfDay`).

- If the server was linked with the stubs for only `TimeOfDay`, the `DateTime` object is sliced to `TimeOfDay` in the server.

Use classes if you need polymorphic *values* (instead of *interfaces*).

Classes as Unions

You can use derivation from a common base class to model unions.

It is often useful to include a discriminator in the base class, so the receiver can use a `switch` statement to find which member is active (instead of an `if-then-else` chain of dynamic casts).

```
class UnionDiscriminator {
    int d;
};
class Member1 extends UnionDiscriminator {
    // d == 1
    string s;
};
class Member2 extends UnionDiscriminator {
    // d == 2
    double d;
};
```

Self-Referential Classes

Like interfaces, classes can be self-referential:

```
class Link {
    SomeType value;
    Link next; // Note: NOT Link* !
};
```

This looks like `Link` includes itself but really means that `next` contains a pointer to another `Link` instance that is in the same address space.

Passing an instance of `Link` as a parameter passes *the entire chain* of instances to the receiver.

You can use self-referential classes to model arbitrary graphs.

Passing a node of the graph as a parameter marshals the entire graph that is reachable using that node as a starting point.

Cyclic graphs are permitted, as are graphs with nodes of in-degree > 1.

Forward declarations are legal (with the same syntax as for interfaces).

Classes with Operations

```
class TimeOfDay {
    short hour;          // 0 - 23
    short minute;        // 0 - 59
    short second;        // 0 – 59

    string format();
};
```

Classes with operations are mapped to abstract base classes with abstract methods.

The application provides the implementation for the operations.

Invoking an operation on a class invokes the operation in the local address space of the class.

It follows that, if a class with operations is sent as a parameter, the code for the operation must exist at the receiving end. The Ice run time only marshals the data, not the code.

Classes with operations allow you to implement client-side processing.

Classes Implementing Interfaces

```
interface Time {
    idempotent TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time);
};

interface Radio {
    idempotent void setFrequency(long hertz);
    idempotent void setVolume(long dB);
};

class RadioClock implements Time, Radio {
    TimeOfDay time;
    long hertz;
};
```

Classes can implement one or more interfaces (in addition to extending a single other class).

The derived class inherits all of the operations of its base interface(s).

ZeroC

Class Inheritance Limitations

Operations and data members must be unique within a hierarchy:

```
interface BaseInterface {
    void op();
};

class BaseClass {
    int member;
};

class DerivedClass
        extends BaseClass
        implements BaseInterface {
    void someOperation();        // OK
    int op();                    // Error!
    int someMember;              // OK
    long member;                 // Error!
};
```

As for interfaces, you cannot inherit the same operation from different base interfaces.

Pass-by-Value Versus Pass-by-Reference

You can create proxies to classes:

```
class TimeOfDay { /* ... */ };

interface Clock {
    TimeOfDay  getTime();        // Returns class
    TimeOfDay* getTimeProxy();   // Returns proxy
};
```

Invoking an operation on a *class* invokes the operation locally.

Invoking an operation on a *proxy* invokes the operation remotely.

Only operations (but not data members) of a class are accessible via its proxy.

You can also pass an interface by value:

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

interface Clock {
    void set(Time t); // Note: NOT Time* !
};
```

Architectural Implications of Classes

- Classes enable client-side processing and avoid RPC overhead.

- The price is that the behavior of (that is, the code for) class operations must be available wherever the class is used.

- If you have a C++ class with operations, and want to use it from a Java client, you must re-implement the operations of the class in Java, with identical semantics.

- Classes with operations destroy language- and OS-transparency (if they are passed by value).

- Use classes with operations only if you can control the deployment environment for the entire application!

ZeroC

Classes Versus Structures

Classes can model structures, so why have structures?

Structures are more efficient because they can be stack-allocated whereas classes are always heap-allocated.

Classes are slower to marshal than structures, and consume more bandwidth on the wire.

Use classes if you need one or more features not provided by structures:

- inheritance
- pointer semantics
- client-side local operations
- choice of local versus remote invocation

ZeroC

The :: Scope Qualification Operator

The :: scope resolution operator allows you to refer to types that
are not in the current scope or immediately enclosing scope:

```
module Types {
    sequence<long> LongSeq;
};

module MyApp {
    sequence<Types::LongSeq> NumberTree;
};
```

You can anchor a lookup explicitly at the global scope with a
leading :: operator: ::Types::LongSeq

Type Identifiers

Each Slice type has a unique internal identifier, call the type ID:

- For built-in types, the type ID is the name of the type, e.g. `int` or `string`.

- For user-defined types, the type name is the fully-scoped name:
  ```
  module Times {
      struct Time { /* ... */ };

      interface Clock { /* ... */ };
  };
  ```
  The type IDs for this definition are `::Times`, `::Times::Time`, and `::Times::Clock`.

- For proxies, the type ID has a trailing `*`, so the type ID of the proxy for the `Clock` interface is `::Times::Clock*`.

Operations on `Object`

All interfaces and classes implicitly inherit from `Object`:

```
sequence<string> StringSeq;

interface Object {                          // "Pseudo" Slice!
    void       ice_ping();
    bool       ice_isA(string typeID);
    string     ice_id();
    StringSeq ice_ids();
    // ...
};
```

- `ice_ping` provides a basic reachability test.

- `ice_isA` tests whether an interface is compatible with the supplied type.

- `ice_id` returns the type ID of the interface.

- `ice_ids` returns all types IDs of the interface (the type ID of the interface itself, plus the type IDs of all base interfaces).

Local Types

The APIs for the Ice run time are (almost) completely defined in Slice. Most of the Slice definitions use the `local` keyword, for example:

```
module Ice {
    local interface Communicator { /* ... */ };
};
```

`local` types cannot be accessed remotely; they define library objects.

`local` types do *not* inherit from `Object`. Instead, they derive from a common base `LocalObject`.

Therefore, you cannot pass a local object where a non-local object is expected and vice-versa.

You can define your own `local` interfaces, but there will rarely be a need to do so.

ZeroC

Metadata

Any Slice construct can be preceded by a metadata directive, for example:

```
["java:type:java.util.LinkedList<Integer>"] sequence<int> IntSeq;
```

Metadata directives can also appear at global scope:

```
[["java:package:com.acme"]]
```

Global metadata directives must precede any Slice definitions in a source file.

Metadata directives affect the code generator only.

Metadata directives *never* affect the client–server contract: no matter how you add, remove, or change metadata directives, the information that is exchanged on the wire is always the same.

The `slice2java` Compiler

The `slice2java` command compiles one or more Slice definition files.

`slice2java [options] file...`

For example:

`slice2java MyDefs.ice`

This generates a number of source files, one for each class, using the usual directory hierarchy for modules (which map to Java packages).

Commonly used options:

- **-D**_NAME_, -D_NAME=DEF_, -U_NAME_

  Define or undefine preprocessor symbol _NAME_.

- -I _DIR_

  Add _DIR_ to the search path for `#include` directives.

- --impl

  Create sample implementation files.

Ice Programming with Java

3. Assignment 1
Creating Slice Definitions

Exercise Overview

In this exercise, you will:

- gain hands-on experience of how to create Slice definitions by designing interfaces for a simple application.

By the completion of this exercise, you will have gained experience in creating Slice definitions, the syntax and semantics of the language, and how to use the `slice2java` compiler.

Simple Remote File System

Functionality

- The file system consists of directories and files. The usual hierarchical structure applies, so the file system has a single root directory that, recursively, can contain other directories and files.

- Each directory and file has a name; names within the same parent directory must be unique, as for a Windows or UNIX file system.

- Directories provide a way to list their contents.

- The content of files can be read and written. (Only text files are supported, not binary files.)

- For the time being, the file system does not permit life cycle operations, that is, clients can read and write the contents of files and list the contents of directories, but cannot create or delete files or directories.

What You Need to Do

Create Slice definitions for this application.

- In your `lab1` directory, locate the file named `Filesystem.ice`.
- Place your definitions into this file. The directory also contains a project file `build.xml` that you can use to compile your definitions.

Consider the following:

- What interfaces need to be present in your definitions, and how they should relate to each other.
- What error conditions can arise and how to best inform clients of any errors.
- What interaction patterns are clients likely to exhibit. Would it be advisable to modify your definitions to accommodate such patterns and, if so, why?

Once you have compiled your definitions, have a look at the generated code. What parts of your specification do you recognize in the generated code?

One Possible Solution

```
module Filesystem {
    exception IOError {
        string reason;
    };

    interface Node {
        idempotent string name();
    };

    sequence<string> Lines;

    interface File extends Node {
        idempotent Lines read() throws IOError;
        idempotent void write(Lines text) throws IOError;
    };

    sequence<Node*> NodeSeq;

    interface Directory extends Node {
        idempotent NodeSeq list();
    };
};
```

Ice Programming with Java

4. Client-Side
Slice-to-Java Mapping

Lesson Overview

- This lesson presents:

  - the mapping from Slice to Java for the client side.

  - the relevant APIs that are necessary to initialize and finalize the Ice run time

  - instructions for compiling a Java Ice client.

- By the end of this lesson, you will know how each Slice type maps to Java and be able to write a working Ice client.

Client-Side Java Mapping

The client-side Java mapping defines rules for:

- initializing and finalizing the Ice run time
- mapping each Slice type into Java
- invoking operations and passing parameters
- handling exceptions

The mapping is fully thread-safe: you need not protect any Ice-internal data structures against concurrent access.

The mapping rules are simple and regular: know them! The generated files are no fun to read at all!

slice2java-generated code is platform independent.

Initializing the Ice Run Time

```
public static void main(String[] args)
{
    int status = 1;
    Ice.Communicator ic = null;
    try {
        ic = Ice.Util.initialize(args);
        // client code here...
        status = 0;
    } catch (Exception e) {
    }
    finally {
        if (ic != null) {
            try {
                ic.destroy();
            } catch (Exception e) {
            }
        }
    }
    System.exit(status);
}
```

ZeroC

Mapping for Identifiers

Slice identifiers map to corresponding Java identifiers:

```
struct Employee {
    int number;
    string name;
};
```

The generated Java contains:

```
public class Employee
    implements java.lang.Cloneable, java.io.Serializable
{
    public int number;
    public String name;
    // ...
}
```

Slice identifers that clash with Java keywords are escaped with a `_` prefix, so Slice `while` maps to Java `_while`.

Mapping for Modules

Slice modules map to Java packages. The nesting of definitions is preserved:

```
module M1 {
    module M2 {
        // ...
    };
    // ...
};
```

This maps to Java as:

```
package M1;
// Definitions for M1 here...

package M1.M2;
// Definitions for M1.M2 here...
```

Mapping for Built-In Types

The built-in Slice types map to Java types as follows:

Slice Type	Java Type
bool	boolean
byte	byte
short	short
int	int
long	long
float	float
double	double
string	String

Mapping for Enumerations

Slice enumerations map unchanged to the corresponding Java enumeration.

```
enum Fruit { Apple, Pear, Orange };
```

This maps to the Java definition:

```
public enum Fruit implements java.io.Serializable {
    Apple,
    Pear,
    Orange;
    // ...
}
```

Mapping for Structures

Slice structures map to Java classes with all data members public:

```
struct Employee {
    string lastName;
    string firstName;
};
```

This maps to:

```
public class Employee
        implements java.lang.Cloneable, java.io.Serializable {
    public String lastName;
    public String firstName;

    public Employee();
    public Employee(String lastName, String firstName);
    public boolean equals(java.lang.Object rhs);
    public int hashCode();
    public java.lang.Object clone();
};
```

Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

Mapping for Sequences

By default, Slice sequences map to Java arrays.

```
sequence<Fruit> FruitPlatter;
```

No code is generated for the sequence. Use it as you would any other array, for example:

```
Fruit[] platter = { Fruit.Apple, Fruit.Pear };


assert(platter.length == 2);
```

Custom Mapping for Sequences

You can change the mapping for a sequence to a custom type:

```
["java:type:java.util.LinkedList<Fruit>"]
sequence<Fruit> FruitPlatter;
```

The nominated type must implement the `java.util.List<T>` interface. You can override members, parameter, or return values, for example:

```
sequence<Fruit> Breakfast;
["java:type:java.util.LinkedList<Fruit>"]
sequence<Fruit> Dessert;

struct Meal1 {
    Breakfast b;
    Dessert d;
};

struct Meal2 {
    ["java:type:java.util.LinkedList<Fruit>"] Breakfast b;
    ["java:type:java.util.Vector<Fruit>"] Dessert d;
};
```

![ZeroC]

Mapping for Dictionaries

Slice dictionaries map to Java maps:

```
dictionary<long, Employee> EmployeeMap;
```

No code is generated for this dictionary. Rather, `slice2java` substitutes `java.util.Map<Long, Employee>` for `EmployeeMap`.

It follows that you can use the dictionary like any other Java map, for example:

```
java.util.Map<Long, Employee> em =
    new java.util.HashMap<Long, Employee>();

Employee e = new Employee();

e.number = 31;
e.firstName = "James";
e.lastName = "Gosling";

em.put(e.number, e);
```

Custom Mapping for Dictionaries

You can change the default mapping via metadata:

```
["java:type:java.util.LinkedHashMap<String, String>"]
dictionary<string, string> StringTable;
```

The type specified for the dictionary must support the

`java.util.Map<K, V>` interface.

As for sequences, you can override the type for individual members, parameters, and return values.

ZeroC

Mapping for Constants

Slice constants map to a Java interface with a `value` member that stores the value.

```
const string   Advice = "Don't Panic!";


enum Fruit { Apple, Pear, Orange };
const Fruit    FavoriteFruit = Pear;
```

This maps to:

```
public interface Advice {
    String value = "Don't Panic!";
}


public interface FavouriteFruit {
    Fruit value = Fruit.Pear;
}
```

Mapping for User Exceptions

User exceptions map to Java classes derived from `UserException`.

```
exception GenericError {
    string reason;
};
```

This maps to:

```
public class GenericError extends Ice.UserException {
    public String reason;
    public GenericError();
    public GenericError(String reason);
    public String ice_name() {
        return "GenericError";
    }
}
```

Slice exception inheritance is preserved in Java, so if Slice exceptions are derived from `GenericError`, the corresponding Java exceptions are derived from `GenericError`.

Mapping for Run-Time Exceptions

Ice run-time exceptions are derived from `Ice.LocalException`. In turn, `Ice.LocalException` derives from `java.lang.RuntimeException`.

As for user exceptions, `Ice.LocalException` provides an `ice_name` method that returns the name of the exception.

ZeroC

Mapping for Interfaces

A Slice interface maps to a number of classes.

```
interface Simple {
    void op();
};
```

This generates the following interfaces and classes:

```
interface    Simple
final class SimpleHolder


interface    SimplePrx
final class SimplePrxHolder
final class SimplePrxHelper


interface    _SimpleOperations
interface    _SimpleOperationsNC
```

The Proxy Interface

An instance of a proxy interface acts as the local ambassador for a remote object. Invoking a method on the proxy results in an RPC call to the corresponding object in the server.

```
interface Simple {
    void op();
};
```

This generates:

```
public interface SimplePrx extends Ice.ObjectPrx {
    public void op();
    public void op(java.util.Map<String, String> __ctx);
}
```

The version without the `__ctx` parameter simply calls the version with the `__ctx` parameter, supplying a default context.

`SimplePrx` derives from `Ice.ObjectPrx`, so all proxies support the operations on `Ice.Object`.

Methods on Ice.ObjectPrx

Ice.ObjectPrx is defined as follows:

```
package Ice;

public interface ObjectPrx {
    boolean equals(java.lang.Object r);
    int hashCode();
    Identity ice_getIdentity();
    boolean ice_isA(String __id);
    String ice_id();
    String[] ice_ids();
    void ice_ping();
    // ...
}
```

Every Ice object supports these operations.

ZeroC

Proxy Helpers

For each interface, the compiler generates a helper class that allows you to do type-safe down-casts:

```
public final class SimplePrxHelper extends Ice.ObjectPrxHelper {
    public static SimplePrx checkedCast(Ice.ObjectPrx b);
    public static SimplePrx checkedCast(
        Ice.ObjectPrx b,
        java.util.Map<String, String> ctx);
    public static SimplePrx uncheckedCast(Ice.ObjectPrx b);
    // ...
}
```

Both casts test an is-a relationship.

- A `checkedCast` checks with the server whether the object actually supports the specified type and so requires sending a message.

- An `uncheckedCast` is a sledgehammer cast (so you had better get it right!) but does not require sending a message.

Mapping for Operations

Slice operations map to methods on the proxy interface.

```
interface Simple {
    void op();
};
```

Invoking a method on the proxy instance invokes the operation in the remote object:

```
SimplePrx p = ...;

p.op(); // Invoke remote op() operation
```

The mapping is the same, regardless of whether an operation is a normal operation or has an `idempotent` qualifier.

Mapping for Return Values and In-Parameters

Return values and In-parameters are passed either by value (for simple types), or by reference (for complex types).

```
interface Example {
    string op(double d, string s);
};
```

The proxy operation is:

```
String op(double d, String s);
```

You invoke the operation like any other Java method:

```
ExamplePrx p = ...;

String result = p.op(3.14, "Hello");
System.out.writeln(result);
```

- To pass a null proxy, pass a null reference.
- You can pass a null parameter for strings, sequences, and dictionaries to pass the empty string, sequence, or dictionary.

Mapping for Out-Parameters

Out-parameters are passed via a `Holder` type:

- Built-in types are passed as `Ice.ByteHolder`, `Ice.IntHolder`, `Ice.StringHolder`, etc. User-defined types are passed as `<name>Holder`.

All holder classes have a public `value` member, for example:

```java
package Ice;

public final class StringHolder {
    public StringHolder() {}
    public StringHolder(String value) {
        this.value = value;
    }
    public String value;
}
```

You pass a holder instance where an out-parameter is expected; when the operation completes, the `value` member contains the returned value.

Exception Handling

Operation invocations can throw exceptions:

```
exception Tantrum { string reason; };

interface Child {
    void askToCleanUp() throws Tantrum;
};
```

You can call askToCleanUp like this:

```
ChildPrx child = ...; // Get proxy...
try {
    child.askToCleanUp(); // Give it a try...
} catch (Tantrum t) {
    System.out.writeln("The child says: " + t.reason);
}
```

Exception inheritance allows you to handle errors at different levels with handlers for base exceptions at higher levels of the call hierarchy.

The value of out-parameters if an exception is thrown is undefined.

Mapping for Classes

Slice classes map to Java classes:

- For each Slice member (if any), the class contains a corresponding public data member.

- If the class has operations, it is abstract and derives from the `_<name>Operations` and `_<name>OperationsNC` interfaces. These interfaces contain method definitions corresponding to the Slice operations.

- The class has a default constructor and a "one-shot" constructor with one parameter for each class member.

- Slice classes without a base class derive from `Ice.Object`.

- Slice classes with a base class derive from the corresponding base class.

- All classes support the operations on `Ice.Object`.

Inheritance from Ice.Object

Classes support the methods on Ice.Object:

```
package Ice;

public interface Object
{
    void ice_ping(Current current);
    boolean ice_isA(String s, Current current);
    String[] ice_ids(Current current);
    String ice_id(Current current);

    int ice_hash();
    void ice_preMarshal();
    void ice_postUnmarshal();
}
```

Abstract Classes

Classes that inherit methods from the _<*name*>Operations interface are abstract, so they cannot be instantiated.

To allow abstract classes to be instantiated, you must create a class that derives from the compiler-generated class. The derived class must provide implementations of the operations:

```
public class TimeOfDayI extends TimeOfDay {
    public String format(Ice.Current current) {
        DecimalFormat df
            = (DecimalFormat)DecimalFormat.getInstance();
        df.setMinimumIntegerDigits(2);
        return new String(df.format(hour) + ":" +
                          df.format(minute) + ":" +
                          df.format(second));
    }
}
```

By convention, implementations of abstract classes have the name <*class-name*>I.

Class Factories

The Ice run time does not know how to instantiate an abstract class unless you tell it how to do that:

```
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };
    // ...
};
```

You must implement the `ObjectFactory` interface and register a factory for each abstract class with the Ice run time.

- The run time calls `create` when it needs to create a class instance.
- The run time calls `destroy` when you destroy the communicator.

ZeroC

Factory Registration

Once you have created a factory class, you must register it with the Ice run time for a particular type ID:

```
module Ice {
    local interface Communicator {
        void addObjectFactory(ObjectFactory factory,
                                    string id);
        ObjectFactory findObjectFactory(string id);
        // ...
    };
};
```

When the run time needs to unmarshal an abstract class, it calls the factory's `create` method to create the instance.

It is legal to register a factory for a non-abstract Slice class. If you do this, your factory overrides the one that is generated by the Slice compiler.

Default Factory

You can register a factory for the empty type ID as a default factory. The Ice run time locates factories in the following order:

1. Look for a factory registered for the specific type ID. If one exists, call `create` on that factory. If the return value is non-null, return the result, otherwise try step 2.

2. Look for the default factory. If it exists, call `create` on the default factory. If the return value is non-null, return the result, otherwise try step 3.

3. Look for a Slice-generated factory (for non-abstract classes). If it exists, instantiate the class.

4. Throw `NoObjectFactoryException`.

If you have both a type-specific factory and a default factory, you can return null from the type-specific factory to redirect class creation to the default factory.

Stringified Proxies

The simplest stringified proxy specifies:

- host name (or IP address)

- port number

- an object identity

For example:

`fred: tcp -h myhost.dom.com -p 10000`

General syntax:

<identity>: <endpoint>[: <endpoint>...]

For TCP/IP, the endpoint is specified as:

`tcp -h` *<host name or IP address>* `-p` *<port number>*

To convert a stringified proxy into a live proxy, use:

`Communicator.stringToProxy.`

A null proxy is represented by the empty string.

ZeroC

Compiling and Running a Client

To compile a client, you must:

- compile the Slice-generated source files(s)

- compile your application code

For Linux:

```
$ mkdir classes
$ javac -d classes -classpath \
> classes:$ICEJ_HOME/lib/Ice.jar \
> Client.java generated/Demo/*.java
```

To compile and run the client, `Ice.jar` must be in your

`CLASSPATH`.

ZeroC

Ice Programming with Java

5. Assignment 2
Creating an Ice Client

Exercise Overview

In this exercise, you will:

- create an Ice client to access a server that implements the filesystem developed in Assignment 1.

By the completion of this exercise, you will have gained experience in the Java language mapping, how to initialize and finalize the Ice run time, how to construct proxies, and how to invoke operations and handle exceptions.

Creating a Client for the Remote Filesystem

- In your `lab2` directory, you will find a `build.xml` file to build a client and a server.

- The server is complete and implements the file system defined in `Filesystem.ice`.

- The server listens on port 10000 for incoming requests; the identity of the root directory object is "RootDir".

What You Need to Do

The client code can be found in `Client.java`.

1. In the body of `main`, initialize the Ice run time, create a proxy to the root directory, and pass that proxy to the `listRecursive` function.

2. Following the call to `listRecursive`, shut down the Ice run time.

3. The body of `listRecursive` is empty, so you need to provide an implementation.

4. Test your client against the provided server.

5. Try running the client without first starting the server.

6. Change the client to use the identity "Fred" for the root directory.

The main Method

Note that the code catches and handles any exceptions, and that the communicator is destroyed only if it was successfully initialized.

ZeroC

The `listRecursive` Method

Note that the code, for each proxy returned by `list`, uses an `uncheckedCast` to down-cast the proxy. This avoids the overhead of using a `checkedCast`, which requires a remote message.

Ice Programming with Java

6. Server-Side
Java Mapping

Lesson Overview

- This lesson presents:

  - the mapping from Slice to Java relevant to the server side.

  - the relevant APIs that are necessary to initialize and finalize the Ice run time

  - how to implement and register object implementations.

- By the end of this lesson, you will be able to write a working Ice server.

Server-Side Java Mapping

All of the client-side Java mapping also applies to the server side.

Additional server-side functionality you must know about:

- how to initialize and finalize the server-side run time
- how to implement servants
- how to pass parameters and throw exceptions
- how to create servants and register them with the run time

Initializing the Ice Run Time

```java
public class Server {
    public static void
    main(String[] args)
    {
        int status = 1;
        Ice.Communicator ic = null;
        try {
            ic = Ice.Util.initialize(args);
            // server code here...
            status = 0;
        } catch (Exception e) {
        }
        if (ic) {
            try {
                ic.destroy();
            } catch (java.lang.Exception ex)
            {
            }
        }
        System.exit(status);
    }
}
```

ZeroC

Server-Side Initialization

Servers must create at least one object adapter, activate that adapter, and then wait for the Ice run time to shut down:

```
ic = Ice.Util.initialize(args);
Ice.ObjectAdapter adapter
    = ic.createObjectAdapterWithEndpoints(
        "MyAdapter", "tcp -p 10000");

// Instantiate and register one or more servants here...

adapter.activate();
ic.waitForShutdown();
```

An object adapter provides one or more endpoints at which the server listens for incoming requests. An adapter has a name that must be unique within its communicator.

Adapters must be activated before they start accepting requests.

You must call `waitForShutdown` from the main thread to wait for the server to shut down (or otherwise prevent the main thread from exiting).

Mapping for Interfaces

Interfaces map to skeleton classes with an abstract method for each Slice operation:

```
module M {
    interface Simple {
        void op();
    };
};
```

This generates:

```
package M;
public interface _SimpleOperations
{
    void op(Ice.Current current);
}
public interface Simple extends Ice.Object,
                                _SimpleOperations,
                                _SimpleOperationsNC;

public abstract class _SimpleDisp
                extends Ice.ObjectImpl
                implements Simple;
```

Mapping for Interfaces (1)

You *must* implement all abstract methods that are inherited from the skeleton class.

You can add whatever else you need to support your implementation:

- constructors and finalizers

- public or private methods

- public or private data members

- other base interfaces

Mapping for Parameters

Server-side operation signatures are identical to the client-side operation signatures (except for a trailing parameter):

- In-parameters are passed by value or by reference.

- Out-parameters are passed by holder types.

- Return values are passed by value or by reference.

- Every operation has a single trailing parameter of type `Ice.Current`.

```
string op(int a, string b, out float c, out string d);
```
Maps to:
```
String op(int a, String b,
          Ice.FloatHolder c, Ice.StringHolder d,
          Ice.Current __current);
```

Throwing Exceptions

```
exception GenericError { string reason; };

interface Example {
    void op() throws GenericError;
};
```

You could implement op as:

```
public void op(Ice.Current c) throws GenericError
{
    throw new GenericError("something failed");
}
```

Do not throw Ice run-time exceptions. You can throw `ObjectNotExistException`, `OperationNotExistException`, or `FacetNotExistException`, which are returned to the client unchanged. But these have specific meaning and should not be used for anything else.
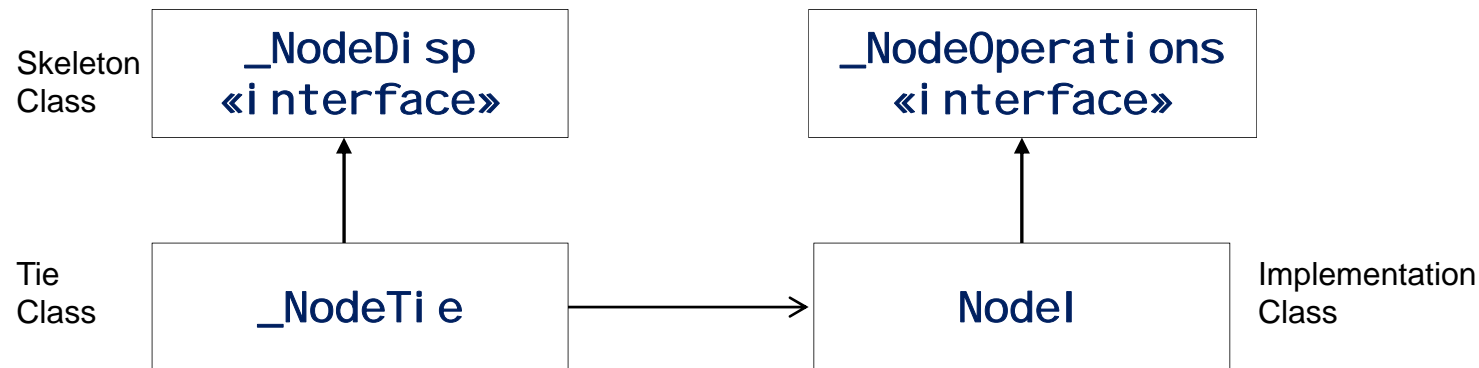
If you throw any other run-time exception, the client will get an `UnknownLocalException` or `UnknownException`.

Tie Classes

Tie classes are an alternative mechanism for implementing servants.

The `--tie` option for `slice2java` generates tie classes in addition to the normal server-side code.

Tie classes replace inheritance with delegation. This way, your implementation class need not inherit from the skeleton class:

Skeleton Class	**_NodeDisp** «interface»
Tie Class	**_NodeTie**

_NodeDisp «interface» ← _NodeTie → NodeI

_NodeOperations «interface» ← NodeI (Implementation Class)

Use the tie mapping when your implementation class must inherit from some other application class (and therefore cannot be derived from the skeleton class).

Creating an ObjectAdapter

Each server must have at least one object adapter. You create an adapter with:

```
local interface ObjectAdapter;

local interface Communicator {
    ObjectAdapter createObjectAdapter(string name);
    ObjectAdapter createObjectAdapterWithEndpoints(
                            string name,
                            string endpoints);
    // ...
};
```

The endpoints at which the adapter listens are taken from configuration (first version), or from the supplied argument (second version).

Example endpoint specification:

```
tcp -p 10000:udp -p 10000:ssl -p 10001
```

Endpoints have the general form:

```
<protocol> [-h <host>] [-p <port>] [-t timeout] [-z]
```

Object Adapter States

An object adapter is in one of three possible states:

- Holding (initial state after creation)

  The adapter does not read incoming requests off the wire (for TCP and SSL) and throws incoming UDP requests away.

- Active

  The adapter processes incoming requests. You can transition freely between the Holding and Active state.

- Inactive

  This is the final state, entered when you initiate destruction of the adapter:

  – Requests in progress are allowed to complete.

  – New incoming requests are rejected with a `ConnectionRefusedException`.

Controlling Adapter State

The following operations on the adapter relate to its state:

```
local interface ObjectAdapter {
    void activate();
    void hold();
    void deactivate();
    void waitForHold();
    void waitForDeactivate();
    void destroy();

    // ...
};
```

The operations to change state are non-blocking.

If you want to know when a state transition is complete, call `waitForHold` or `waitForDeactivate` as appropriate.

`destroy` blocks until deactivation completes.

You can re-create an adapter with the same name once `destroy` completes.

Object Identity

Each Ice object has an associated object identity.

Object identity is defined as:

```
struct Identity {
    string name;
    string category;
};
```

- The `name` member gives each Ice object a unique name.

- The `category` member is primarily used in conjunction with default servants and servant locators. If you do not use these features, the category is usually left as the empty string.

The identity must be unique within the object adapter: no two servants that incarnate an Ice object can have the same identity.

The *combination* of `name` and `category` must be unique.

An identity with an empty `name` denotes a null proxy.

Stringified Object Identity

Two helper functions on the communicator allow you to convert between identities and strings:

```
interface Communicator
{
    Identity stringToIdentity(String ident);
    String identityToString(Identity id);
    // ...
}
```

Stringified identities have the form *<category>/<name>*, for example:
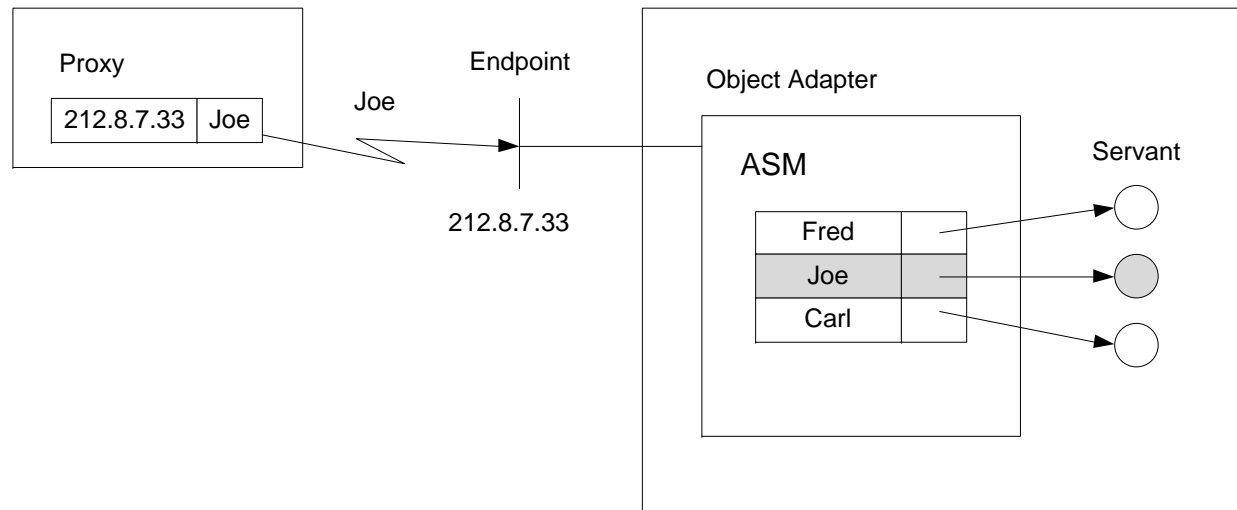
```
    person/fred
```

If no slash is present, the string is used as the name, with the category assumed to be empty.

The object adapter has a `getCommunicator` method that returns the communicator. You use the communicator to convert between strings and object identities.

ZeroC

The Active Servant Map (ASM)

Each adapter maintains a map that maps object identities to servants:

- Incoming requests carry the object identity of the Ice object that is the target.

- The ASM allows the server-side run time to locate the correct servant for the request.

- Object identities must be unique per ASM.

Activating Servants

To make a servant available to the Ice run time, you must activate it. This adds an identity–servant pair to the ASM:

```
local interface ObjectAdapter {
    Object* add(Object servant, Identity id);
    Object* addWithUUID(Object servant);
    // ...
};
```

Both operations return the proxy for the servant, for example:

```
SimplePrx sp = SimplePrxHelper.uncheckedCast(
    adapter.add(new SimpleI("hello"),
                adapter.getCommunicator().
                    stringToIdentity("fred")));
```

As soon as a servant is added to the ASM, the run time will dispatch requests to it (assuming that the adapter is activated).

`addWithUUID` adds the servant to the ASM with a UUID as the name, and an empty category.

Creating Proxies

You can create a proxy without activating a servant for the proxy:

```
interface ObjectAdapter {
    // ...
    Object* createProxy(Identity id);
};
```

The returned proxy contains the passed identity and the adapter's endpoints.

Note that the return type is `Object*` so, typically, you need to downcast the proxy before you can use it.

The `Ice.Application` Class

`Ice.Application` is a utility class that makes it easy to initialize and finalize the Ice run time.

```
public abstract class Application
{
    public Application();
    public final int main(String appName, String[] args);
    public final int main(String appName, String[] args,
                          String configFile);
    public final int main(String appName, String[] args,
                          InitializationData id);
    public abstract int run(String[] args);
    public static Communicator communicator();
    public static String appName();
    // ...
}
```

You call `Application.main` from the real `main`, and implement the body of your client or server in the `run` method.

Shutdown Hook

`Ice.Application` provides control of the JVM shutdown hook:

```
package Ice;

public enum SignalPolicy { HandleSignals, NoSignalHandling }
public abstract class Application
{
    public Application();
    public Application(SignalPolicy signalPolicy);
    // ...
    synchronized public static void destroyOnInterrupt();
    synchronized public static void shutdownOnInterrupt();
    synchronized public static void defaultInterrupt();

    synchronized public static boolean interrupted();
}
```

The default behavior on interrupt is to destroy the communicator, allowing all currently running operation invocations to complete first.

Compiling and Running a Server

To compile a client, you must:

- compile the Slice-generated source files(s)

- compile your application code

For Linux:

```
$ mkdir classes
$ javac -d classes -classpath \
> classes:$ICEJ_HOME/lib/Ice.jar \
> Server.java \
> generated/Demo/*.java
```

To compile and run the server, `Ice.jar` must be in your `CLASSPATH`.

Note that these commands are the same as for the client side—you need not supply server-specific options or use a server-specific class file or library.

Ice Programming with Java

7. Assignment 3
Creating an Ice Server

Exercise Overview

In this exercise, you will:

- create an Ice server that implements the filesystem we developed in Assignment 1.

By the end of this exercise, you will have gained experience in how to implement servants and how to use the `Ice.Application` class to initialize and finalize the Ice run time.

ZeroC

Creating a Server for the Remote Filesystem

- In your `lab3` directory, you will find a `build.xml` file to build a client and a server.

- The client is complete and implements the solution shown in Assignment 2.

- Use this client to test your server.

ZeroC

What You Need to Do

1. Find `Server.java`, `Filesystem/FileI.java`, and `Filesytem/DiretoryI.java`. The places in the code where you need to add something are marked with a `// MISSING` comment.

2. Have a look at the code in `DirectoryI.java`. Add the new directory to the parent's `_contents` member and then add the new directory to the ASM.

3. Have a look at the code in `FileI.java`. Implement the `read` and `write` methods. Use the relevant member variable to store the contents of the file.

4. Implement the missing parts of `Server.java`.

5. Use the provided client to test your server and check that the contents of the file system are as expected.

The Server Class

The Directory Class

The `FileI` Class

ZeroC

Ice Programming with Java

8. Properties
and Configuration

Lesson Overview

- This lesson presents:

  - how to use properties to control various aspects of the Ice run time.

  - how to use properties in your own applications.

- By completion of the chapter, you will know how the Ice run time can be configured using properties, how property values are evaluated, and how to use the property mechanism to configure your own applications.

Ice Properties

The Ice run time and its various subsystems are configured using properties.

- Properties are name–value pairs, e.g.:

  `Ice.UDP.SndSize=65535`

- By convention, Ice property names use the syntax

  `<application>.<category>[.<sub-category>]`

  For your own properties, you can use any number of categories and sub-categories (including none).

- Some property prefixes are reserved for Ice:

  `Ice, IceBox, IceGrid, IcePatch2, IceSSL, IceStorm, Freeze, and Glacier2.`

- Property names are sequences of characters, excluding space (' '), hash ('#'), and ('=').

- Property values are sequences of characters, excluding hash ('#').

Configuration Files

Properties are often set in a configuration file.

- Configuration files contain one property setting per line, e.g.:

```
# Example config file

Ice.MessageSizeMax = 2048    # 2MB message size limit
Ice.Trace.Network=3          # Trace all network activity
Ice.Trace.Protocol=          # No protocol tracing
```

- Leading and trailing white space around a property value are ignored, as are empty lines. Backslashes must be escaped as \\.

- The # character introduces a comment to the end of the current line.

- If a property is set more than once, the last setting takes effect.

- Assigning nothing to a property unsets the property.

- You can set the ICE_CONFIG environment variable to the path of a configuration file. The file is read when you create a communicator.

- Configuration files use UTF-8 encoding.

Setting Ice Properties on the Command Line

You can set Ice properties on the command line, e.g.:

`java Server --Ice.UDP.SndSize=65535 --Ice.Trace.Network`

- `--Ice.Trace.Network` is the same as
  `--Ice.Trace.Network=1`
- `--Ice.Trace.Network=` is the same as
  `--Ice.Trace.Network=''`

- The `--Ice.Config` property determines the path of a configuration file:
  `--Ice.Config=/opt/Ice/default.config`

- `--Ice.Config` overrides the setting of the `ICE_CONFIG` environment variable.

- If you set properties on the command line, and the same properties are set in a configuration file, the properties on the command line override the ones in the configuration file.

Ice Initialization

`Ice.Util.initialize` accepts an argument holder:

`Communicator initialize(StringSeqHolder args);`

The function scans the argument vector for any Ice-specific options and returns an argument vector with those options removed.

Example:

`java Server --Ice.Config=cfg --Ice.Trace.Network=3 -o f`

After calling `Ice.Util.initialize`, the cleaned-up vector contains:

`-o f`

You should parse the command line for your application *after* calling `Ice.Util.initialize`. That way, you do not need to write code to skip Ice-related command-line options.

If you want the program name to appear in trace and log messages, set `Ice.ProgramName` before initializing the communicator.

ZeroC

Reading Properties Programatically

You can access property values programmatically:

```
dictionary<string, string> PropertyDict;
local interface Properties {
    string getProperty(string key);
    string getPropertyWithDefault(string key,
                                  string value);
    int getPropertyAsInt(string key);
    int getPropertyAsIntWithDefault(string key,
                                    int value);

     PropertyDict getPropertiesForPrefix(string prefix);
     // ...
};

local interface Communicator {
    Properties getProperties();
    // ...
};
```

Using InitializationData

Ice.Util.initialize is overloaded:

```
static Communicator initialize();
static Communicator initialize(String[] args);
static Communicator initialize(StringSeqHolder ah);
static Communicator initialize(InitializationData id);
static Communicator initialize(String[] args, InitializationData id);
static Communicator initialize(StringSeqHolder ah, InitializationData id);


final class InitializationData implements Cloneable
{
    public InitializationData();
    public java.lang.Object clone();
    public Properties properties;
    public Logger logger;
    public Stats stats;
    public ThreadNotification threadHook;
    public ClassLoader classLoader;
    public Dispatcher dispatcher;
}
```

Command-Line Application Properties

To be able to set application-specific properties on the command line, you must initialize a property set before you initialize the communicator:

```
public static void main(String[] args)
{
    Ice.InitializationData initData = new Ice.InitializationData();
    initData.properties = Ice.Util.createProperties();
    args = initData.properties.parseCommandLineOptions(
                "Filesystem", args);

    // Parse other application-specific options here...

    Ice.Communicator communicator =
        Ice.Util.initialize(args, initData);
}
```

- `createProperties` creates an empty property set.

- `parseCommandLineOptions` converts properties with the specified prefix, strips them from `args`, and returns the remaining arguments.

ZeroC

Commonly-Used Ice Properties

- `Ice.Trace.Network` (0-3)

  Trace network activity.

- `Ice.Trace.Protocol` (0 or 1)

  Trace protocol messages.

- `Ice.Warn.Dispatch`

  Print warnings for unexpected server-side exceptions.

- `Ice.Warn.Connections` (0 or 1)

  Print warnings if connections are lost unexpectedly.

- `Ice.MessageSizeMax` (value in kB)

  Set maximum size of messages that can be sent and received.

- `Ice.ThreadPool.Server.Size`

  Set the number of threads in the server-side thread pool.

See the Ice manual for a complete list of properties.

Converting Properties to Proxies

A convenience operation on the communicator allows you to convert a property value to a proxy.

```
ObjectPrx p = comunicator.propertyToProxy("App.Proxy");
```

This reads the stringified proxy from the property App.Proxy.

App.Proxy is the base name of the property. You can define additional aspects of the proxy in separate subordinate properties. For example:

- App.Proxy.CollocationOptimized
- App.Proxy.ConnectionCached
- App.Proxy.EndpointSelection

The subordinate properties of the property group define the local behavior of the proxy, such as how to select endpoints, prefer secure transports over non-secure ones, and so on.

Object Adapter Properties

Object adapters support a number of configuration properties.

The adapter's name is used as the prefix for its properties:

```
ObjectAdapter adapter =
    communicator.createObjectAdapter("MyAdapter");
```

Commonly-used adapter properties:

- `MyAdapter.AdapterId`
- `MyAdapter.Endpoints`
- `MyAdapter.ProxyOptions`
- `MyAdapter.PublishedEndpoints`
- `MyAdapter.Router`
- `MyAdapter.ThreadPool`

Properties must be defined in the communicator's property set prior to calling `createObjectAdapter`.

Ice Programming with Java

9. Assignment 4
Using Properties

ZeroC

Exercise Overview

In this exercise, you will:

- modify the client we created in Assignment 2 to use application-specific properties.

By the end of this exercise, you will have gained experience in how to use properties to configure the Ice run time as well as your own applications.

Adding an Application-Specific Property

- In your `lab4` directory, you will find a `build.xml` file to build a client and a server.

- Both client and server are complete.

- You will modify the client to use application-specific properties.

What You Need to Do

1. Modify the client such that it picks up its property settings from a configuration file. Add the missing initialization of `base` to denote the proxy to the root directory.
2. Modify the `run` method such that it retrieves the property setting and sets the `_showSize` member variable accordingly.
3. Create a configuration file `config` and add a setting for both properties to it.
4. Modify `main` such that you can invoke the client.
5. Change the proxy for the root directory to port 9999 and run the client.
6. Change the proxy for the root directory to use port 10000 again. Now run the client with `--Ice.Trace.Protocol=1`.

ZeroC

Using Properties

The missing line of code to initialize the _showSize member is:

```
_showSize = communicator().getProperties().
        getPropertyAsInt("Filesystem.ShowSize") != 0;
```

The code to add at the beginning of main so the properties can be set on the command line is:

```
Ice.InitializationData initData = new Ice.InitializationData();
initData.properties = Ice.Util.createProperties();
args = initData.properties.parseCommandLineOptions("Filesystem",
args);
```

Ice Programming with Java

10. Multi-Threaded Ice

ZeroC

Lesson Overview

- This lesson presents:
  - the threading models available with the Ice run time and how to configure them.
  - some general threading strategies that you can use in your servers
- By the completion of the chapter, you will understand how the Ice run time uses threads and how to implement a simple thread-safe server.

Ice Threading Model

Ice uses a thread pool concurrency model.

For each communicator, Ice maintains:

- a client-side thread pool to process replies for outgoing requests and to dispatch incoming requests on bi-directional connections.

- a server-side thread pool to dispatch incoming requests.

You can create additional per-adapter thread pools.

The default size for both client- and server-side thread pools is 1.

Thread Pool Configuration

By default, the client- and server-side thread pools contain a single thread.

You can configure the pool size:

- `Ice.ThreadPool.Client.Size=<num>`

   The client-side thread pool can normally be left at 1, unless you need to support concurrent asynchronous or bi-directional callbacks (or if these callbacks might block).

- `Ice.ThreadPool.Server.Size=<num>`

   The server-side thread pool determines how many requests can be processed concurrently by the server.

Both properties set the initial number of threads in the pool.

Thread Pool Configuration (1)

Thread pools initially contains the number of threads specified by

`Ice.ThreadPool.Client.Size` and

`Ice.ThreadPool.Server.Size`.

You can also set a maximum size:

- `Ice.ThreadPool.Client.SizeMax=<num>`
- `Ice.ThreadPool.Server.SizeMax=<num>`

These properties allow a thread pool to temporarily grow larger than its initial size due to increased demand.

During idle periods, the size of a pool can shrink to just one thread. The idle timeout is specified by

`Ice.ThreadPool.Client.ThreadIdleTime` and

`Ice.ThreadPool.Server.ThreadIdleTime`.

Thread Pool Configuration (2)

- `Ice.ThreadPool.Client.SizeWarn=<num>`

  `Ice.ThreadPool.Server.SizeWarn=<num>`

  These properties log a warning once the number of threads in a pool exceeds the specified threshold.

- `Ice.ThreadPool.Client.StackSize=<bytes>`

  `Ice.ThreadPool.Server.StackSize=<bytes>`

  These properties set the stack size of the threads in a pool (byte units).

  The default value is zero, which gives threads the default stack size as determined by the OS.

ZeroC

Thread Safety

All APIs in the Ice run time are thread safe:

- You never have to lock something against concurrent access on behalf of the run time.

- Ice run-time APIs are deadlock free, so you can call any Ice API at any time and from any thread without fear of deadlock.

  Exception:

  Do not call `waitForShutdown`, `waitForDeactivate`, or `waitForHold` from within an executing operation on the corresponding adapter. If you do, you *will* deadlock.

Access to collections (sequences and dictionaries) is *not* interlocked. If you manipulate the same collection concurrently from different threads, you must establish a critical region yourself.

For multi-threaded servers, you must protect your own application-specific data against concurrent access.

Ice Programming with Java

11. Assignment 5
Thread Safety

Exercise Overview

In this exercise, you will:

- modify the server we created in Assignment 3 to be thread-safe.

By the completion of this exercise, you will have gained experience in how to use synchronization to make a server implementation thread-safe, and will know how to create threads to make concurrent invocations.

Thread Safety

- The file system server is not thread-safe.

- Modify the server to support concurrent invocations by clients and modify the client to make concurrent invocations on the server.

ZeroC

What You Need to Do

1. Modify the server to provide mutual exclusion.

2. Add trace statements to the beginning and end of `list`: your code should print the calling thread's ID as it enters and leaves `list`.

3. For testing purposes, add a statement to `list` that causes the calling thread to sleep for one second.

4. Modify the client to create three threads, each of which will call `listRecursive`.

5. At the end of `Client.run`, add code to create three threads of type `ListThread`.

6. Add code to join with the three threads you created in step 5.

7. Run client and server in separate windows and examine the trace produced by each program.

ZeroC

Server Modifications

- To make the server thread-safe, we need to make Slice operations synchronized.

- The operations for which this is necessary are `read`, `write` and `list`.

- It is also necessary to acquire a lock in `addChild:` without this lock, if the server concurrently instantiates nodes from different threads, the `_contents` member of the parent can be corrupted.

Client Modifications

- The client simply creates three threads that each call `listRecursive` and joins with these threads.

ZeroC

Ice Programming with Java

12. Object Life Cycle

ZeroC

Lesson Overview

- Object life cycle refers to the issues that surround creation and destruction of objects.

- This lesson shows you how you can create and destroy Ice objects in response to client requests, and how to ensure that these operations are thread-safe. The lesson also discusses issues regarding the uniqueness of object identities, and how to deal with objects that are abandoned by clients.

- By the completion of this lesson, you will have a thorough understanding of how to provide life cycle operations in a thread-safe manner.

Object Creation

Object creation typically relies on the factory pattern:

```
exception NameInUse {};

interface Directory extends Node {
    Directory* createDir(string name)
        throws NameInUse;
};
```

- The factory operation creates a new Ice object as a side-effect and returns the proxy to the newly-created object.

- As far as the Ice run time is concerned, a factory operation is no different from any other operation.

- The factory operation behaves like a constructor and can accept whatever arguments are necessary to create the new object.

- Often, factory operations also throw exceptions to indicate errors that might be caused by invalid arguments or that are detected by the operation implementation.

Object Creation and Thread Safety

If clients can call `create` concurrently, you must interlock:

```
public synchronized FilePrx
createFile(String name, Ice.Current c)
    throws NameInUse
{
    if(!nameIsValid(name))
    {
        throw NameInUse();
    }

    // Instantiate servant, add to ASM,
    // and return proxy here...
}
```

ZeroC

The Current Object

Every operation invocation is passed an object of type `Ice::Current`:

```
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;
};
```

The `Current` object provides information about the current invocation.

Object Destruction

To destroy an object, add an operation that instructs the object to commit suicide:

```
exception DirNotEmpty {};

interface Node {
    void destroy() throws DirNotEmpty;
};
```

- The implementation of `destroy` removes the servant from the ASM and destroys whatever resources are held by the servant.

- Clients invoking on the proxy for the destroyed object receive `ObjectNotExistException`.

- As far as the Ice run time is concerned, `destroy` is an ordinary operation without special significance.

- Do not add `destroy` to the factory. If you do, you need to keep track of which factory created what object.

Implementing Object Destruction

The object adapter provides a `remove` operation that removes an entry from the ASM:

```
local interface ObjectAdapter {
    Object remove(Identity id);
    // ...
};
```

`remove` breaks the link between the object identity and the servant, effectively destroying the Ice object.

- The operation returns the servant that was removed.

- Calling `remove` on an object identity that is not in the ASM raises `NotRegisteredException`.

- If the server code does not hold a reference to the servant elsewhere, the servant becomes eligible for garbage collection as soon as the last executing operation leaves the servant.

Object Destruction and Thread Safety

You must avoid a race condition if `destroy` can be called concurrently:

```java
public synchronized void
destroy(Ice.Current c)
{
    if(_destroyed)
        throw new Ice.ObjectNotExistException();
    // Remove any servant-specific state here...
    c.adapter.remove(c.id);
    _destroyed = true;
}

public synchronized void
write(String[] text, Ice.Current c) throws IOError
{
    if(_destroyed)
        throw new Ice.ObjectNotExistException();
    // ...
}
```

When to Remove Servant State?

Avoid removing servant state in the servant's finalizer:

- `destroy` often must perform clean-up that can fail, such as closing network connections or flushing files.

  If you delay physical removal of servant resources until the finalizer runs and something goes wrong, you end up with inconsistent state: `destroy` has completed successfully, but physical servant state is still there!

- If anything goes wrong in the finalizer, the finalizer cannot throw exceptions. (The best it can do is log the error.)

Object Identity and Uniqueness

The Ice object model assumes that Ice objects have globally-unique identities.

- If you use UUIDs as object identities, this is guaranteed to be the case.

- If you use application-specific data as object identities, this is not guaranteed—the application must enforce sufficient uniqueness.

Technically, object identities must be unique per object adapter.

Object identity is embedded in the proxy for an object and sent over the wire with each invocation.

If object identities are globally unique, `ObjectNotExistException` is reliable:

- Once a client receives `ObjectNotExistException` from an object, all future attempts to contact the object will either fail, or also raise `ObjectNotExistException`.

Object Identity and Uniqueness (1)

```
interface File {
    void destroy();
    // ...
};

interface FileFactory {
    File* create(string pathname);
};
```

Assume that the path name is used as the object identity. A client can now do:

```
FileFactoryPrx ff = ...;
FilePrx f = ff.create("/fred");
// Write to new file...
// Pass f to some other process...

// Later:
f.destroy();
f = ff.create("/fred");
// Write to new file...
```

ZeroC

Object Identity and Uniqueness (2)

```
FileFactoryPrx ff = ...;
FilePrx f = ff.create("/fred");

// Pass proxy to some other process...

// Later...

f.destroy();

// Still later...

// Use same identity for different type of object:
ThingFactoryPrx tf = ...;
ThingPrx t = tf.create("/fred");
```

If a client invokes on the `File` proxy after the object is reincarnated as a `Thing`, it may get an `OperationNotExistException`, `MarshalException`, or even undefined behavior!

Uniqueness Recommendations

Consider using UUIDs as object identity. UUIDs are convenient because they make name clashes impossible.

If you use application-assigned object identity, pay attention to uniqueness:

- Ideally, do not ever re-use an identity.
- If you re-use identities, write your application to cope with this:
    - Avoid storing proxies in clients beyond their "use-by date."
    - Do not build semantics into your application that expect `ObjectNotExistException` to be a definitive death certificate.
    - Use separate namespaces for object identities for different object types (for single object adapters), or
    - Use different object adapters for different types of objects.
- If you want to use IceGrid's well-known objects, you *must* use an identity that is unique within the IceGrid domain.

ZeroC

Dealing with Stale Objects

Consider stateful client–server interactions, such as for an online shop:

- The client creates a shopping cart object via a factory.

- Purchases are added to the cart by invoking operations on the cart.

- When the client is finished, and presses the "Buy" button, the order is processed and the cart is destroyed.

What happens if the client never finishes the purchasing process or crashes?

The server holds onto resources on behalf of the client so, unless the server does something in this case, it will eventually run out of resources.

Dealing with Stale Objects (1)

Basic approach for cleaning up stale objects:

- Instead of creating objects directly, each client creates a single session object.

- The session object is the object factory that allows the client to create all the objects it needs.

- The session keeps track of which objects were created.

- The session offers a `refresh` operation. The client is expected to call refresh every *n* seconds.

- If the client fails to call `refresh` for more than *n* seconds, the server destroys the session object, and all objects created by that session.

This approach guarantees:

- resources will be reclaimed if a client crashes

- resources are not reclaimed prematurely (while still being used)

Dealing with Stale Objects (2)

```
interface SomeObject {
    // Lots of operations here...

    void destroy();
};

interface Session { // One session per client
    SomeObject* create(/* params */);
    idempotent string getName();
    void refresh();
    idempotent void destroy();
};
interface SessionFactory { // Singleton
    Session* create(string name);
};
```

Dealing with Stale Objects (3)

The reaper thread:

- maintains a list of existing sessions

- provides an `add` operation so sessions can be added

- runs an infinite loop:

    - sleep for n seconds

    - get the current time

    - for each existing session, if the session's timestamp is older than n seconds, call `destroy` on the session and remove the session from the list

    - if a call to `destroy` raises `ObjectNotExistException`, remove the session from the list

Dealing with Stale Objects (4)

```
class SessionFactoryI extends _SessionFactoryDisp
{
    public SessionFactoryI(ReapThread r)
    {
        _reaper = r;
    }
    public SessionPrx create(String name, Ice.Current c)
    {
        SessionI session = new SessionI();
        SessionPrx proxy = SessionPrxHelper.uncheckedCast(
                                c.adapter.addWithUUID(session));
        _reaper.add(proxy, session);
        return proxy;
    }
    private ReapThread _reaper;
}
```

```java
class SessionI extends _SessionDisp
{
    public synchronized SomeObjectPrx
    create(Ice.Current c);

    public synchronized void
    destroy(Ice.Current c);

    public void
    refresh(Ice.Current c);

    public long
    timestamp();

    private long _timestamp = System.currentTimeMillis();
    private boolean _destroyed = false;
    private java.util.LinkedList<SomeObjectPrx> _objs =
                new java.util.LinkedList<SomeObjectPrx>();
}
```

Dealing with Stale Objects (6)

The server's main program:

- instantiates an object adapter

- creates the reaper thread and starts it

- creates the session factory and adds it to the ASM

- activates the object adapter

- waits for shut-down

Once shut-down is complete, the server:

- calls `terminate` on the reaper thread

- joins with the reaper thread

Dealing with Stale Objects (7)

The client must call `refresh` every *n* seconds to keep the session alive.

Instead of arbitrarily sprinkling calls to `refresh` through the code, in the hope that they get executed often enough, run a background thread:

- The refresh thread sits in a loop and calls `refresh` periodically. To be safe, make the refresh interval a little bit shorter than the server's reap interval.

- The refresh thread provides a `terminate` method so the client's main thread can join with it when the time comes to shut down.

Ice Programming with Java

13. Assignment 6
Object Life Cycle

Exercise Overview

In this exercise, you will:

- add life cycle operations to the file system server.

By the completion of this exercise, you will have gained experience in how to implement thread-safe life cycle operations.

ZeroC

Life Cycle

- In this exercise, you will add life cycle operations to the file system server.

- The server is the thread-safe server you developed in Assignment 5, so your implementation will need to be thread-safe.

What You Need to Do

1. Examine the code in `Server.java`.
2. Look at the implementations of `makeRootDir` and the `DirectoryI` constructors in `Filesystem/DirectoryI`.
3. Implement the `createDir` method.
4. Implement the `createFile` method.
5. Use the test client that is contained in `Client.java` to test your create operations.
6. Implement the `DirectoryI.destroy` method.
7. Implement the `FileI.destroy` method.
8. Edit `Client.java` and enable the commented-out section marked `PART_2`.

Thread Safety Modifications

- To prevent a race condition, we need to test, on entry to every operation, whether the object has been previously destroyed. Rather than repeat the same code in every operation, we can bundle the test into a helper function.

`createDir` Implementation

- `createDir` calls `checkDestroyed` in case this directory has been destroyed before the operation body started to execute and then calls `checkNameInUse` to make sure that clients cannot create two directories with the same name.

createFile Implementation

- The `createFile` implementation is analogous to `createDir`.
- The `FileI` constructor does much of the work

`DirectoryI.destroy` Implementation

```
public void
destroy(Ice.Current c)
{
    synchronized(this)
    {
        checkDestroyed();
        _destroyed = true;
    }
    c.adapter.remove(c.id);
    _parent.removeChild(_myID);
}
```

FileI.destroy Implementation

The `FileI.destroy` implementation is analogous to `DirectoryI.destroy`:

```
public void
destroy(Ice.Current c)
{
    synchronized(this)
    {
        checkDestroyed();
        _destroyed = true;
    }
    c.adapter.remove(c.id);
    _parent.removeChild(_myID);
}
```

ZeroC

Ice Programming with Java

14. Glacier2

Lesson Overview

- Glacier2 is the Ice firewall traversal service. It allows clients and servers to communicate even if they are separated by a firewall.

- By the completion of this lesson, you will understand how Glacier2 works, how to configure it correctly, and how to modify your applications to work with Glacier2.

Running an Ice Server Behind a Firewall

If a server is behind a firewall, clients can access the server if:

- The firewall opens an incoming port for clients.
- The firewall port-forwards incoming connections on that port to the real server port.
- The server is configured to advertise the firewall's host name and port in its proxies instead of its own name and port by setting the `<adapter-name>.PublishedEndpoints` property.

Problems of this approach:

- Each server requires a separate hole in the firewall.
- If clients need to connect to the server from the inside network as well as the outside network, either:
  - traffic is routed from the inside network to the firewall and back into the inside network again (inefficient), or
  - the server must publish internal and external addresses in its proxies.

Glacier2

Glacier2 is the Ice firewall-traversal service. It provides:

- firewall traversal for servers with no change to code or configuration
- firewall traversal for clients with minimal code and configuration changes
- callbacks from servers to clients via a bidirectional connection (with minimal changes)
- authentication via user name and password (among others)
- session management
- secure communication via SSL
- request batching and filtering

Glacier2 requires only a single port to be opened in the firewall to support an arbitrary number of clients and servers.

Alternatively, Glacier2 can also *be* the firewall for Ice servers. (No port forwarding is required in this case.)
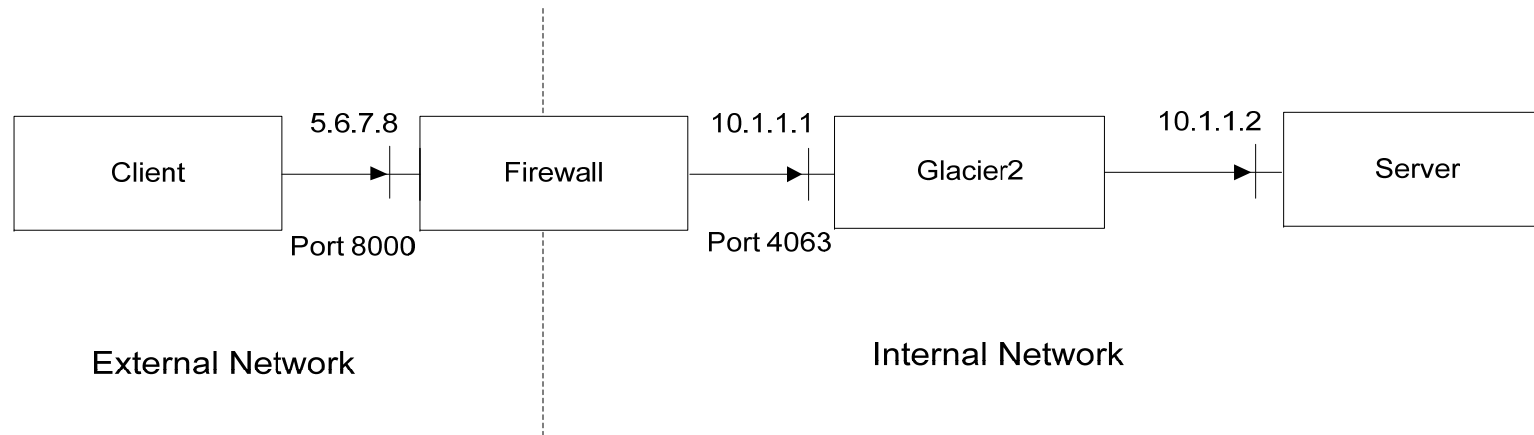
Glacier2 as a Firewall

Glacier2 acting as an Ice firewall, running on a machine with external and internal interfaces:
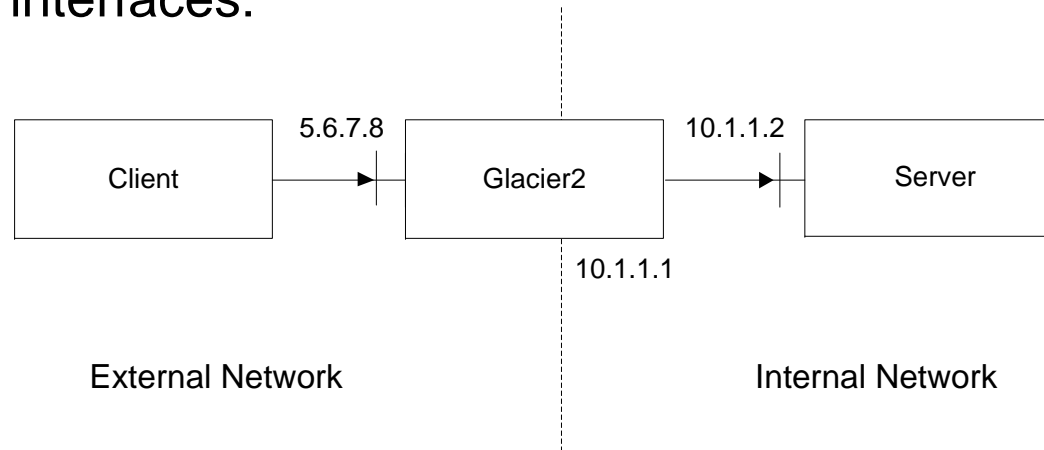
- Clients on the external network connect to Glacier2's external interface, instead of directly connecting to the server.
- Glacier2 forwards the request to the server on the internal network.
- Glacier2 receives the server's reply on the internal interface and forwards the reply to the client via the external interface.

Glacier2 Behind a Firewall

Glacier2 with a single internal interface running behind a firewall:

	5.6.7.8		10.1.1.1		10.1.1.2	
Client		Firewall		Glacier2		Server
	Port 8000		Port 4063			

External Network

Internal Network

- Firewall is configured to port-forward 5.6.7.8:8000 to 10.1.1.1:4063.

- Client connects to firewall, which forwards traffic to Glacier2 on the internal network.

- Glacier2 forwards the request to the server.

- Glacier2 receives the server's reply and forwards the reply to the firewall.

- The firewall forwards the reply to the client.

Running Glacier2

Glacier2's executable is called `glacier2router`.

UNIX daemon options:

- `--daemon`
- `--noclose`
- `--nochdir`

Use the `iceserviceinstall` utility to configure it as a Windows service.

ZeroC

Glacier2 Configuration

To configure Glacier2 to be usable by clients, you must minimally set the `Glacier2.Client.Endpoints` property.

It specifies the endpoint at which Glacier2 listens for incoming client requests, for example:
`Glacier2.Client.Endpoints=tcp -p 4063`

- If Glacier2 cannot be accessed by hostile clients, TCP is usually the appropriate protocol.

- If Glacier2 should allow access only for specific clients or if you require secure communcations, you should specify SSL as the protocol.
  If you specify SSL, you must also configure the IceSSL plugin by setting:
  `Ice.Plugin.IceSSL=IceSSL:createIceSSL`
  `IceSSL.DefaultDir=...`
  `IceSSL.CertAuthFile=...`
  `IceSSL.CertFile=...`
  `IceSSL.KeyFile=...`

Glacier2 Sessions

For each client, Glacier2 maintains a session. Clients must obtain a Router proxy and use it to create a session:

```
module Glacier2 {
    exception CannotCreateSessionException {
        string reason;
    };
    exception PermissionDeniedException {
        string reason;
    };
    interface Session {
        void destroy();
    };
    interface Router extends Ice::Router {
        // ...
        Session* createSession(string userId,
                               string password)
            throws PermissionDeniedException,
                   CannotCreateSessionException;
    };
};
```

Client Configuration

You must set two properties for the client to use Glacier2:

- `Ice.Default.Router=Glacier2/router:tcp \`
  `-h 5.6.7.8 -p 4063`
- `Ice.ACM.Client=0`

`Ice.Default.Router` specifies which Glacier2 router the client will use. The endpoint details must match the configuration of Glacier2.

`Ice.ACM.Client` must be disabled by explicitly setting it to zero (or set to a value larger than Glacier2's session timeout). (ACM is enabled by default.)

You should also disable retries by setting:

- `Ice.RetryIntervals=-1`

ZeroC

Creating a Password File

Glacier2 uses a password file to authenticate clients.

The password file contains one line for each user, with the user name and encrypted password:

```
joe ZpYd5t1p4.dOY
marc G8YOZ67QgnIwI
```

You must configure the name and location of the password file by setting `Glacier2.CryptPasswords` to the path name of the file.

You can use the `openssl` utility to create encrypted passwords:

```
$ openssl
OpenSSL> passwd
Password: openSesame
Verifying - Password: openSesame
ZpYd5t1p4.dOY
```

Custom Authentication

You can implement a custom authentication mechanism by implementing the `PermissionsVerifier` interface:

```
module Glacier2 {
    interface PermissionsVerifier
    {
        idempotent bool checkPermissions(
                        string userId,
                        string password,
                        out string reason);
    };
};
```

Set `Glacier2.PermissionsVerifier` to the proxy of this object.

If set, Glacier2 uses the specified verifier instead of the default password mechanism.

`checkPermissions` must return true if authentication is successful, false otherwise.

The Admin Interface

The Admin interface allows remote shut-down of Glacier2:

```
module Glacier2 {
    interface Admin
    {
        void shutdown();
    };
};
```

The default identity of this interface is `Glacier2/admin`.

The endpoint at which this object listens is determined by the property `Glacier2.Admin.Endpoints`.

If the property is not set, Glacier2 does not enable this interface.

Do not make this object available on a public network or, if you do, only use an SSL endpoint!

Custom Object Identities

If you are running several Glacier2 processes, you will need to use different object identities for each one.

- `Glacier2.InstanceName`

This property changes the identity of the `Router` and `Admin` objects in Glacier2. For example:

- `Glacier2.InstanceName=Fred`

results in `Fred/router` and `Fred/admin` as the identities of the router and admin objects.

If you change the identity, the client configuration must be changed accordingly:

- `Ice.Default.Router=Fred/router:tcp -h 5.6.7.8 -p 4063`

Session Timeouts

Unless you configure a timeout, session state is maintained by Glacier2 indefinitely.

To configure a timeout, set the property `Glacier2.SessionTimeout` to the timeout value in seconds.

Any client activity resets the timeout. If there is no activity on the session for the specified timeout, Glacier 2 destroys the session.

If a session is destroyed, the client must create a new session, reauthenticating itself.

A destroyed session results in a `ConnectionLostException` in the client.

Explicit Session Management

If you need to track session activity of clients, you can create an external session:

- Implement the Glacier2 `SessionManager` interface
- Configure Glacier2 to use your session manager by setting `Glacier2.SessionManager` to the session manager's proxy.
- Implement the Glacier2 `Session` interface.

If you use explicit session management, your `create` operation can return the session's proxy to the client. In that case, the client receives a non-null proxy (instead of the null proxy it gets for an internal session).

Explicit session management is useful to, for example, log when clients create and destroy a session.

Your `create` operation must handle re-creating a dropped session.

Note that, for SSL, there is also an `SSLSessionManager`.

ZeroC

Supporting Callbacks
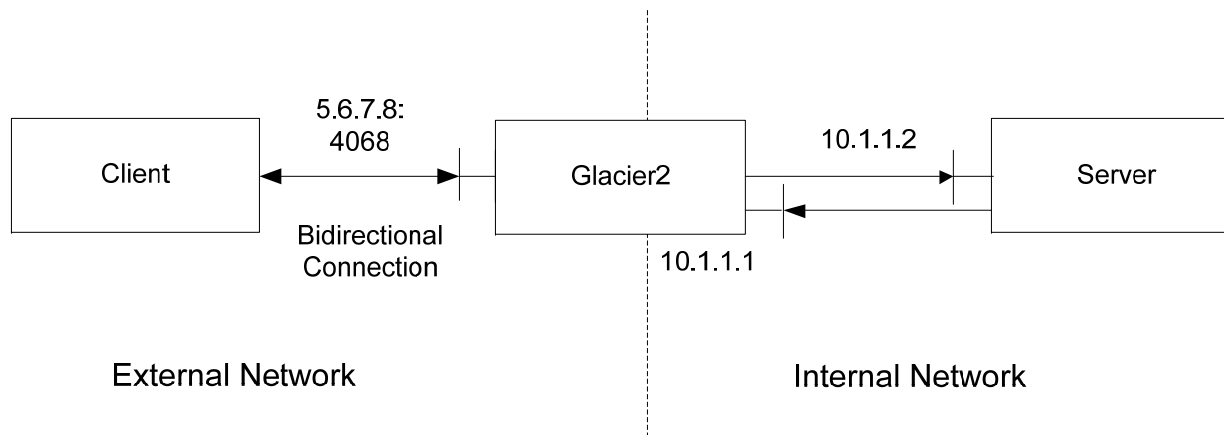
To support callbacks from server to client, Glacier2 must have an endpoint in the internal network.

The `Glacier2.Server.Endpoints` property configures that endpoint. The property does not require a port, only a host name or

IP address:

    `Glacier2.Server.Endpoints=tcp –h 10.1.1.1`

No code changes are required in the server for callbacks.

Supporting Callbacks (1)

Client requirements for callbacks:

- The client must have an object adapter (but no local endpoint is necessary).

- Callback proxies created by this object adapter must use the *router's* server endpoint so that callback invocations from back-end servers are sent to the router, and not sent directly to the client.

- To achieve this, the client must configure its callback object adapter with a proxy for the router, using the *<adapter-name>.*`Router` property or by calling `createObjectAdapterWithRouter`.

ZeroC

Supporting Callbacks (2)

When a server makes a callback, Glacier2 has to work out which client the callback should go to.

For each client session, Glacier2 generates a unique category. That category *must* be used by a client in the identity of its callback objects.

```
Ice.RouterPrx r = communicator.getDefaultRouter();
Glacier2.RouterPrx router =
    Glacier2.RouterPrxHelper.checkedCast(r);
String category = router.getCategoryForClient();

Ice.Identity id = new Ice.Identity();
id.category = category;
id.name = java.util.UUID.randomUUID().toString();
SomeObject p = new SomeObjectI();
adapter.add(p, id);
```

Because each client uses a different category, Glacier2 can examine the category to determine to which client to forward a callback made by the server.

Helper Classes

Ice includes helper classes that provide functionality commonly needed by Glacier2 clients:

- `Glacier2.Application` is a subclass of `Ice.Application` that simplifies the use of Glacier2 in command-line applications.

- `Glacier2.SessionFactoryHelper` and `Glacier2.SessionHelper` offer greater flexibility for graphical clients.

Glacier2.Application

The `Glacier2.Application` class should look familiar to users of `Ice.Application`:

```
package Glacier2;
public abstract class Application extends Ice.Application {
    public class RestartSessionException extends Exception { }
    public Application();
    public Application(SignalPolicy signalPolicy);
    public abstract Glacier2.SessionPrx createSession();
    public abstract int runWithSession(String[] args)
        throws RestartSessionException;
    public static Glacier2.RouterPrx router();
    public static Glacier2.SessionPrx session();
    // ...
}
```

Subclasses must implement `createSession` and `runWithSession`.

Glacier2.Application (1)

Additional convenience methods simplify callbacks and session management:

```
package Glacier2;
public abstract class Application extends Ice.Application {
    // ...
    public void sessionDestroyed();
    public void restart()
        throws RestartSessionException;
    public String categoryForClient()
        throws SessionNotExistException;
    public Ice.Identity createCallbackIdentity(String name);
    public Ice.ObjectPrx addWithUUID(Ice.Object servant);
    public Ice.ObjectAdapter objectAdapter()
        throws SessionNotExistException;
}
```

ZeroC

Ice Programming with Java

15. Assignment 7
Using Glacier2

Exercise Overview

In this exercise, you will

- modify the file system application to work with Glacier2.

By the end of this exercise, you will have gained experience in how to configure Glacier2 and your applications, create Glacier2 sessions, and communicate via Glacier2.

Using Glacier2

- In this exercise, you will modify the application you developed in Assignment 6 to communicate via Glacier2.

What You Need to Do

1. The client needs to communicate with the server via Glacier2. Change the client to use `Glacier2.Application` and add the missing code to create a Glacier2 session for the client.

2. Create a configuration file for Glacier2. For this exercise, because we do not have a real firewall, you will run the client, server, and Glacier2 on the same machine. Use the loopback address (127.0.0.1) for the configuration. Glacier2 should listen for client requests on port 4063. Configure a session timeout of 30 seconds.

3. Create a password file for Glacier2 and modify the client source code to use the correct user name and password.

4. Create a configuration file for the client to work with Glacier2.

5. Run Glacier2, the client, and the server. If you have set things up correctly, the client will list the contents of the server's file system.

6. Run Glacier2, the client, and the server with network tracing enabled. Examine the port numbers that are used to convince yourself that the client indeed communicates via Glacier2.

7. Change the client to use an invalid password and verify that Glacier2 correctly rejects session creation.

Client Modifications

Client Configuration

```
Ice.Default.Router=Glacier2/router:tcp -h 127.0.0.1 -p 4063
Ice.ACM.Client=0
Ice.RetryIntervals=-1
```

Glacier2 Configuration

```
Glacier2.Client.Endpoints=tcp -h 127.0.0.1 -p 4063
Glacier2.CryptPasswords=passwords
Glacier2.SessionTimeout=30
```

Glacier2 Password File

You need one line in the password file with a user name and encrypted password, for example:

```
joe OWULk8FE9fmwo
```

The encrypted password in this file is "joe".

Ice Programming with Java

16. The IceGrid

Lesson Overview

- IceGrid is the location and server activation service for Ice.

- In this lesson you will learn to:
  - use IceGrid to start servers on demand
  - avoid hard-coding addresses and port numbers into proxies
  - advertise application objects
  - monitor the status of servers.

- By the completion of this lesson, you will understand how IceGrid works, how to configure clients and servers to take advantage of indirect binding and automatic activation, and how to administer and troubleshoot IceGrid.

ZeroC

IceGrid

IceGrid is a location and activation service:

- IceGrid allows clients to use indirect proxies that do not contain host names (or IP addresses) and port numbers.

- IceGrid can activate servers on demand, when clients first issue a request.

- IceGrid allows well-known proxies to be advertised. Clients can bootstrap using proxies that contain the name of well-known objects, instead of endpoint information.

IceGrid also provides advanced features:

- Replication and load balancing with automatic failover

- Allocation of servers to clients

- Status monitoring

- Application distribution

- Centralized application deployment

IceGrid Components

IceGrid consists of a single registry and one or more nodes:

- The IceGrid registry is a database that keeps track of known applications and the servers that make up each application.

  The registry also knows details such as how to start each server, what command-line options to provide at server start-up, and the values of environment variables to be set for each server.

  A single registry is used for a number of machines.

- An IceGrid node is essentially a server start-up and monitoring process.

  On each machine on which IceGrid-aware servers run, an IceGrid node process is required.

  Each IceGrid node communicates with its IceGrid registry to keep it informed of the status of servers.

ZeroC

IceGrid Architecture

A simple IceGrid architecture:

Registry	Node	Node
	Server A	Server B
		Server C
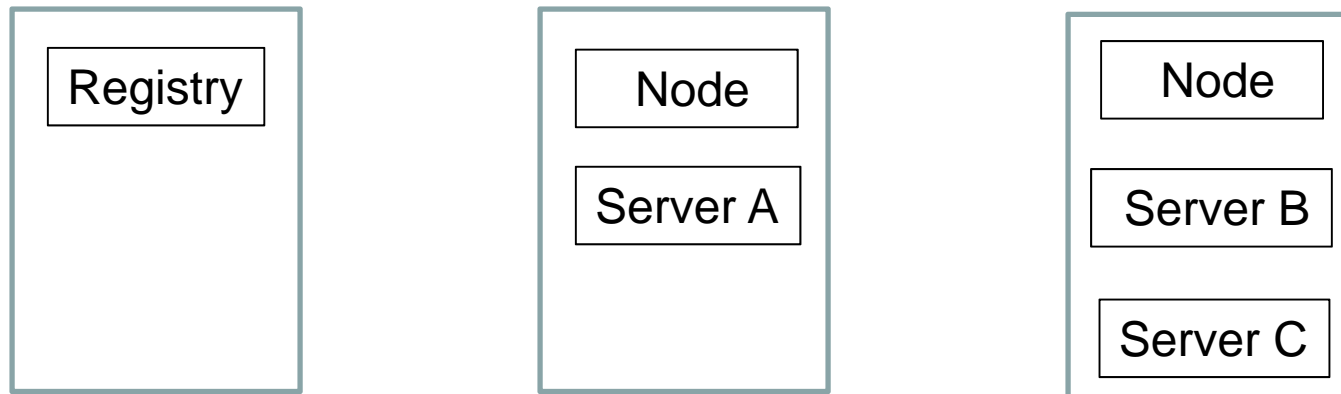
In this example, the machine running the registry does not run servers or a node.

More commonly, the machine running the registry also runs a node and servers.

If the machine running the registry also runs a node, you can (but need not) collocate the registry and node into a single process by setting `IceGrid.Node.CollocateRegistry=1`.

ZeroC

Indirect Proxies

An indirect proxy has the form

<object-identity>@<adapter-id>

For example:

RootDir@fsadapter

The adapter *ID* is different from the adapter *name* that is used by the server. The adapter ID is configured with the adapter property *<adapter-name>*.AdapterId.

The advantage of indirect proxies is that they do not contain endpoint information. If a server is moved to a different machine or port, the client need not be updated.

Client Configuration

To work with IceGrid, clients require only a single configuration item:

`Ice.Default.Locator` must be set to the proxy of the IceGrid registry:

```
Ice.Default.Locator=IceGrid/Locator:tcp -h registryhost \
-p 4061
```
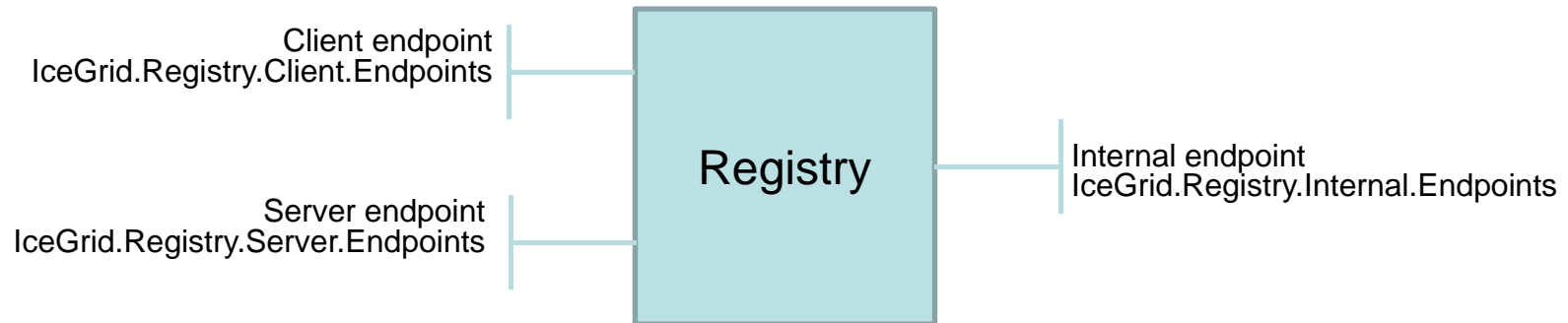
This property tells the client-side run time where it can obtain endpoint information for indirect proxies.

`IceGrid/Locator` is the default identity of the registry's locator service.

Note that the proxy for the locator cannot be an indirect proxy: the run time requires one fixed endpoint at which it can resolve addresses.

ZeroC

Registry Endpoints

The registry provides three endpoints:


```
Client endpoint
IceGrid.Registry.Client.Endpoints
```
```
Registry
```
```
Internal endpoint
IceGrid.Registry.Internal.Endpoints
```
```
Server endpoint
IceGrid.Registry.Server.Endpoints
```

- Client endpoint
  Used by the administrative tools and by clients to resolve indirect proxies

- Server endpoint

  Used by servers for status updates and registration

- Internal endpoint

  Used by nodes and registry replicas

Two additional endpoints are used if IceGrid runs in conjunction with Glacier2.

Registry Configuration

You must set the following properties for the registry to work:

- `IceGrid.Registry.Client.Endpoints`

- `IceGrid.Registry.Server.Endpoints`

- `IceGrid.Registry.Internal.Endpoints`

- `IceGrid.Registry.Data`

Only the client endpoint requires a port number.

The server and internal endpoints only require a protocol, but not a host or port.

The `IceGrid.Registry.Data` property defines the path to a directory in which the registry places its database files.

ZeroC

Node Configuration

Each node requires at least the following configuration:

- `Ice.Default.Locator`

  Defines the registry's location service proxy.

- `IceGrid.Node.Endpoints`

  Defines the node's endpoint for communication with the registry.

- `IceGrid.Node.Name`

  A unique name for the node within the IceGrid domain.

- `IceGrid.Node.Data`

  The location of the configuration files of servers started by the node.

`IceGrid.Node.Name` must be different for each node!

If you want to collocate the registry, you can set `IceGrid.Node.CollocateRegistry=1` on exactly one of the nodes in the IceGrid domain.

Starting an IceGrid Node

--nowarn: Don't print security warnings.

--readonly: Start the master registry in read-only mode.

UNIX:

--daemon: Run as UNIX daemon

--noclose: Don't close open file descriptors for daemon

--nochdir: Don't change directory to /

--pidfile *file*: Write process ID into specified file.

Windows:
Use the **iceserviceinstall** utility to configure it as a Windows service.

Starting the Registry

The registry command is `icegridregistry`.

It supports the `--nowarn` option as well as the same UNIX daemon options as `icegridnode`:

    --daemon, --noclose, --nochdir, --pidfile *file*

Use the `iceserviceinstall` utility to configure it as a Windows service.

If you run a separate registry, and start nodes before starting the registry, the nodes will periodically attempt to contact the registry and establish a connection once the registry is running.

ZeroC

Server Configuration

Server configuration is accomplished via XML files.

The XML descriptor at a minimum describes:

- the application name
- the node(s) on which the server(s) run
- for each server:
    - a server ID
    - the server executable file name

Additional descriptor elements can specify:

- the adapter name and protocol
- activation mode (manual, on-demand, etc.)
- command-line options
- property settings
- environment variables

Server Configuration (1)

A server element can have several `option` child elements:

```
<icegrid>
  <application name="filesystem">
    <node name="Node1">
      <server id="fsserver" exe="/usr/bin/fsserver">
        <option>--myoption</option>
        <option>myoptarg</option>
        <adapter name="Lab8" endpoints="tcp"/>
      </server>
    </node>
  </application>
</icegrid>
```

Command-line arguments are appended to the executable in the order specified.

Server Configuration (2)

You can set properties as part of a server descriptor:

```
<icegrid>
  <application name="filesystem">
    <node name="Node1">
      <server id="fsserver" exe="/usr/bin/fsserver">
        <option>Server</option>
        <property name="Ice.ServerIdleTime" value="20"/>
        <property name="Ice.GC.Interval" value="60"/>
        <adapter name="Lab8" endpoints="tcp"/>
      </server>
    </node>
  </application>
</icegrid>
```

Property settings are written into a configuration file that is passed to the server on start-up.

Server Configuration (3)

You can set environment variables for the server:

```
<icegrid>
  <application name="filesystem">
    <node name="Node1">
      <server id="fsserver" exe="/opt/app1/bin/server">
        <env>LD_LIBRARY_PATH="/opt/app1/lib"</env>
        <adapter name="App1" endpoints="tcp"/>
      </server>
    </node>
  </application>
</icegrid>
```

For UNIX, use Bourne shell syntax for environment variables.

For Windows, use Windows syntax:

```
<env>PATH=%PATH%;C:/opt/Ice/lib</env>
```

`$PATH` (and `$${PATH}`) substitute the setting of an environment variable.

Server Configuration (4)

Each `server` element must have an `adapter` element for each adapter to which clients bind indirectly.

```
<icegrid>
  <application name="filesystem">
    <node name="Node1">
      <server id="fsserver" exe="/usr/bin/fsserver">
        <adapter name="Lab8" endpoints="tcp"/>
      </server>
    </node>
  </application>
</icegrid>
```

The `adapter` element must minimally specify the adapter name (as used by the server).

The endpoint usually only specifies a protocol, so the operating system can assign a port. However, you can specify a port as well, if you want the server to use a specific port.

ZeroC

Server Configuration (5)

If you specify an id attribute, the server's externally visible adapter ID becomes that ID:

```
<icegrid>
  <application name="filesystem">
    <node name="Node1">
      <server id="fsserver" exe="/opt/app1/server">
        <adapter name="Lab8" id="fsa" endpoints="tcp"/>
      </server>
    </node>
  </application>
</icegrid>
```

The client's indirect proxy now becomes:

```
RootDir@fsa
```

ZeroC

Command-Line Administration

You can maintain the registry from the command line with `icegridadmin`.

The program requires the property `Ice.Default.Locator` to be set so it can find the registry.

`icegridadmin` allows you to:

- Add, update, and remove applications
- Start, stop, and check the status of servers
- Add, remove, and check the status of adapters
- Add, remove, and list well-known objects

`icegridadmin` Application Commands

- `application add` *file.xml*

  Add the application described in *file.xml*.

- `application remove` *name*

  Remove the application name.

- `application update` *file.xml*

  Update an already deployed application with *file.xml*.

- `application describe` *name*

  List details of application name.

- `application list`

  List all deployed applications.

- `application diff` *file.xml*

  Show differences between deployed application descriptor and *file.xml*.

icegridadmin Node Commands

- `node list`

  List all nodes.

- `node describe` *name*

  Show information about node *name*.

- `node ping` *name*

  Test whether node *name* is running.

- `node show` *name* `[stdout | stderr]`

  Show the node's `stdout` and/or `stderr` output.

- `node load` *name*

  Show the load of node *name*.

- `node shutdown` *name*

  Shut down the node *name*.

`icegridadmin` Server Commands

- `server list`

  List all server IDs.

- `server describe` *id*

  Show details of server *id*.

- `server enable` *id*

  Enable server *id*.

- `server disable` *id*

  Disable server *id*. (A disabled server cannot be started, either on demand or explicitly.)

- `server stdout` *id message*

  Write *message* on server *id's* standard output.

- `server stderr` *id message*

  Write *message* on server *id's* standard error.

icegridadmin Server Commands (1)

- server state *id*

  Show the state of the server *id* (running, inactive, enabled or disabled).

- server pid *id*

  Show the process ID of server *id*.

- server signal *id signal*

  Send signal *signal* to server *id* (UNIX only).

- server start *id*

  Start server *id*.

- server stop *id*

  Stop server *id*.

- server remove *id*

  Remove server *id*.

`icegridadmin` Server Commands (2)

- `server show [options]` *`id`* `[stdout | stderr | log ]`

  Print text from the server's `stdout, stderr`, or specified log file.

- `server properties` *`id`*

  Show the run-time properties of the server *id*.

- `server property` *`id name`*

  Show the setting of the property *name* for the server *id*.

- `server patch` *`id`*

  Apply updates to the server *id* via IcePatch2.

ZeroC

Server Activation and Deactivation

The `activation` attribute of a server element can be set to "manual", "on-demand", "session", or "always". (The default is "manual".)

`<server id="fsserver" exe="java" activation="on-demand">`

- If set to "manual", the server must be started using the `icegridadmin server start` command.

- If set to "on-demand", IceGrid transparently activates the server when a client resolves the first indirect proxy to an object in the server.

The easiest way to deactivate a server is to set `Ice.ServerIdleTime` to a timeout in seconds.

If the server is idle for the specified timeout, its object adapters shut down and `waitForShutdown` completes.

The Process Administrative Facet

```
module Ice {
    interface Process {
        idempotent void shutdown();
        void writeMessage(string message, int fd);
    };
};
```

IceGrid adds a `Process` facet to an adapter that runs at the server's `Ice.Admin.Endpoint` (127.0.0.1 by default). The `server stop` command calls `shutdown` on the facet and the implementation of that operation calls `shutdown` on the communicator.

This allows the server to shut down when asked to do so by `icegridadmin`.

You can specify a different admin endpoint for the server by setting the `Ice.Admin.Endpoints` property for the server.

Server Environment

When you use `application add` or `application update`, `icegridadmin` writes a configuration file for a server.

The configuration file is stored in

`<node-dir>/servers/<server-id>/config/config`

For example:

`Node1/servers/fsserver/config/config`

This configuration file contains any property settings you specified in the deployment descriptor.

When IceGrid starts a server, it passes

`--Ice.Config=Node1/servers/fsserver/config/config`

as an option to the server, so the server gets the correct configuration.

Server Code Changes

The server's object adapter obtains its endpoints from the property settings generated by the IceGrid node.

When creating an object adapter, use:

`communicator.createObjectAdapter("`*`<adapter-name>`*`")`

without specifying any endpoints.

The adapter name must match the `name` attribute of the `adapter` element in the server's deployment descriptor.

The adapter will listen on the endpoints specified by the `adapter` element, which are transferred to the *`<adapter-name>.`*`Endpoints` property.

The adapter informs IceGrid of its endpoints so the registry can resolve indirect proxies.

ZeroC

The Graphical Admin Tool

Ice provides a GUI tool that provides most of the functionality of `icegridadmin`.

The tool is provided as a stand-alone jar file in the Ice distribution.

To start the tool:

```
java -jar IceGridGUI.jar
```

The tool prompts for the value of `Ice.Default.Locator` so it can find the registry.

For the GUI tool to work, either:

- set `IceGrid.Registry.CryptPasswords`

or set one of the following properties to a custom verifier:

- `IceGrid.Registry.AdminPermissionsVerifier` (for TCP)
- `IceGrid.Registry.AdminSSLPermissionsVerifier` (for SSL)

Well-Known Objects

The registry maintains a table of well-known objects. The table stores a name–proxy pair. You can populate the table

- via the deployment descriptor

- via the `icegridadmin` or `IceGridGUI.jar` tools

- programmatically, via the registry's Slice interface

A proxy for a well-known object consists of only an identity.

Minimally (using the default protocol), a proxy is:

    RootDir

You can add an object element as a child of an adapter element to declare a well-known object:

```
<adapter name="Lab8" id="fsadapter" endpoints="tcp">
  <object identity="RootDir"/>
</adapter>
```

`RootDir` and `RootDir@fsadapter` are now equivalent.

Well-Known Proxies (1)

```
module IceGrid {
  interface Admin {
  void addObject(Object* obj)
        throws ObjectExistsException,
        DeploymentException;
  void updateObject(Object* obj)
        throws ObjectNotRegisteredException,
            DeploymentException;
  void addObjectWithType(Object* obj, string type)
        throws ObjectExistsException,
            DeploymentException;
  void removeObject(Ice::Identity id)
        throws ObjectNotRegisteredException,
            DeploymentException;
  idempotent ObjectInfoSeq getAllObjectInfos(
                                  string expr);
  idempotent ObjectInfo getObjectInfo(Ice::Identity id)
        throws ObjectNotRegisteredException;
  };
};
```

Security Considerations

Do not permit the server, node, and internal endpoints to be accessible in hostile environments.

In hostile environments, you must use SSL and appropriate certificates to secure these endpoints.

- Under UNIX, if you run the node as a user other than root, servers are started with that user ID.

- If you run the node as `root`:

  - If you do not specify a user attribute for the server descriptor, the server runs as nobody.

  - Otherwise, it runs as the specified user.

Troubleshooting

You can set various tracing properties to diagnose problems:

`IceGrid.Registry.Trace.Adapter=3`

`IceGrid.Registry.Trace.Node=2`

`IceGrid.Registry.Trace.Server=1`

`IceGrid.Registry.Trace.Object=1`

`IceGrid.Registry.Trace.Locator=2`

`IceGrid.Node.Trace.Activator=3`

`IceGrid.Node.Trace.Adapter=3`

`IceGrid.Node.Trace.Server=3`

These properties produce trace messages for the corresponding area of interest.

If you run the registry/node in a window from the command line, trace output is written to the terminal.

Beware of relative pathnames for executables and files.

Failure to start a server can be related to `LD_LIBRARY_PATH`.

Check for core files in the node's working directory.

Other Features

IceGrid provides a number of other features (not further covered here):

- Templates

  Templates are generic deployment descriptors so you can describe a whole family of servers with a single descriptor.

- Server allocation

  You can reserve a specific number of server instances for allocation and exclusive use by particular clients.

- Replication

  You can replicate objects across a number of servers; if one server is down, IceGrid transparently chooses a working replica in a different server on behalf of clients. You can also replicate the IceGrid registry to achieve fault tolerance.

- Load balancing

  For replicated objects, IceGrid can dynamically load balance among the objects.

Ice Programming with Java

17. Assignment 8
Using IceGrid

ZeroC

Exercise Overview

In this exercise, you will:

- modify the file system application to work with IceGrid.

By the completion of this exercise, you will have gained experience in how to run IceGrid, deploy a server, and use indirect proxies in clients to bind indirectly to Ice objects.

Using IceGrid

- In this exercise, you will modify the application you developed in Assignment 6 to use IceGrid.

ZeroC

What You Need to Do

1. Run an IceGrid registry in a window.

2. Run an IceGrid node in a separate window.

3. Create a deployment descriptor for your server in `filesystem.xml`.

4. Run `icegridadmin` in a separate window.

5. The server in `Server.java` does not create an object adapter. Add the missing code to create the adapter.

6. Start the server using `icegridadmin`.

7. Verify that you can cleanly stop the server using `icegridadmin`.

8. The client requires configuration to bind indirect references. Place the missing configuration for the client into `config.client`.

9. Modify the client source code to specify an indirect reference for the root directory that matches the deployment of your server.

10. Run the client with protocol tracing and examine the messages that are exchanged between the client and the registry.

11. Run the IceGridGUI tool and use it to modify the server's deployment to advertise the root directory as a well-known object with the identity "RootDir".

Registry Configuration

```
IceGrid.Registry.Client.Endpoints=tcp -p 4061
IceGrid.Registry.Server.Endpoints=tcp
IceGrid.Registry.Internal.Endpoints=tcp
IceGrid.Registry.Data=registry
IceGrid.Registry.AdminPermissionsVerifier=IceGrid/Nu
llPermissionsVerifier

IceGrid.Registry.Trace.Locator=2
IceGrid.Registry.Trace.Adapter=3
IceGrid.Registry.Trace.Node=2
IceGrid.Registry.Trace.Server=1
IceGrid.Registry.Trace.Object=1
```

ZeroC

Node Configuration

```
Ice.Default.Locator=IceGrid/Locator:tcp -p 4061

IceGrid.Node.Endpoints=tcp
IceGrid.Node.Name=Node
IceGrid.Node.Data=node

IceGrid.Node.Trace.Activator=3
IceGrid.Node.Trace.Adapter=3
IceGrid.Node.Trace.Server=3
```

ZeroC

Deployment Descriptor

```
<icegrid>
  <application name="filesystem">
    <node name="Node">
      <server id="fsserver" exe="java" activation="on-demand">
        <adapter name="Lab8" id="fsadapter" endpoints="tcp">
        </adapter>
        <property name="Ice.ServerIdleTime" value="20"/>
        <option>Server</option>
      </server>
    </node>
  </application>
</icegrid>
```

Admin Configuration

- `Ice.Default.Locator=IceGrid/Locator:tcp -p 4061`

ZeroC

Server Source Modification

The server must call `createObjectAdapter` (instead of `createObjectAdapterWithEndpoints`) to create the adapter:

```
Ice.ObjectAdapter adapter =
    communicator().createObjectAdapter("Lab8");
```

ZeroC

Client Configuration

- `Ice.Default.Locator=IceGrid/Locator:tcp -p 4061`

Client Modification

- For step 9, the client needs to use the proxy:

  `RootDir@fsadapter`.

- For step 11, the proxy is:

  `RootDir`.

Ice Programming with Java

18. The Ice Run Time in Detail

Lesson Overview

- This lesson:

  - takes closer look at the Ice run time.

  - explains some advanced implementation techniques that allow you to take precise control of the performance–footprint trade-off for a server.

- By the completion of this chapter, you will know how to build realistic server applications that can scale to millions of objects.

ZeroC

The `Ice::Communicator` Interface

Ice::Communicator is the main handle to the Ice run time.

The communicator is associated with a number of resources:

- Client- and server-side thread pool
- Configuration properties
- Object factories to instantiate Slice classes
- A logger object to redirect warning and error messages
- A statistics objects to collect statistics on traffic volumes
- A default router (used by Glacier2)
- A default locator (used by IceGrid)
- A plug-in manager to load plug-ins (such as IceSSL)
- One or more object adapters

You can have more than one communicator in a process (for example, to use different configuration properties with each).

The `Ice::Communicator` Interface (1)

```
module Ice {
    local interface Communicator {
        string proxyToString(Object* obj);
        Object* stringToProxy(string str);

        ObjectAdapter createObjectAdapter(string name);
        ObjectAdapter createObjectAdapterWithEndpoints(
                                    string name,
                                    string endpoints);
        void shutdown();
        void waitForShutdown();
        void destroy();
        // ...
    };
    // ...
};
```

ZeroC

Creating a Communicator

```
final class InitializationData implements Cloneable
{
    public java.lang.Object clone();
    public Properties properties;
    public Logger logger;
    public Stats stats;
    public ThreadNotification threadHook;
    public ClassLoader classLoader;
    public Dispatcher dispatcher;
}
static Communicator initialize();
static Communicator initialize(String[] args);
static Communicator initialize(StringSeqHolder ah);
static Communicator initialize(InitializationData id);
static Communicator initialize(String[] args,
                               InitializationData id);
static Communicator initialize(StringSeqHolder ah,
                               InitializationData id);
```

Object Adapters

Object adapters link the server-side run time to the server-side application code.

Each server has at least one object adapter.

Each object adapter provides one or more transport endpoints at which it listens for incoming requests.

Each object adapter provides an Active Servant Map to dispatch incoming requests.

Operations that manipulate the ASM:

```
Object* add(Object servant, Identity id)

Object* addWithUUID(Object servant)

Object remove(Identity id)

idempotent Object find(Identity id)
```

ZeroC

Servant Locators

By default, if the identity for an incoming request cannot be found in the ASM, the run time returns `ObjectNotExistException` to the client.

You can register one or more servant locators with an object adapter.

The job of a servant locator is to locate or create a servant for a request.

```
local interface ServantLocator
{
    Object locate(Current curr,
                    out LocalObject cookie);
    void finished(Current curr,
                    Object servant,
                    LocalObject cookie);

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

Threading Guarantees for Servant Locator

Guarantees provided by the Ice run time:

- Every call to `locate` is balanced by a call to `finished`.

- `locate`, the servant operation, and `finished` are called by the same thread. (When using AMD, `finished` may be called by a different thread.)

- No call to `locate` or `finished` can arrive after `deactivate` is called, and `deactivate` is not called concurrently with `locate` or `finished`.

Note that:

- Multiple calls to `locate` can proceed concurrently.
- Multiple calls to `finished` can proceed concurrently.
- `locate` and `finished` can proceed concurrently.

Concurrency can involve the same object ID!

Servant Locator Registration

```
local interface ObjectAdapter {
    void addServantLocator(ServantLocator locator,
                           string          category);

    ServantLocator findServantLocator(string category);

    // ...
};
```

Note that a servant locator is registered for a specific category.

- If the target identity of an incoming request has a matching category, the run time calls the corresponding servant locator.

- Otherwise, if you have servant locator with an empty category, the run time calls that servant locator (known as the *default locator*).

Call Dispatch Rules

1. Look for the identity in the ASM. If the ASM contains an entry, dispatch the request. Finished.

2. If the category of the request is non-empty, look for a matching servant locator.

   - If a matching locator is found, call its `locate` operation. If `locate` returns a servant, dispatch the request; otherwise, throw `ObjectNotExistException`. Finished.

   - If no matching locator is found, continue with Step 3.

3. Look for a default servant locator.

   - If a default servant locator is found, call its `locate` operation. If `locate` returns a servant, dispatch the request; otherwise, throw `ObjectNotExistException`. Finished.

   - If no default locator is found, continue with Step 4.

4. Raise `ObjectNotExistException` in the client.

Implementing Servant Locators

Each servant locator must be derived from the `Ice::ServantLocator` base class:

```
public class MyServantLocator implements Ice.ServantLocator
{
    public Ice.Object
    locate(Ice.Current c, Ice.LocalObjectHolder cookie);

    public void
    finished(Ice.Current c,
             Ice.Object servant,
             Object cookie);

    public void
    deactivate(String category);
}
```

Implementing locate

```
public Ice.Object
locate(Ice.Current c, Ice.LocalObjectHolder cookie)
{
    MyServantDetails d = null;
    try {
        d = DB_lookup(c.id.name);
    } catch (DB_error e)
        return null;
    }
    return new MyInterfaceI(d);
}
```

This implementation locates the state for a servant in a database.

Note that, for each request, a new servant is created.

- Depending on the relative costs of operations and initialization, this may be inefficient.

- Without interlocks, this can result in multiple servants for the same Ice object, if requests arrive concurrently.

Information Provided to `locate`

`locate` is passed the `Ice::Current` object for the incoming request.

`Ice::Current` contains the identity for the incoming request, and the operation name.

Typically, this is all the information you need to locate the correct servant for a request.

Note that `locate` must usually instantiate a servant, but the type of the servant's interface is *not* part of the `Current` object.

If you have locators for servants with different interfaces, you must register a separate locator for each interface type.

The category can be any identifier you choose; you can use the type ID of the servant's interface, or any other suitable identifier.

Lazy Initialization

```
Ice.Object
locate(Ice.Current c, Ice.LocalObjectHolder cookie)
{
    MyServantDetails d = null;
    try {
        d = DB_lookup(c.id.name);
    } catch (DB_error e)
        return null;
    }
    myInterfaceI servant = new MyInterfaceI(d);
    try {
        c.adapter.add(servant, c.id);
    } catch (Ice.AlreadyRegisteredException ex)
        return c.adapter.find(c.id);
    }
    return servant;
}
```

Creating Proxies

You can create a proxy for an Ice object without instantiating a servant for that object:

```
local interface ObjectAdapter {
    Object* createProxy(Identity id);
    // ...
};
```

This is more efficient than instantiating a servant and adding it to the ASM in order to obtain its proxy.

`createProxy` is particularly useful for `list` operations that return a large number of proxies to clients.

When used in combination with servant locators, this avoids having to instantiate a servant for each Ice object returned in a list.

Default Servants

A servant that implements many different Ice objects simultaneously is called a *default servant*.

Default servants are useful for servers that act as a front end to backend storage, such as servers that sit in front of a database and present database records as Ice objects.

Ice provides an API similar to servant locators that makes it easy to register your default servants.

Each operation implementation uses the object identity for the request to determine which servant state to operate on.

Default servants allow unlimited scalability with very small memory footprint.

Default Servants (1)

With a default servant, the implementation of each operation:

- uses the `Current` object to get the object identity

- uses the `name` member of the identity to locate the state of the Ice object (for example, by using it as the key of a database table). If no state can be found for the identity, the operation throws `ObjectNotExistException`.

- Implements the operation to operate on the retrieved state.

This makes the server completely stateless. Each operation retrieves the state, operates on it, and forgets the state again.

Default Servants (2)

If you use default servants, you should override the `ice_ping` operation on the skeleton to do the right thing.

The inherited default implementation always succeeds. However, if you use a default servant, a client may ping an Ice object that truly does not exist.

```
void
ice_ping(Ice.Current c)
{
    try {
        DB_lookup(c.id.name);
    } catch (DB_error ex)
        throw new Ice.ObjectNotExistException() ;
    }
}
```

You need to override `ice_ping` only if clients actually use it (but it is good practice to do so).

Hybrid Approaches and Caching

You can combine the ASM and a default servant:

- Put performance-critical servants that are accessed frequently into the ASM.

  The implementation of these servants should cache all servant state in memory to get good performance.

- Use a default servant for less frequently-accessed servants.

  The implementation of these servants retrieves state on demand from back-end storage, to keep memory consumption low.

This approach is useful if the access patterns to servants are known in advance and static.

ZeroC

Evictors

An *evictor* is a servant locator that instantiates servants up to some predefined maximum number of instances:

- If a request arrives for a servant that is not yet in memory, the servant locator instantiates a new servant and returns it, provided that the limit of servants is not exceeded.

- If a request arrives for a servant that is already in memory, the servant locator returns that servant.

- If a request arrives for a servant that is not in memory, and the number of servants is already at the limit, the servant locator destroys an existing servant and instantiates a new one.

    The servant that is evicted is the least-recently-used servant.

Evictors allow control of the footprint–performance trade-off in a server.

By choosing the evictor size appropriately, you get good performance for the most frequently-used servants, with acceptable memory consumption.
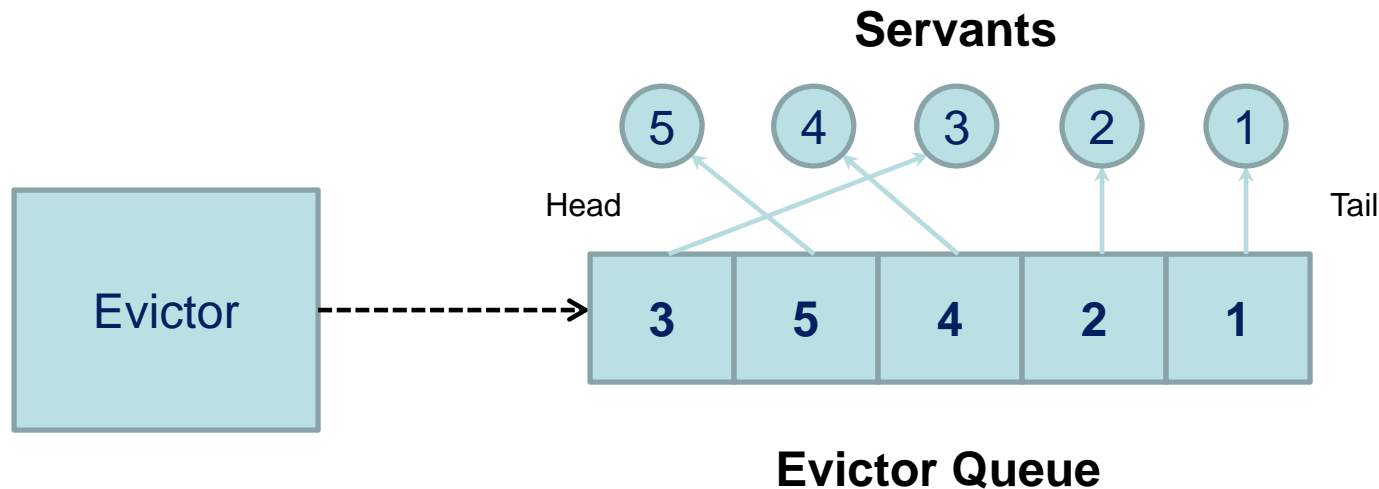
Evictors (1)

An evictor after requests for object identities 1 to 5 have arrived (in that order):

Servants

5   4   3   2   1

Head   Tail

Evictor - - - - - -> | 5 | 4 | 3 | 2 | 1 |
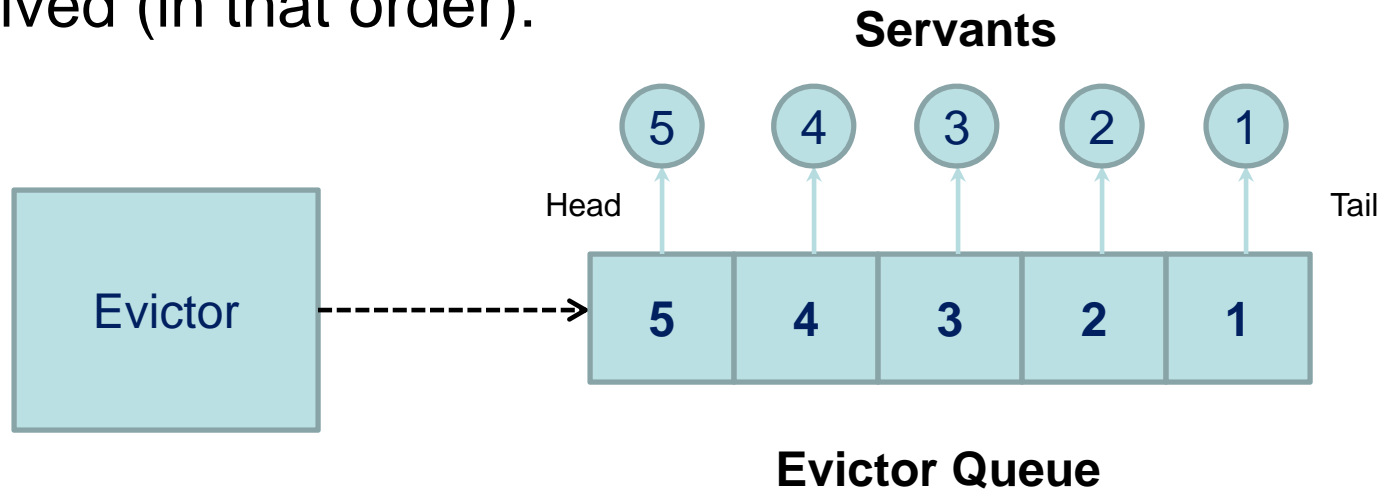
Evictor Queue

The evictor has instantiated five servants. Servant 1 is the least recently-used servant.

Evictors (2)

Same evictor after a client accesses servant 3:

Servants

Evictor Queue

The evictor has dequeued the entry for servant 3 and placed it at the head of the evictor queue, making servant 3 the most recently-used servant.

Evictors (3)

Same evictor after a client accesses servant 6:

Servants

Evictor Queue
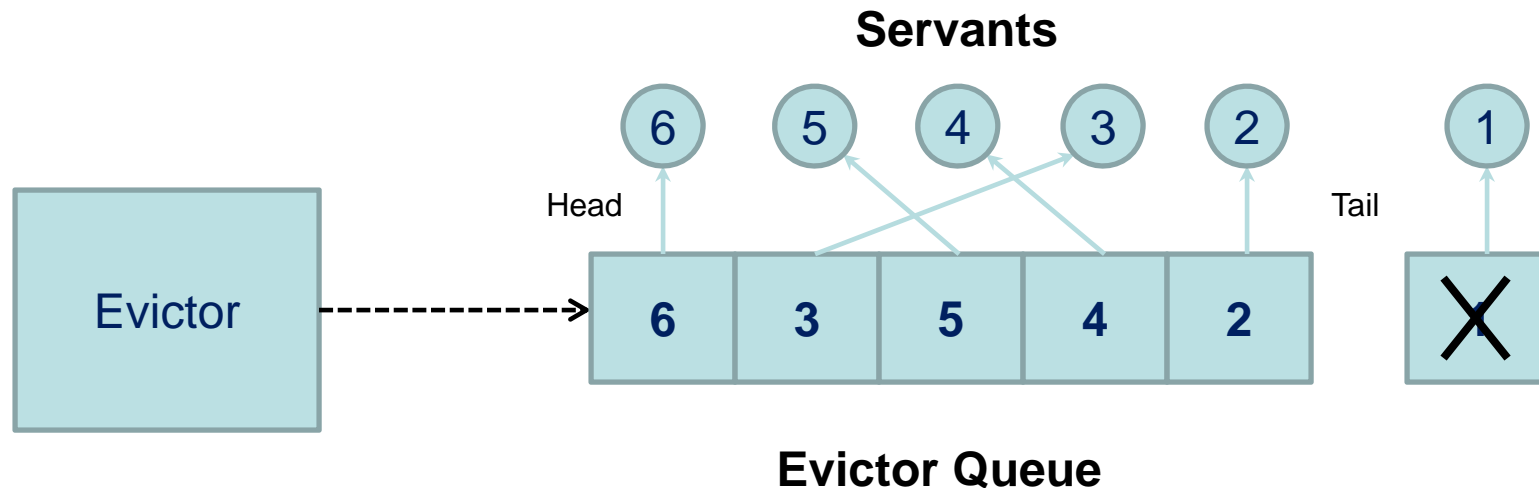
The least recently-used servant (servant 1) has been removed from the evictor and is destroyed once it no longer services a request.

ZeroC

Evictor Implementation

Implementation goals:

- Reusable, so it can be used for any type of servant.

- Non-intrusive to servant implementation. (Servant implementation should not know about evictor.)

- High performance for both locating a servant and evicting a servant.

- Easily configurable evictor size.

Basic implementation:

- Use a map to store identity–servant pairs for quick lookup.

- Use a queue to maintain LRU order. Queue entries point at map entries. Enqueing, dequeuing, and maintaining LRU order can be performed in constant time.

- Implementation is inherited from a base class.

Evictor Implementation (1)

The private part of `EvictorBase`:

- stores the cookie that is returned from `add` (so it can be passed to `evict`) in a map,

- stores an iterator into the evictor queue that marks the position of the servant in the queue,

- stores a use count for each servant that is incremented when an operation is dispatched, and decremented when an operation completes.

Using EvictorBase

```
public class MyInterfaceEvictor extends EvictorBase
{
    public Ice.Object
    add(Ice.Current c, Ice.LocalObjectHolder cookie)
    {
        MyServantDetails d = null;
        try {
            d = DB_lookup(c.id.name);
        } catch (DB_error ex)
            return null;
        }
        return new MyInterfaceI(d);
    }

    public void
    evict(Ice.Object servant, Object cookie)
    {
    }
}
```