CU CS 4995-03: Internet Systems Programming

Corba

Overview

- architecture
- IDL
- C++ interface
- naming service
- trader

- Corba: specified by OMG (created 1989)
- “portable distributed applications for heterogeneous systems”
- distributed objects
- object = encapsulated distinct identity accessed through well-defined interfaces

Services

Object services: common services, e.g., naming, broker

Domain services: application-area, e.g., finance, telecom, manufacturing, health care, ...

Application interfaces: specific to one application

Terminology

CORBA object: virtual object, realized in a program

Target object: target of request, *single-dispatching*

Client: invokes request on object; may share address space

Server: application housing one or more objects

Request: from client to target object

Object reference: opaque identifiers

Servant: prog. language entity that implements (“incarnates”) one or more objects; C++: object instances of class

Corba request styles

synchronous: block; = RPC

deferred synchronous: send then poll; needs DII

oneway request: best-effort, no response

asynchronous requests: occasionally connected client/servers; future!

Corba implementation

same process	function call
same host, different process	shared memory
different host	remote procedure call

Corba Interface Definition Language (IDL)

- define object interfaces
- independent of programming language
- simple types (signed/unsigned integer, characters, Boolean, strings, ...) and constructed types (enum, structures, discriminated union, sequences, exceptions, ...)
- example interface:

```
interface Employee {
  long number();
};
```

- object:

```
interface EmployeeRegistry {
  Employee lookup (in long emp_number);
};
```

- must declare direction: `in`, `out`, `inout`

IDL interface inheritance

```
interface Printer {
  void print();
}
interface ColorPrinter : Printer {
  enum ColorMode { BlackAndWhite, FullColor };
  void set_color(in ColorMode mode);
}
```

all inherit from `Object`

Language mappings

	C++	Java	C
interface	classes	public	
operations	member f.		functions
object ref.			void *

also: Smalltalk, COBOL, Ada; Eiffel, Modula 3, Perl, Tcl, Objective-C, Python

Dispatch

Static: stubs and skeletons, *compiled* into application

- *proxy*: represent remote object
- skeleton = server-side function (BOA or POA)
- generally used by C++ and Java

Dynamic: run-time

- with dynamic information about interfaces and types ("Interface Repository")
- gateways and bridges: forward generic requests

Inter-Orb Protocols

- CORBA 2.0: General Inter-ORB protocol (GIOP)
- Internet Inter-ORB protocol (IIOP) implements GIOP over TCP/IP
- object reference format: Interoperable Object Reference (IOR): host name, port number, object key

Request invocation

- location transparency; objects can move!
- server transparency
- language independence: Java calls C++
- implementation independence: OO vs. pointers
- architecture independence
- operating system independence
- protocol independence
- transport independence

Object references

$\approx$ class instance pointers

- each reference $\longrightarrow$ one object
- multiple references per object
- nil reference is allowed
- references can dangle: `non_existent` operation
- references are opaque
- strongly typed
- late binding = C++ virtual functions
- references can be persistent (but usually aren't)
- references interoperable

How to get references?

Servers publish references:

- return value, `inout` parameter: “object web”
- advertise in naming or trading service
- convert to text string
- out-of-band: http, email, ...

IOR

- repository ID: locate interface in Interface Repository
- endpoint information: domain name, port; several protocols
- object key

Application development

1. define objects in IDL
2. compile IDL into C++ stubs and skeletons (e.g., echo.hh and echoSK.cc)
3. declare and implement C++ servant classes
4. server main program
5. compile and link with stubs and skeletons
6. write, compile and link client code

IDL

- contract between client and server
- independent of implementation language
- declarative language
- object interfaces, operations, exceptions
- IDL compiler generates `.hh` file
- free-form language
- C++ preprocessor, comments
- case-insensitive

Basic IDL types

`short,unsigned short`	$\geq$ 16 bits
`long,unsigned long`	$\geq$ 32 bits
`float`	IEEE single precision
`double`	IEEE double precision
`char`	ISO Latin-1
`string`	ISO Latin-1, except NUL
`boolean`	TRUE or FALSE
`octet`	0–255
`any`	run-time identifiable type

User-defined types

```
typedef short YearType;
typedef YearType Year;  // bad idea
```

can be either new type or alias (C++: compatible)

```
enum Food {Turkey, Dressing, Beans, Cranberries, Sweetpotatoe};
```

but: no value assignments allowed!

```
struct TimeOfDay {
  short hour;
  short minute;
  short second;
}
```

User-defined types

```
union ColorCount switch (Color) {
  case red:
  case green:
  case blue:
    unsigned long num_in_stock;
  case black:
    float discount;
  default:
    order_details;
}
// for optional element
union AgeOpt switch(boolean) {
  case TRUE:
    unsigned short age;
}
```

Arrays

```
typedef Color ColorVector[10];
typedef string IDtable[10][20];
```

- all array dimensions must be specified
- must be `typedef`'d
- $\longrightarrow$ C++ arrays

Sequences

```
typedef sequence<Color> Colors; // unbounded
typedef sequence<long, 100> Numbers;  // 100 elements
typedef sequence<sequence<long, 100>> ListOfNumberVectors;
```

Typically, $O(1)$ access time, but no guarantees

Recursive Data Types

- IDL has no pointers
- but: can recurse via anonymous sequence of incomplete type
- e.g., for graph

```
struct Node {
  long value;
  sequence<Node> children;
}
```

Recursive Data Type: Tree

```
enum OpType {OP_AND, OP_OR, OP_NOT};
enum NodeKind {LEAF, UNARY, BINARY};
union Node switch (NodeKind) {
case LEAF:
  long value;
case UNARY:
  struct UnaryOp {
    OpType op;
    sequence<Node,1> child;
  }
case BINARY:
  struct BinaryOp {
    OpType op;
    sequence<Node,2> children;
  }
}
```

Constant definitions and literals

Almost exactly like C++:

```
const float E = 2.7183;
const char NUL = '\0';
const string FRAN_WORDS = "Not a mouse stirring.";
typedef short Age;
const Age MAX_AGE = 120;
```

IDL constant expressions

```
const short MIN_TEMP = -10;
const short MAX_TEMP = 35;
const short AVG_TEMP = (MAX_TEMP + MIN_TEMP)/2;
const long ALL_ONES = -1; // 0xffffffff
const long RHW_MASK = ALL_ONES >> 16; // 0x0000ffff - logical shift
```

IDL interfaces

IDL describes interface to objects

- IDL interface = name space
- IDL interfaces ≈ C++ classes (definitions)
- IDL operations ≈ C++ member functions
- Corba objects ≈ C++ class instances
- but…
  - no public, private or protected: everything is public
  - no member variables: state hidden by interface
  - no nested classes (↔ C++)
  - scoping via `::`

IDL example

```
interface Student {
  typedef unsigned long ID;
  ID create(in string first, in string last);
  boolean graduate(in ID id);
  exception NotFound {
    unsigned long searched;
  }
}

interface Class {
  readonly attribute unsigned long size;
  typedef string Student;      // by SSN

  void add(in Student s) raises (Student::NotFound);
  void grade(in Student s, in float grade);
}
```

- object references $\approx$ C++ class instance pointer
- *cooperating interfaces*: behind-the-scenes (here: `grade`)

IDL operations

- just side effects: `void op();`
- directional attributes:

`in`	from client to server
`out`	from server to client
`inout`	initialized by client, modified by server

- needed for efficiency, memory management
- choice `int op(in arg);` or `void op(in arg, out result);`
- `inout` dictate policy, generally bad
- no overloading of operations
- no "read-write" distinction: can't say `void op() const;`

User Exceptions

```
interface Finding {
  enum WhyType {dead, never_existed};
  exception NotFound {
    unsigned long ID;
    unsigned WhyType why;
  }
  ID find(in string name) raises(NotFound);
}
```

- Unlike C++, no exception inheritance
- don’t use for common return values

Rules for exceptions

- only for exceptional conditions
- carry useful information (why, where, what to do, ...)
- precise information: one error case = one exception
- complete, without additional calls
- cater to user, not implementor
- don't mix return values and exceptions for the same type of errors

System exceptions

•

```
enum completion_status {
  COMPLETED_YES,   // after side effect (idempotency)
  COMPLETED_NO,    // before side effect
  COMPLETED_MAYBE
};
exception NAME {
  unsigned long minor;  // ORB-defined
  completion_status completed;
};
```

Examples:

`BAD_PARAM`	invalid parameter
`COMM_FAILURE`	communication failure
`NO_MEMORY`	memory allocation failure

Don't (generally) use for application errors!

Oneway Operations

```
interface Events {
  oneway void send(in EventData data);
};
```

- return type `void`
- no `inout` parameters
- can't raise exceptions
- "at most once", but no implementation guarantees
- may block, may be reordered

Attributes

- similar to C++ public member variables

```
interface Student {
  readonly attribute float GPA;
  attribute MaritalStatus ms;
}
```

- just abbreviation for get/set operations
- can't raise user exceptions; limited error reporting
- thus, syntactic sugar

Modules

- name space ($\approx$ C++), without hiding
- can be nested
- can be re-opened for incremental definition

```
module University {
  typedef string ID;
  typedef float  GPA;

  interface Student {
    FAS::Account paid_from();
    ...
  }
  interface Class {
    ...
  }
}
```

Inheritance

```
interface Thermometer {
  typedef short TempType;
  readonly attribute TempType temperature;
}
interface Thermostat : Thermometer {
  void set_nominal_temp(in TempType t);
}
```

Polymorphism $\longrightarrow$ allow adding thermostats:

```
interface Logger {
  long add(in Thermometer t, int unsigned when);
}
```

Inheritance

- all interfaces inherit from `Object`

```
interface Generic {
  void accept(in Object o);
  Object lookup(in KeyType key);
}
```

- empty interfaces are legal: allow inheritance and object-passing (abstract base interface)
- just *definition* inheritance $\longrightarrow$ C++ implementation inheritance:
  - same address space, C++ inheritance
  - delegation
  - separate implementations in same address space
- derived interfaces can (but shouldn't) redefine types, constants, exceptions, but not attributes or operations

Multiple inheritance

```
interface MSstudent {};
interface PhDstudent {};
interface Student : MSstudent, PhDstudent {};
```

- Can't inherit same operation or attribute twice

```
interface MSstudent {
  attribute dob;
};
interface PhDstudent {
  attribute dob;
};
```

Repository identifiers

- run-time access to interfaces:
  - IDL format: `IDL:acme.com/CCS/TempType:1.0`; add prefix

    `#pragma prefix "acme.com"`
  - DCE UUID format: `DCE:48ad438-0419-ac8f:1`
  - local format: `LOCAL:some random string`
- versioning doesn't work (always effectively 1.0)

Mapping IDL for C++

- names are mapped as is: C++ for IDL `enum Color..;` is

  `enum Color {red, green, blue};`

- also preserves scoping
- avoid C++ reserved names, __
- modules $\longrightarrow$ name spaces

Mapping IDL to C++

IDL	C++
short, long, ...	`CORBA::Short,CORBA::Long,...`
unsigned short	`CORBA::UShort`
string	`char *`

String allocation:

```
char *p = CORBA::string_alloc(5);
char *s = CORBA::string_dup("Hello");
```

Variable-length types: memory management

- strings, object references, `any`, sequences, structures/unions/arrays with variable-length members,
- uses "return pointer to dynamic memory" approach
- caller responsible for deallocation
- generates two classes: `name` (low-level, with pointers) and `name_var` (memory management wrapper)

String_var

```
class String_var {
public:
  String_var();
  String_var(char *);
  String_var(const char *);
  ~String_var();
  String_var & operator=(char *);
  String_var & operator=(const char *); // deep copy
  String_var & operator=(const String_var & s);
  char & operator[](ULong index);
  operator char *();
private:
  char *s;
}
```

String copy

```
CORBA::String_var target = CORBA::string_dup("Hello");
CORBA::String_var source = CORBA::string_dup("World");
target = source;  // deallocates "Hello"
```

- Can be passed transparently as `char *`, but careful with string literals in classical C++ (`char *`).
- use `const char *` instead of `CORBA::String_var` for read-only:

```
void print_string(CORBA::String_var s)
void print_string(const char *s)
```

Fixed-Length structures

```
struct Details {
  double weight;
  unsigned long count;
}
```

maps to

```
struct Details {
  CORBA::Double weight;
  CORBA::ULong count;
  typedef Details_var _var_type;
}
```

Client and server interfaces

```
interface Echo {
  string echoString(in string mesg);
}

class Echo;
typedef Echo* Echo_ptr;

class Echo : public virtual omniObject, public virtual CORBA::Object {
  virtual char *echoString(const char *, mesg) = 0;
  static Echo_ptr _nil();
  static Echo_ptr _duplicate(Echo_ptr);
  static Echo_ptr _narrow(CORBA::Object_ptr);
}
```

Standard functions

`CORBA::is_nil()`: test for nil object

`CORBA::is_equivalent(`x`)`: object equivalence

`Echo::_nil()`: return nil object

`_narrow()`: cast from object to type

`_duplicate()`: create a copy

object references inherited from `Object_ptr`

Object implementation

may differ from ORB to ORB!

```
class _sk_Echo : public virtual Echo {
  _sk_Echo(const omniORB::objectKey& k);
  virtual char *echoString(const char *mesg) = 0;
  Echo_ptr _this();
  void _obj_is_ready(BOA_ptr);
  void _dispose();
  BOA_ptr _boa();
  omniORB::objectKey _key();
}
```

Object implementation

`_this()`: object reference; convert to string via `object_to_string(o)`

`_object_is_ready()`: ready to serve

`_dispose()`: stop serving calls

`_boa()`: BOA object reference

`_key()`: ORB identifier

Writing the server

- create implementation class, inherits from `_sk`:

```
class Echo_i : public virtual _sk_Echo {
public:
  Echo_i() {}
  virtual ~Echo_i() {}
  virtual char * echoString(const char *mesg);
};
// do the actual work
char * Echo_i::echoString(const char *mesg) {
  char *p = CORBA::string_dup(mesg);
  return p;
}
```

Writing the server: storage

dir.	owner
`in`	client or caller to implementation function
	others must copy
`out`	object implementation allocates
	client releases

Object implementation

```
CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");
CORBA::BOA_ptr boa = orb->BOA_init(argc,argv,"omniORB2_BOA");

Echo_i *myobj = new Echo_i();
myobj->_obj_is_ready(boa);

{
  Echo_var myobjRef = myobj->_this();
  CORBA::String_var p = orb->object_to_string(myobjRef);
  cerr << "'" << (char*)p << "'" << endl;
}

boa->impl_is_ready();
```

Client implementation

```
CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");
try {
  CORBA::Object_var obj = orb->string_to_object(argv[1]);
  hello(obj);
}
catch(CORBA::COMM_FAILURE& ex) {
}
catch(omniORB::fatalException& ex) {
}
catch(...) {
}
orb->NP_destroy();
```

Client implementation: object use

```
void hello(CORBA::Object_ptr obj)
{
  Echo_var e = Echo::_narrow(obj);
  if (CORBA::is_nil(e)) {
    cerr << "hello: nil object reference.\n" << endl;
    return;
  }

  CORBA::String_var src = (const char*) "Hello!";
  CORBA::String_var dest;
  dest = e->echoString(src);
}
```

CORBA pitfalls

- multiple hops if server A returns object on server B
- object granularity: object hierarchy vs. “flat”
- sockets vs. calls on same host
- latency issues: *fat* operations: `structs` instead of interfaces
- coarse objects: objects into data (C-style)
- client-side caching

CORBA ORB Naming Service (COS)

- map names to object references: *name binding*
- “white pages”
- hierarchical naming, similar to file system
- *orphaned contexts*: roots
- start with `resolve_initial_references`
- can add names hierarchically

Federated naming

- single logical service, distributed physically
- reliability
- load sharing
- administration
- must copy a stringified reference for initial naming context (for now)
- architecture: fully connected or hierarchical

OMG Trading Service

- dynamic object discovery
- trader stores *descriptions* of services: yellow pages
- dynamic binding
- service offer = advertisement
- *export* services by *service provider*
- properties
- *withdraw* services (owner or others)
- *import*: search for services
- *service type* for categories
- search: service type, constraints, preferences
- federation of traders

Constraint language

```
Location == 'San Francisco Bay'
and Speed >= 120
and Size == 'P205/65R15'
and (Manufacturer == 'Bridgestone'
  or Manufacturer == 'Goodyear')
and ('Visa' in CreditCards)
```

similar to SQL “where”

Trading service issues

- problem: object selection
- “ORB meeting place”
- efficient at evaluating constraints
- properties can be dynamic
- objects need to be aware of properties used for advertisement
- “CORBA search engine”
- but: yet another component
- incomplete answers (federation failures, policies)

Corba event service

- consumer
- supplier
- event channel
- push vs. pull style
- canonical push: to all registered consumers
- canonical push: event channel as procurer
- hybrid push/pull

Event model

- lack of reliability
- multiple suppliers
- filtering
- event vs. asynchronous messaging