Object-Oriented Network Programming

Writing CORBA Applications

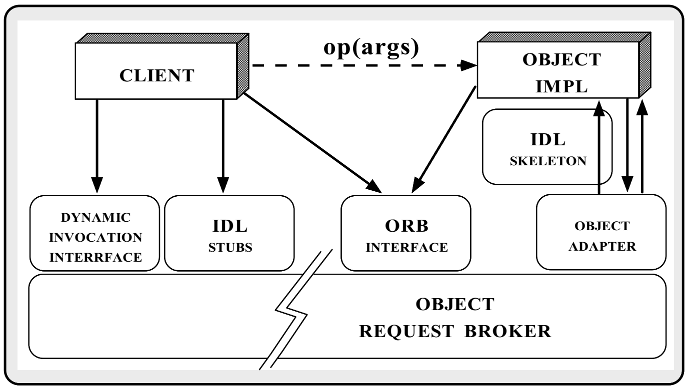
Dr. Douglas C. Schmidt
schmidt@cs.wustl.edu

Introduction

- CORBA addresses two challenges of developing distributed systems:

  1. Making distributed application development no more difficult than developing centralized programs
     - Easier said than done due to:
       - ▷ *Partial failures*
       - ▷ *Impact of latency*
       - ▷ *Load balancing*
       - ▷ *Event ordering*
  2. Providing an infrastructure to integrate application components into a distributed system
     - *i.e.*, CORBA is an "enabling technology"

General ORB structure



- Note that an ORB is a logical set of services, rather than just a particular process or library

CORBA Interface Definition Language (IDL)

- OMG IDL is an object-oriented interface definition language
  - Used to specify interfaces containing *methods* and *attributes*
  - OMG IDL support interface inheritance (both single and multiple inheritance)

- OMG IDL is designed to map onto multiple programming languages
  - *e.g.*, C, C++, Smalltalk, COBOL, Modula 3, DCE, etc.

OMG IDL Compiler

- A OMG IDL compiler generates client *stubs* and server *skeletons*
- Stubs and skeletons automate the following activities (in conjunction with the ORB):
  - *Client proxy factories*
  - *Parameter marshalling/demarshalling*
  - *Implementation class interface generation*
  - *Object registration and activation*
  - *Object location and binding*
  - *Per-object/per-process filters*

OMG IDL Features

- OMG IDL is a *superset* of a *subset* of C++
  - Note, it is not a complete programming language, it only defines interfaces
- OMG IDL supports the following features:

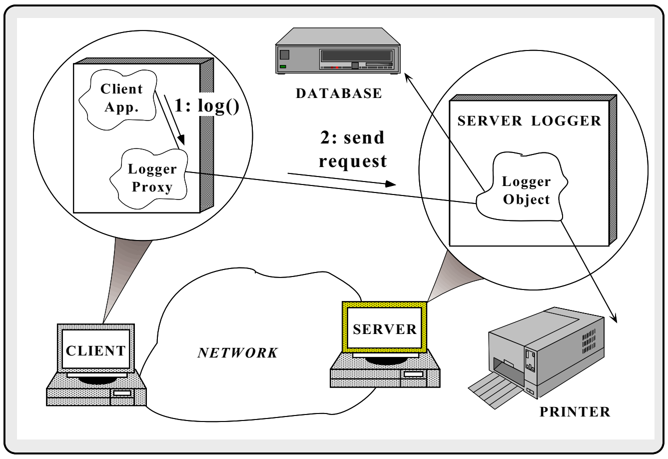
  * `modules`
  * `interfaces`
  * methods
  * attributes
  * inheritance
  * arrays
  * `sequence`
  * `struct, enum, union, typedef`
  * `consts`
  * `exceptions`

OMG IDL vs. C++

- Differences from C++

  * No data members
  * No pointers
  * No constructors or destructors
  * No overloaded methods
  * No **int** data type
  * Contains parameter passing modes
  * Unions require a tag
  * String type
  * Sequence type
  * Different exception interface
  * No templates
  * No control constructs

A Sample CORBA Application



- *Distributed logging facility*

Behavior of the Distributed Logging Facility

- The logging server collects, formats, and outputs logging records forwarded from applications residing throughout a network or internetwork
- An application interacts with the server logger via a CORBA interface

Server-side OMG IDL Specification

- Defines the interface of the Logger

```
// IDL specification
interface Logger
{
  // Types of logging messages
  enum Log_Priority {
      LOG_DEBUG,   // Debugging messages
      LOG_WARNING, // Warning messages
      LOG_ERROR,   // Errors
      LOG_EMERG    // A panic condition, normally broadcast
  };

  exception Disconnected { };

  struct Log_Record {
    Log_Priority type;  // Type of logging record.
    long host_addr; // IP address of the sender.
    long time; // Time logging record generated.
    long pid; // Process ID of app. generating the record.
    sequence<char> msg_data; // Logging record data.
  };

  // Transmit a Log_Record to the logging server
  void log (in Log_Record log_rec) raises (Disconnected);

  attribute boolean verbose; // Use verbose formatting
};
```


OMG IDL Mapping Rules

- The CORBA specification defines mappings from CORBA IDL to various programming languages
  - *e.g.*, C++, C, Smalltalk
- Mapping OMG IDL to C++
  - Each `interface` is mapped to a nested C++ class
  - Each operation is mapped to a C++ method with appropriate parameters
  - Each read/write attribute is mapped to a pair of get/set methods
    - ▹ A read-only attribute is only mapped to a single get method

Creating Server-side Implementations

- Running the Logger interface definition through the IDL compiler generates a *client* stub and a *server* skeleton
  - The client stub acts as a proxy and handles *object binding* and *parameter marshalling*
  - The server skeleton handles *object registration*, *activation*, and *parameter demarshalling*
- CORBA defines two techniques for generating server skeletons:

1. Inheritance-based implementations (*e.g.*, Orbix BOAImpl)
2. Object composition-based implemenations (*e.g.*, Orbix TIE)

Inheritance-based Implementations

- In Orbix, inheritance-based implementations are supported by the BOAImpl approach:
  - The drawback with this approach is that the implementation must inherit from the generated skeleton

```
class Logger_i
  // Note the use of inheritance from automatically
  // generated class LoggerBOAImpl
  : public virtual LoggerBOAImpl
{
public:
  Logger_i (bool verb): verbose_ (verb) {}
  virtual void log (const Log_Record &log_rec,
                    CORBA::Environment &);
  virtual bool verbose (void,
                    CORBA::Environment &);
  virtual void verbose (bool enable,
                    CORBA::Environment &);

private:
  bool verbose_;
};
```


Object Composition-based Implementations

- In Orbix, object composition-based implementations are supported by the TIE approach:

```
// Note, there is no use of inheritance and
// methods need not be virtual!
class Logger_i
{
public:
  // Start with verbose mode enabled.
  Logger_i (bool verb = true): verbose_ (verb) {}
  void log (const Log_Record &log_rec,
            CORBA::Environment &);
  bool verbose (void,
                CORBA::Environment &);
  void verbose (bool enable,
                CORBA::Environment &);

private:
  bool verbose_;
};
```


Object Composition-based Implementations (cont'd)

- Orbix provides a set of macros that tie the Logger interface together with the Logger_i implementation

```
DEF_TIE (Logger, Logger_i);
Logger_i *log = new Logger_i;
Logger *logger = new TIE (Logger, Logger_i) (log);
```

- This scheme works by placing a pointer to the implementation object within the TIE class and then delegating method calls to the implementation object

Writing the Server-side Method Definitions

- Using either the BOAImpl or the TIE approach, a developer then writes C++ definitions for the methods in class Logger_i:

```
void Logger_i::log (const Log_Record &log_rec,
                    CORBA::Environment &)
{
  // Formatting and outputting the contents
  // of log_rec omitted...
}

bool Logger_i::verbose (void,
                        CORBA::Environment &);
{
  return this->verbose_;
}

void Logger_i::verbose (bool enabled,
                        CORBA::Environment &);
{
  this->verbose_ = enabled;
}
```

Writing Main Server Program

- The main program for the logging server looks like:

```
// Shared activation
int
main (void)
{
  // BOAImpl instance.
  Logger_i logger ();

  try {
    // Will block forever waiting for incoming
    // invocations and dispatching method callbacks
    CORBA::Orbix.impl_is_ready ("Logger");
  } catch (...) {
    cerr << "server failed\n";
    return 1;
  }
  cout << "server terminating\n";
  return 0;
}
```


Exception Handling

- The preceeding example illustrated how CORBA uses C++ exception handling to propagate errors.
  - However, many C++ compilers don't support exceptions yet
  - Therefore, CORBA implementations provide an alternative mechanism for handling errors

```
// Shared activation
int main (void) {
  // Start with verbose mode enabled
  Logger_i logger (true);

  TRY {
    // Will block forever waiting for incoming
    // invocations and dispatching method callbacks
    CORBA::Orbix.impl_is_ready ("logger", IT_X);
  } CATCHANY {
    cerr << "server failed due to " << IT_X << endl;
  } ENDTRY;
  cout << "server terminating\n";
  return 0;
}
```


Object Activation

- If the service isn't running when a client invokes a method on an object it manages, the ORB will automatically start the service
- Services must be registered with the ORB, *e.g.*,

  % putit Logger /usr/svcs/Logger/logger.exe

- Service(s) may be installed on any machine
- Clients may bind to a service by using a location broker or by explicitly naming the server

Generated Client-side Stubs

- The OMG IDL compiler automatically generates a client-side stub used to define "proxy objects," *e.g.*,

```
typedef Logger *LoggerRef; // Generated by Orbix

class Logger
  // Base class for all IDL interfaces...
  : public virtual CORBA::Object
{
public:
  static Logger *_bind (/* Many binding formats */);
  virtual void log (const Log_Record &log_rec,
                    CORBA::Environment &);
  virtual void verbose (bool enabled,
                        CORBA::Environment &);
  virtual bool verbose (void,
                           CORBA::Environment &);
};
```

Binding a Client to a Target Object

- Steps for binding a client to a target object

1. A CORBA client (requestor) obtains an "object reference" from a server
   - May use a name service or locator service
2. This object reference serves as a local proxy for the remote target object
   - Object references may be passed as parameters to other remote objects
3. The client may then invoke methods on its proxy

Client-side Example

- A client programmer writes the following:

```
int
main (void)
{
  LoggerRef logger;
  Log_Record log_rec;

  logger = Logger::_bind (); // Bind to any logger.

  // Initialize the log_record
  log_rec.type = Logger::LOG_DEBUG;
  log_rec.time = ::time (0);
  log_rec.host_addr = // ...
  // ...

  try {
    logger->verbose (false); // Disable verbose logging.
    logger->log (log_rec); // Xmit logging record.
  }
  catch (Logger::Disconnected) {
    cerr << "logger disconnected" << endl;
  }
  catch (...) { /* ... */ }
  return 0;
}
```


Summary

- CORBA helps to reduce the complexity of developing distributed applications
  - However, there are many hard issues remaining...
- Other OMG documents (*e.g.*, COSS) specify higher level
  - *e.g.*, transactions, events, naming, security, etc.