CORBA ORB Implementation

Dr Duncan Grisby
Apasphere Ltd
dgrisby@apasphere.com

Apasphere

Outline

1. Introduction
2. omniORB structure
3. Threading
4. C++ and platform issues
5. Micro-optimisations
6. omniORB for Python

Introduction

1. About me
2. Overview
3. omniORB history
4. CORBA specifications

About me

- BA and PhD at the University of Cambridge Computer Laboratory.
- Worked at AT&T Laboratories Cambridge from 1999 to April 2002.
  - Part of the distributed systems group.
  - Worked on omniORB and omniORBpy.
  - Lab closed in April 2002.
- Founder of Apasphere Ltd.
  - CORBA and distributed systems consultancy.
  - omniORB commercial support.

Overview

- CORBA is an open standard framework for distributed applications.
  - Application authors do not have to know how the ORB works.
  - Please ignore this talk… :-)
- Wide scope
  - Interface Definition Language
  - Object model
  - Object Request Broker, Object Adapters
  - General Inter-ORB Protocol
  - Language mappings
  - …

Overview

- The CORBA standard specifies interfaces and semantics, not implementations.
  - No reference implementation.
  - First version sometimes not implementable.
- It often strongly implies an implementation.
  - Often not the most efficient way to do it.
  - If you don't do it the obvious way, it's harder to be sure you've done it right.
  - Testing...

omniORB

- An open source CORBA implementation.
  - Released under GNU LGPL (for libraries) and GPL (for tools).
- Current release is 4.0.0.
- Robust, high performance (often the fastest in tests), standards-compliant.
  - A difficult combination…
- Hosted at SourceForge
  - `omniorb.sourceforge.net`
- Commercial support available
  - `www.omniorb-support.com`

omniORB history

- Developed at Olivetti Research Ltd (ORL).
  - Originally designed for embedded platforms.
  - omniORB 1 used Orbix proprietary protocol.
  - omniORB 2 designed for IIOP.
- May 1997, omniORB 2.2 released to the world.
- March 1999, ORL became AT&T Laboratories Cambridge.
- April 2002, lab closed.
- omniORB lives on! (So does VNC: `www.realvnc.com`)
- Total of only 8 developers over time.

CORBA specifications

- CORBA is defined by the Object Management Group.
- All specifications available for free from `www.omg.org`.
- To contribute to specifications you have to pay.
- CORBA 3.0 recently released.
- Most (all?) implementations currently target 2.x.
- Specification consists of CORBA core, language mappings, services, domains.

CORBA core

- Object model
- Interface Definition Language (IDL)
- ORB interface
- Portable Object Adapter
- GIOP / IIOP
- Interface Repository
- Portable Interceptors
- …

Language mappings

- C++
- Python
- Java
- C
- Ada
- Lisp
- Smalltalk
- PL/1
- COBOL

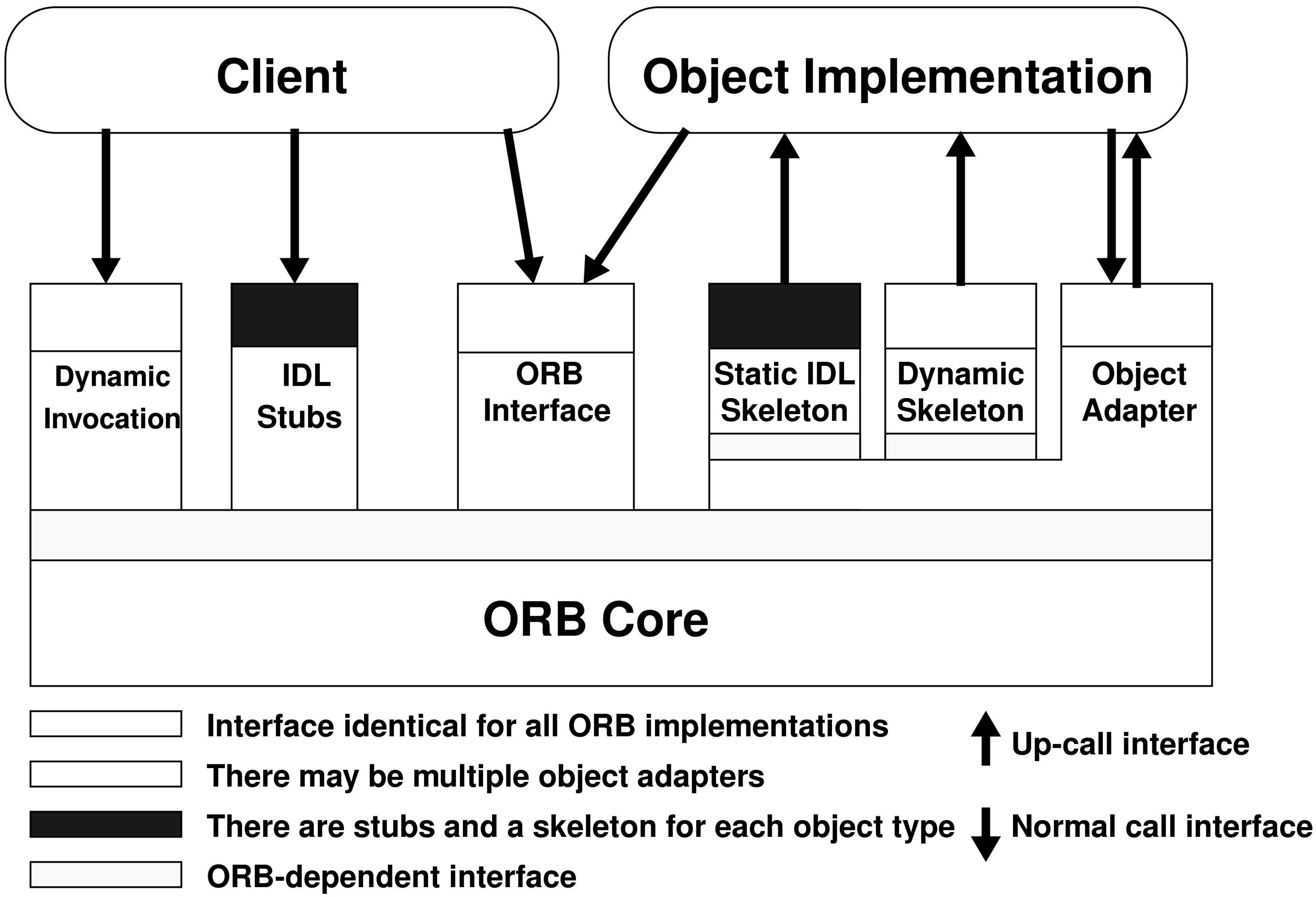
CORBA services

- Naming
- Event
- Notification
- Trading
- Security
- Property
- Life Cycle
- …

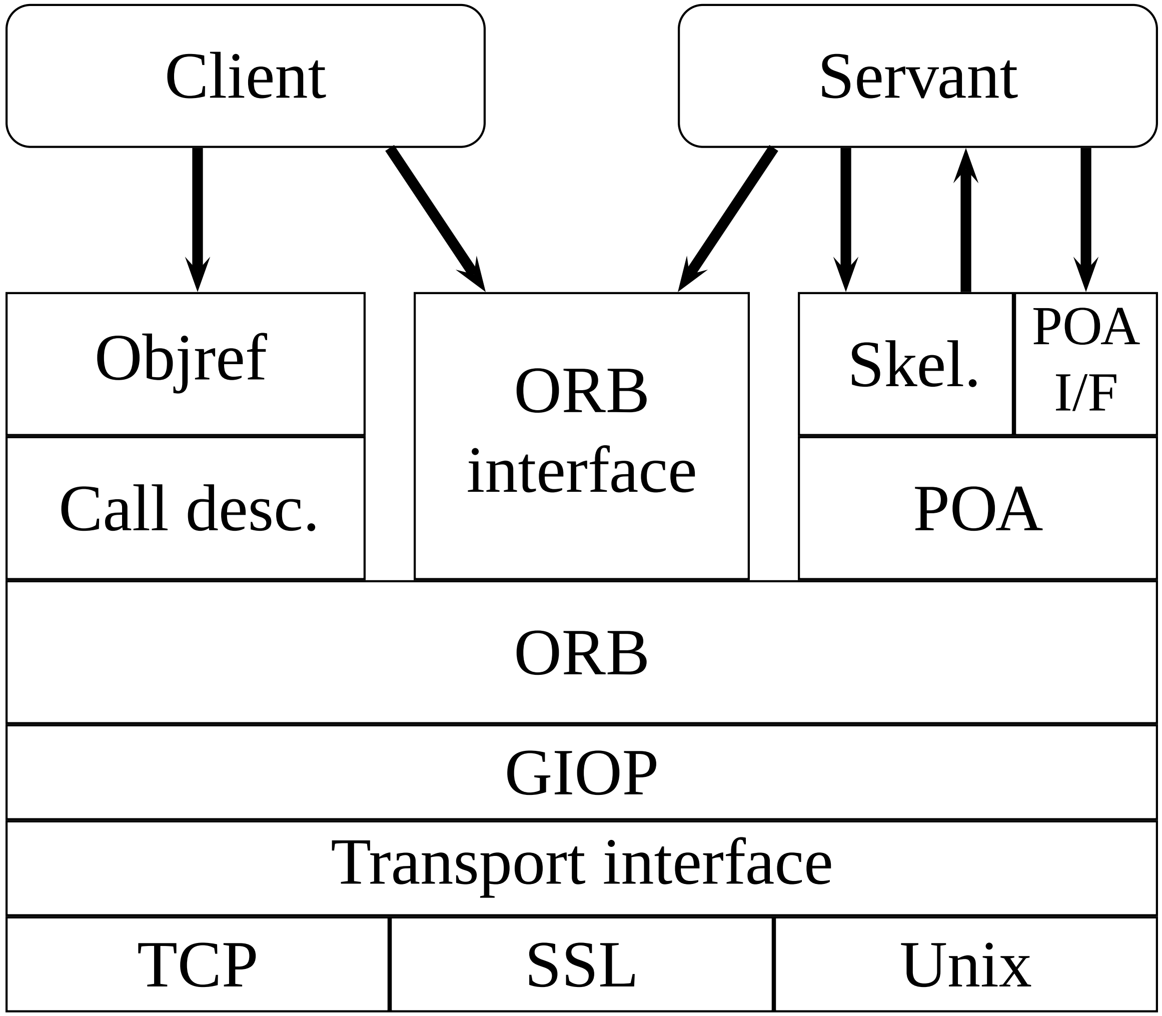
omniORB structure

1. ‘Conventional’ ORB overview
2. omniORB overview
3. Transport layer
4. CDR streams
5. GIOP
6. Object references
7. Identities
8. Object Adapters

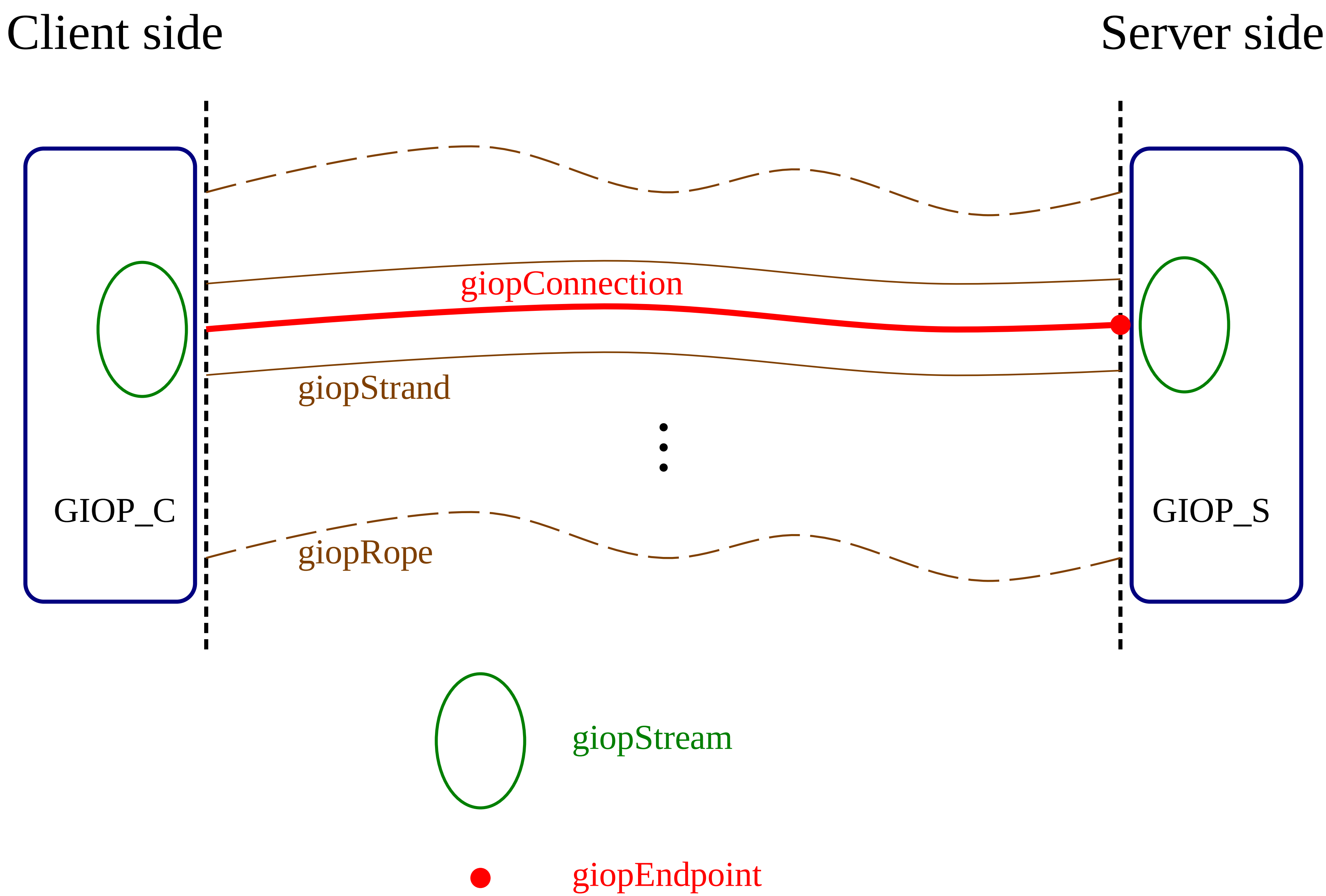
ORB overview



omniORB overview



Transport layer



Transport layer

- Set of abstract base classes:
  - `cdrStream` — marshalling of CORBA types.
  - `Strand` — a network connection between two address spaces.
  - `Rope` — a bundle of strands joining the same pair of address spaces.
  - `IOP_C` — client side of an inter-ORB protocol.
  - `IOP_S` — server side of an inter-ORB protocol.
- Specialisation to GIOP: `giopStream`, `giopStrand`, `giopRope`, `GIOP_C`, `GIOP_S`.

CDR streams

- CORBA data is transferred using Common Data Representation (CDR).
- Formats for basic types (numbers, strings, etc.)
- Formats for constructed types (structs, unions, sequences, etc.)
- Data alignment rules (always to natural boundary).
- Dual endianness (sender chooses).

CDR streams

- Abstract class `cdrStream`.
  - `giopStream`, `cdrMemoryStream`, `...`
- Virtual functions for filling / emptying buffers.
- Inline functions for marshalling into / out of buffers.
- Virtual functions for marshalling would be more flexible
  - Non-CDR marshalling (e.g. XML).
  - Enormous performance impact.
- Undecided if / how to do both in future. (Template trickery?)

GIOP transport

- `giopStream` knows how to drive a generic GIOP communication, with any GIOP version.
- `giopStreamImpl` implements a particular GIOP version (1.0, 1.1, 1.2).
  - Functions e.g. `marshalRequestHeader`, `sendSystemException`.
- Represented as a collection of function pointers in an object.
  - Avoids overhead of virtual function calls.
  - Makes a noticeable difference.

GIOP transport

- Abstract class `giopConnection` encapsulates a network connection.
- `tcpConnection`, `sslConnection`, etc.
- Responsible for
  - Reading / writing buffers of marshalled data.
  - Reading / writing application buffers.
  - ‘Select’ing for events (data available, data sent, timeout).
- Abstract class `giopAddress` represents / connects to a network address.
- Abstract class `giopEndpoint` accepts incoming connections.

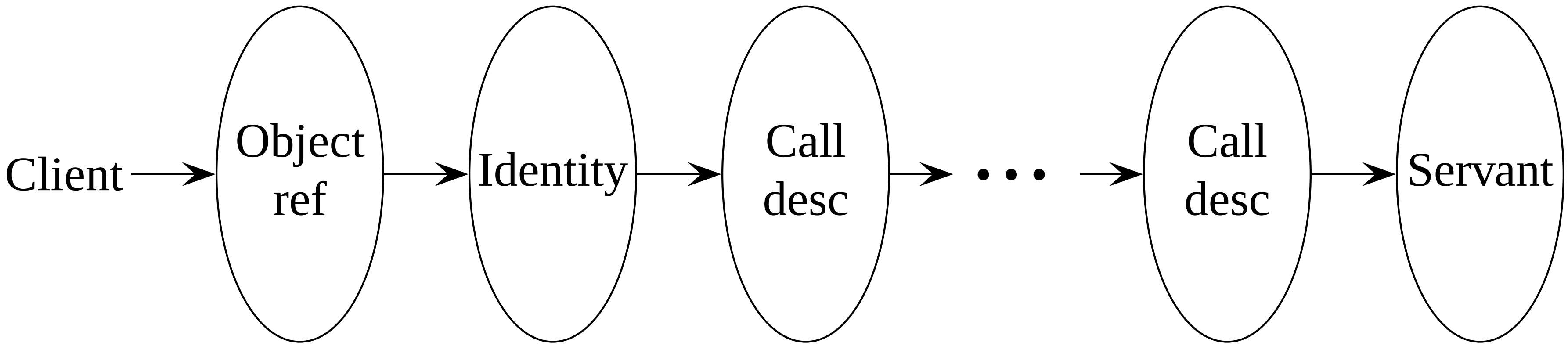
GIOP transport

- GIOP message header contains message length.
  - Have to know how much data you are going to send before you send it.
- GIOP 1.1 + support fragmentation.
  - Send fragments corresponding to marshalled buffers, so no need to pre-calculate complete message size.
- For GIOP 1.0, must know message size before transmission.
  - Start marshalling without knowing size.
  - If buffer fills up, divert to calculate size.

Code set conversion

- String and wstring are transformed on the fly.
- Native code set
  - Code set in use by the application
- Transmission code set
  - Code set in use on the wire.
  - ‘Negotiated’ at connection set-up time.
- If TCS understands NCS, marshal directly.
- Otherwise, marshal via Unicode.
- 8-bit char code sets use look-up tables.

End-to-end call



Object references

- Object reference represents a CORBA object on the client side.
- For interface `I`, IDL compiler generates object reference class `_objref_I`.
- Objref class provides methods according to the IDL interface.
- Real work is done by an *identity*, controlled with a *call descriptor*.

Identities

- An identity encapsulates knowledge of how to contact an object.
- Abstract base class `omniIdentity`.
- Currently four implementations:
  - `omniRemoteIdentity` — contacts object over the network.
  - `omniLocalIdentity` — object is local and activated.
  - `omniInProcessIdentity` — object is local but not activated, or not directly callable.
  - `omniShutdownIdentity` — dummy used during shutdown.

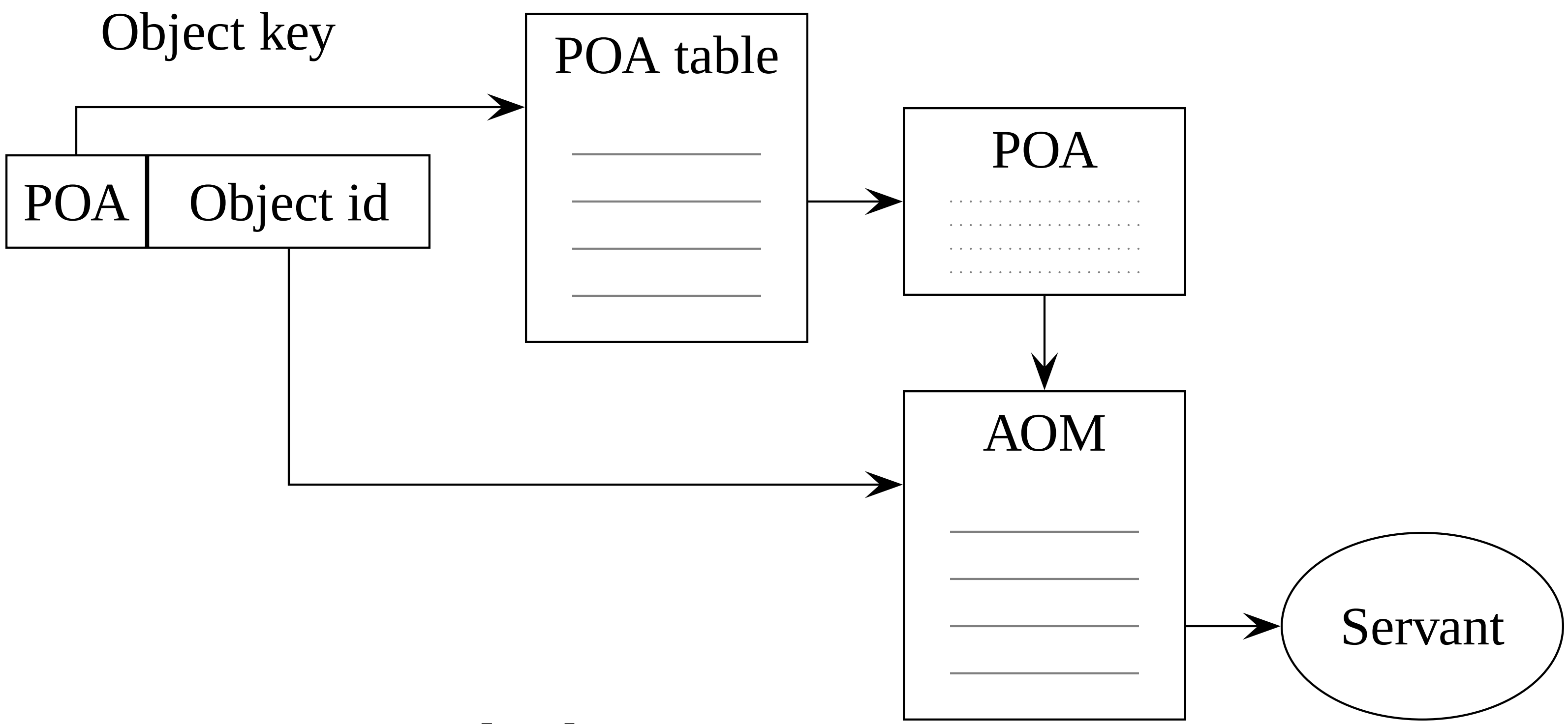
Call Descriptors

- Call descriptor knows operation specific details:
  - How to perform a local call.
  - How to marshal / unmarshal parameters.
  - Memory management rules.
- Call descriptor instance holds:
  - Parameters / return values.
  - Call timeout.
- One call descriptor class per operation signature.

Object adapters

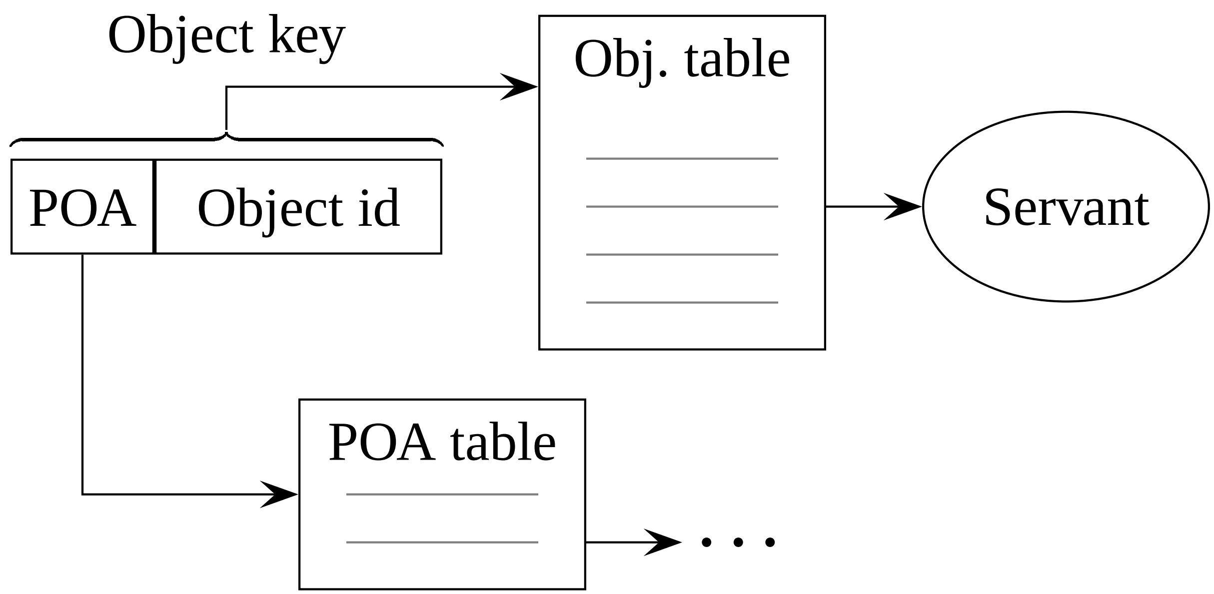
- Object adapters manage objects on the server side.
- Two object adapters in omniORB
  - BOA compatible with omniORB 2 BOA.
  - POA adheres to CORBA standard.
- Abstract base class `omniObjAdapter`.
  - Management functions
  - Dispatch functions

POA by the spec



- Two stage look-up
  - Find the POA.
  - Look in the POA’s Active Object Map (if it has one).

omniORB POA



- Global object table.
  - Dynamically resized open hash table.
  - Stores active objects for *all* POAs, and BOA.
  - One stage look-up for active objects.
- Only look to individual POA if object inactive.

Local (active) call dispatch

1. Client calls objref.
2. Construct call descriptor on stack.
3. Pass call descriptor to local identity.
4. Check object is still active and callable.
5. Pass local id / call descriptor to POA.
6. Check POA policies (threading, manager state).
7. Tell call descriptor to do a local call.
8. Unwind call stack.

- Single call chain minimises function call overhead.

Remote (active) call dispatch

1. `GIOP_S` receives an incoming call.
2. Create a ‘call handle’ on the stack.
3. Find local identity for object in object table.
4. Local id passes local id / call handle to POA.
5. Check POA policies.
6. Call servant’s `_dispatch` virtual function.
7. Create call descriptor on stack.
8. Unmarshal arguments.
9. Tell call descriptor to do a local call.
10. Marshal results.
11. Unwind the stack.

In process call dispatch

1. Client calls objref.
2. Construct call descriptor on stack.
3. Pass call descriptor to in process identity.
4. See if object is now active -> (maybe) replace with local id.
5. Find POA for object.
6. Create call handle.
7. Dispatch through POA with call handle.
8. …

Call dispatch principles

- Try to keep one call chain rather than separate function calls.
  - Avoids function call/return overhead.
  - Makes exception unwinding easier.
  - Harder to see execution flow.
- Try to avoid heap allocations.
  - Stack is *much* faster.
  - Easier to clean up when things go wrong.
- Minimise use of virtual functions.
  - Branching code can be better if few branches.

POA pain

- The POA has some awkward features
  - Servant Locator postinvoke.
  - Servant Activator incarnate/etherealize interaction.
  - Holding state.
  - Complex policy interactions.
  - Single thread policy.
  - Main thread policy.
- Try to optimise the fast, common case where nasty things aren’t being used.
- omniORB POA doesn’t look much like the spec implies it would.

Multi-threading

1. Overview
2. Issues
3. Thread use strategies

Multi-threading

- Everything happens in a multi-threaded environment.
- Small omnithread library to abstract away platform differences.
- Threading is hard
  - Race conditions.
  - Deadlocks.
  - Locking overhead.
  - Thread creation overhead.
  - Thread switching overhead.
  - Thread ‘crosstalk’ overhead.

Threading issues

- Locking/unlocking a mutex is expensive (200 ns – 1μs).
- Try to avoid doing too many locks
  - Minimise number of different mutexes.
  - Hold locks across function calls.
- Holding locks for a long time reduces concurrency.
  - Different locks for different subsystems.
  - Hold locks only for as long as necessary.
- A balancing act…

Race conditions / deadlocks

- Always a problem in concurrent systems.
- Strategy of holding locks across function calls makes it worse
  - Easy to forget to hold a lock.
  - Easy to forget you are holding a lock.
- Partial order on locks to avoid deadlock
  - Increases thread cross-talk.
- Assertions throughout code
  - On data, to help detect race conditions (and other errors).
  - On mutexes and condition variables.

Assertions

```
void
omniOrbPOA::dispatch(omniCallHandle& handle, omniLocalIdentity* id)
{
  ASSERT_OMNI_TRACEDMUTEX_HELD(*omni::internalLock, 1);
  OMNIORB_ASSERT(id);  OMNIORB_ASSERT(id->servant());
  OMNIORB_ASSERT(id->adapter() == this);

  handle.poa(this);

  enterAdapter();

  if( pd_rq_state != (int) PortableServer::POAManager::ACTIVE )
    synchronise_request(id);

  startRequest();

  omni::internalLock->unlock();
  ...
```

- Traced mutexes/conditions are slow.
  - Turn them off for releases.

Race condition optimism

- In some cases, structure code path to assume a race condition will not occur.
- Detect race conditions and pick up the pieces.

```
void
giopServer::notifyMrDone(giopMonitor* m, CORBA::Boolean exit_on_error)
{
  omni_tracedmutex_lock sync(pd_lock);

  if (!exit_on_error && !m->collection()->isEmpty()) {
    // We may have seen a race condition in which the Monitor is about
    // to return when another connection has been added to be monitored.
    // We should not remove the monitor in this case.
    if (orbAsyncInvoker->insert(m)) {
      return;
    }
    // Otherwise, we let the following deal with it.
  }
  m->remove();
  ...
```

Thread allocation

- Use threads to perform upcalls into application code.
- Two common strategies: thread per connection, thread pool.
- Thread per connection:
  - When a new network connection arrives, allocate a thread to it.
  - No thread switching along the call chain.
  - Does not scale to very large numbers of connections.
  - Does not handle multiplexed calls on a single connection.

Thread allocation

- Thread pool:
  - One thread watches many connections.
  - When a request arrives, pick a thread from the pool.
  - At least one thread switch along call chain.
  - select() etc. are slower than direct read().
  - Require concurrency control on connections.
  - Have to manage a queue if more requests than threads.
  - Scales to many more connections.

omniORB strategy

- Thread per connection if small number of connections.
  - Just before calling into application code, mark connection as ‘selectable’.
  - Another thread periodically looks for selectable connections and selects on them.
  - If a multiplexed request arrives, pick a thread from the ‘pool’ to handle it.
- Automatically transition to thread pool if too many connections arrive.

omniORB strategy

- Thread pool mode
  - Pool starts empty.
  - New threads started on demand, up to a limit.
  - Idle threads exit after a while.
- Connections under thread pool mode are selectable all the time.
- A common pattern is for a client to perform several calls in sequence.
  - Option to watch a connection for a while after a call.
  - Avoids thread switching overhead.

C++ and platform issues

1. The situation
2. omniORB policies
3. Examples

The situation

- C++
  - A very complex language.
  - It has evolved over time.
  - Nobody gets it right.
- Platforms
  - Unixes vary slightly.
  - Windows is gratuitously different.
  - Platforms like OpenVMS are weird.
  - omniORB developers do not have access to most platforms it runs on.

omniORB policies

- Compiler must support thread safe C++ exceptions.
- Don’t use STL.
  - omniORB pre-dated it.
  - Great variation in implementations.
  - Some robustness, performance concerns.
- Limit other template uses to simple things.
- Don’t use `dynamic_cast<>`, etc.
- Abstract away OS differences as much as possible.
- For really broken / difficult systems, maintain separate patches.

An ugly example

```
class A {
public:
  class B {
  public:
    B(int i) { ... }
  };
};
class C : A::B {
public:
  C(int i) : how to initialise A::B?
};
```

- Some compilers require `A::B(i)`; some require `B(i)`.
- Either is legal standard C++.
- Use `OMNIORB_BASE_CTOR(A::)B(i)`.

Windows…

```
int
SocketSetnonblocking(SocketHandle_t sock) {
# if !defined(__WIN32__)
  int fl = O_NONBLOCK;
  if (fcntl(sock,F_SETFL,fl) == RC_SOCKET_ERROR) {
    return RC_INVALID_SOCKET;
  }
  return 0;
# else
  u_long v = 1;
  if (ioctlsocket(sock,FIONBIO,&v) == RC_SOCKET_ERROR)
    return RC_INVALID_SOCKET;
  }
  return 0;
# endif
}
```

Micro-optimisations

1. String comparisons
2. Dynamic casting

String comparisons

- Profiling showed `strcmp()` to be surprisingly expensive.
- Save around 10% on local call time with inline `omni::strMatch()` function.
- Often strings being compared are the *same* string.
  - `omni::ptrStrMatch()` compares pointers first.
  - Work hard to arrange for strings to be re-used to make the most of this.

Dynamic casting

- Often need to dynamically cast base pointer to derived class.
- omniORB cannot use `dynamic_cast<>` since not all compilers support it.
- For omniORB 4.0, we thought we would use `dynamic_cast<>` where available.
- Performance testing showed that it takes 1 to 20 times longer for `dynamic_cast<>` than a scheme with virtual functions.
- Various schemes in use…

Dynamic casting

```
void*
CosNaming::_objref_NamingContextExt::_ptrToObjRef(const char* id)
{
  if( id == CosNaming::NamingContextExt::_PD_repoId )
    return (CosNaming::NamingContextExt_ptr) this;
  if( id == CosNaming::NamingContext::_PD_repoId )
    return (CosNaming::NamingContext_ptr) this;
  if( id == CORBA::Object::_PD_repoId )
    return (CORBA::Object_ptr) this;

  if( omni::strMatch(id, CosNaming::NamingContextExt::_PD_repoId) )
    return (CosNaming::NamingContextExt_ptr) this;
  if( omni::strMatch(id, CosNaming::NamingContext::_PD_repoId) )
    return (CosNaming::NamingContext_ptr) this;
  if( omni::strMatch(id, CORBA::Object::_PD_repoId) )
    return (CORBA::Object_ptr) this;

  return 0;
}
```

omniORB for Python

1. Overview
2. Data marshalling
3. ORB interactions
4. Calls between C++ and Python
5. Threading

Overview

- omniORBpy is omniORB's mapping to Python.
- Implemented on top of C++ omniORB.
  - Python has a clean C API.
  - Uses omniORB internal features.
  - Small number of changes to omniORB core to support it. (Nothing Python specific, though.)
  - Majority of code is C++, not Python.
- Adheres to the standard Python language mapping.

Data marshalling

- For C++, the IDL compiler emits C++ code to marshal each type.
- For Python, it emits a *type descriptor*.
- C++ code marshals Python data structures according to the descriptors.
- Descriptors are based on CORBA TypeCodes.
  - Pack TypeCode information into Python tuples.
  - Support for Any is almost free.
- Python is dynamically typed.
  - Must check the types of all marshalled values, even for local calls.

Data marshalling

```
// IDL
struct Example {
  long a;
  string b;
};



# Python
class Example:
    _NP_RepositoryId = "IDL:Example:1.0"
    def __init__(self, a, b):
        self.a = a
        self.b = b

_d_Example = (omniORB.tcInternal.tv_struct, Example,
              Example._NP_RepositoryId, "Example",
              "a", omniORB.tcInternal.tv_long,
              "b", (omniORB.tcInternal.tv_string,0))
```

ORB interactions

- All C++ ORB, POA, etc. functions have to be wrapped into Python.
  - Written by hand (no SWIG, etc.).
  - Often quite a simple mapping.
  - Sometimes quite complex.
- Python equivalents of C++ objects keep a ‘twin’ referring to the C++ version.
- Small amounts of Python code to hide some details.

ORB interactions

- Python equivalents of object references, servants, etc.
- Designed to look like normal C++ versions to the ORB core.
- In some cases (creating an object reference, unmarshalling one, etc.), omniORBpy code duplicates ORB core code with small differences.
- Have to be careful when C++ and Python application code are in the same process…

Cross-language calls

- When omniORB creates an object reference, it looks to see if it is a local object.
- If so, it creates a local identity so calls are made directly on the servant.
  - Disaster if client is C++ and servant is Python!
  - ORB core asks a servant if it is ‘compatible’ with an objref before creating a local id.
- In process identity performs calls via a memory buffer.
  - In future, may permit interleaved marshal / unmarshal to save space.

Python threading

- Python has a global interpreter lock.
  - Must release lock whenever doing a blocking call.
  - Must acquire lock before calling into Python.
- Python uses some per-thread state.
  - Must create state for threads started by C++.
  - Creating state is expensive.
  - Maintain a cache of thread states.

Summary

- Implementing an ORB is pretty hard.
  - Complex functionality.
  - Complex specification.
  - Complex language mappings.
  - Portability issues.
- Making it fast is even harder.
  - Think outside of the obvious.
  - Balance concurrency control.
  - Minimise heap allocations.
  - Minimise virtual functions.
  - Consider nesting function calls.
  - Use profiling.