Projektpraktikum

Performance Analysis Ada95/AnnexE, Java/RMI, CORBA

Ronald Blaschke
Universität Salzburg
Institut für Computerwissenschaften

Jakob-Haringer Straße 2
5020 Salzburg
Austria

Projektpraktikum: Performance Analysis Ada95/AnnexE, Java/RMI, CORBA

by Ronald Blaschke
Universität Salzburg
Institut für Computerwissenschaften

Jakob-Haringer Straße 2
5020 Salzburg
Austria

Published $Date: 2000/11/07 21:21:21 $

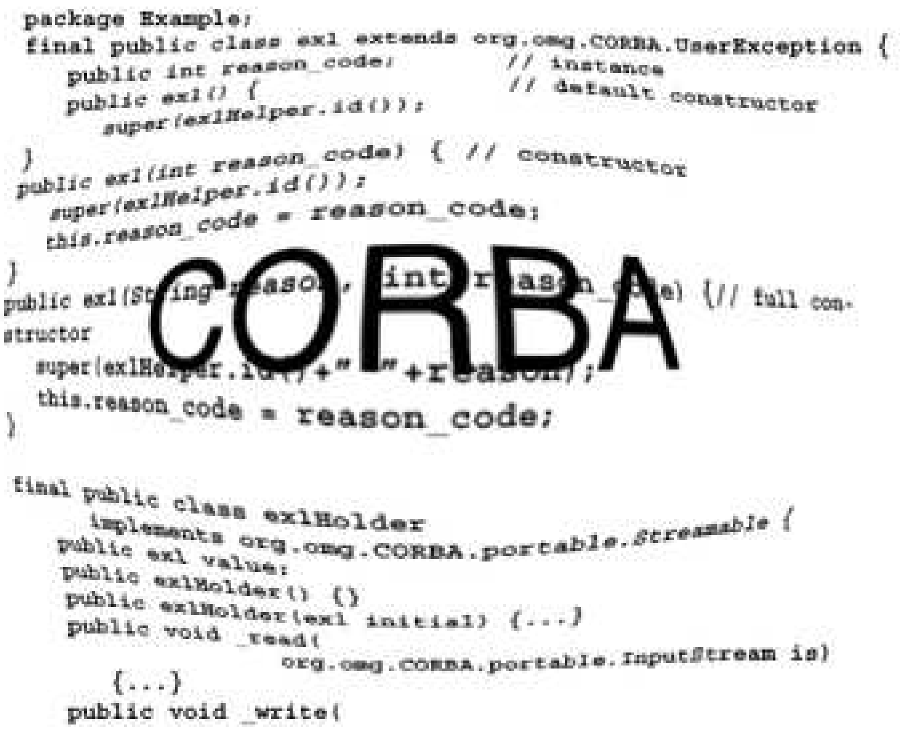
Revision History

Revision $Revision: 1.1 $	$Date: 2000/11/07 21:21:21 $	Revised by: rb
Added std dev error bars.		
Revision $Revision: 1.1 $	$Date: 2000/11/07 21:21:21 $	Revised by: rb
Added/corrected explanations (special thanks to Christof Meerwald!).		
Revision 0.6	2000/07/30 18:23:45	Revised by: rb
General review with minor changes.		
Revision 0.5	2000/05/03 18:16:33	Revised by: rb
Star Architecture completed.		
Revision 0.4	2000/04/12 02:09:19	Revised by: rb
.		
Revision 0.3	30 Mar 2000	Revised by: rb
Compound Data Types section added.		
Revision 0.2	24 Feb 2000	Revised by: rb
Basic Data Types Section.		
Revision 0.1	15 Feb 2000	Revised by: rb
Document Outline.		

Table of Contents

List of Figures

I. CORBA



Acknowledgements

Many thanks to my advisor *Dr. Werner Pohlmann*. Also many thanks to my colleague *Josef Aichhorn,* who worked at the Ada95/AnnexE part, for his review and his valueable feedback. Finally, a special thank you to *Christof Meerwald* for sharing his insights into CORBA with me.

Chapter 1. About CORBA

In short, CORBA (Common Object Request Broker Architecture) is a middleware to distribute objects over different platforms and different programming languages, and is specified by the OMG ([4]).

What follows is a very basic introduction. For a more detailed description consult the CORBA specification ([5]) or a good book, like *Advanced CORBA programming with C++* ([1]).

First, we need an interface definition written in IDL (Interface Definition Language). This is conceptually similar to a Java interface. This interface is then compiled with an IDL compiler into a stub and a skeleton for the target language. A stub is a proxy for the actual object and handles data marshalling and transfer to the actual object, wherever it may be. The skeleton also contains marshalling and data transfer code, but must be extended by the operations the object is supposed to perform.

To use the object we need a server that creates the object and somehow transfers the reference to it to the clients. The easiest way is via a file on a shared file system. The clients then create the proxy object and can use it as if it is the actual object (with minor modifications, like additional CORBA exceptions).

Chapter 2. Test Environment

Before we start the actual tests we have to decide for a particular test environment. This includes decisions for Hardware and Software.

Hardware

All tests were run on

- Sparc 20, 128 Megabytes TMS390Z55

unless otherwise noted.

Software

The natural choice for a comparison with Ada95's AnnexE would have been a CORBA implementation for Ada. There is already an OMG (Object Management Group) Language Mapping for Ada ([6]) available. However, Ada is not very popular in the Freeware world and therefore very few free CORBA implementations exist. The only ones I found are in early development stage ([9]), or rely on an implementation in another language ([10]).

In order not to introduce another programming language I decided to use a Java implementation. There are several free ([14], [15]) and free for non-commercial use ([11]) implementations available. I decided for ORBacus ([11]) because of the good documentation and extensive thread model support. See *CORBA Comparison Project* ([12]) for a comparison with other ORBs.

So, our software environment included

- JDK 1.2
- ORBacus 3.3[1]

Comparing Test results of Java, which usually is converted into bytecode that runs in a virtual machine, and Ada, a compiling language, may be difficult because the program in the virtual machine runs slower. But I don't think this will be a big issue here because

- we use no highly computational code and
- modern virtual machines with JIT (Just In Time) compilers are reasonable fast.

If I think that language specifica are responsible for major differences I will explain them.

Data Collection

The tests were run on preloaded machines, id est machines that already have processes running. These are the usual network software and a process (rc5des) at priority level 19, which is the lowest priority. My processes were run at priority level 0, which is the default. Also, regular network traffic was allowed in the test environment. This should satisfy the need for "real life data," per contra to optimal performance.

Unless otherwise noted, the following applies. Each test was run 100 times and the average was taken. This is the "Total Time" in seconds. A test consisted of 1000 remote object calls, thus the "Total Time" value can also be interpreted as the average time (mean), in milliseconds, for a single call.

The resulting data is compressed into the average time and the standard deviation. I will use the standard deviation for error bars. Assuming a "bell-shape" and using an empirical rule [13] approximately 68% of our data should be in this interval. Double this interval for 95%, or triple it for 99.7%.

Notes

1. As of Feb 25, 2000 the new Version 4 became available, which is CORBA 2.3 compliant. The statements of this report should hold even for the new POA (Portable Object Adapter), although this is *not* verified.

Chapter 3. The Test Programs

For my analysis I had one main server that is used for performing the tests or as a communication point between the clients (e.g. exchange of object references in a mesh). For each test I had a different client, but they all look quite similar. Therefore, I will outline them below and plug in the missing parts in the corresponding sections.

The Server

First, we have to set ORBacus as the used ORB (this is only necessary for JDK >=1.2). We initialize the ORB and set the concurrency model to "thread per request with a thread pool of 512 threads," just like the Ada95/AnnexE implementation *Glade* does. Then we incarnate a factory object and write its reference to the file `ObjFactory.ref`. This factory object will be used to create other objects in the server. Finally, we activate the implementation.

Figure 3-1. Server.java

```
package Perf;

public class Server
{
    public static void main(String argv[])
    {
        // set ORBACUS as ORB
        java.util.Properties props = System.getProperties();
        props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
        props.put("org.omg.CORBA.ORBSingletonClass",
                  "com.ooc.CORBA.ORBSingleton");
        System.setProperties(props);

        // initialize ORB and BOA
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(argv, props);
        org.omg.CORBA.BOA boa =
            ((com.ooc.CORBA.ORB)orb).BOA_init(argv, props);
        ((com.ooc.CORBA.ORB)orb).conc_model(
            com.ooc.CORBA.ORB.ConcModel.ConcModelThreaded);
        ((com.ooc.CORBA.BOA)boa).conc_model(
```

```
            com.ooc.CORBA.BOA.ConcModel.ConcModelThreadPool);
        ((com.ooc.CORBA.BOA)boa).conc_model_thread_pool(512);

        // create a factory object
        ObjFactory_impl factory = new ObjFactory_impl();

        // write factory reference to file
        try {
            String ref = orb.object_to_string(factory);
            String refFile = "ObjFactory.ref";
            java.io.PrintWriter out = new java.io.PrintWriter(
              new java.io.FileOutputStream(refFile));
            out.println(ref);
            out.flush();
            out.close();
        }
        catch (java.io.IOException e) {
            System.err.println("error: " + e.getMessage());
            System.exit(1);
        }

        // run server
        boa.impl_is_ready(null);
    }
}
```

The Clients

The clients start all similar. First, we initialize the ORB. Then we read the factory reference from the file `ObjFactory.ref` and create an object reference. Then we get references to objects in the server, namely Barrier (which is used to synchronize clients) and Round Trip Time test objects (see the section called *The Server Objects*). After that, we read the test parameters from the command line, perform the test and print the used time. Details for the tests are shown in the corresponding sections.

Figure 3-2. xxxClient.java

```
package Perf;
```

```
public class xxxClient
{
    public static void main(String argv[])
    {
        // set ORBACUS as ORB
        java.util.Properties props = System.getProperties();
        props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
        props.put("org.omg.CORBA.ORBSingletonClass",
                  "com.ooc.CORBA.ORBSingleton");
        System.setProperties(props);

        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(argv, props);

        // load factory reference from file
        String ref = null;
        try {
            String refFile = "ObjFactory.ref";
            java.io.BufferedReader in = new java.io.BufferedReader(
                new java.io.FileReader(refFile));
            ref = in.readLine();
        }
        catch (java.io.IOException e) {
            System.err.println("error: " + e.getMessage());
            System.exit(1);
        }

        org.omg.CORBA.Object obj = orb.string_to_object(ref);
        ObjFactory factory = ObjFactoryHelper.narrow(obj);

        Barrier b = null;
        while (b == null) {
            try {
                b = factory.getBarrier();
            }
            catch (Exception e) {}
        }

        // read parameters from commandline

        // run tests
```

```
        // print required RTT
    }
}
```

The Server Objects

There are a few CORBA objects for the tests. They are outlined below.

The Object Factory

The Object Factory is the start point of the server. All other objects are created by this factory. Some are singleton, which means they are created at startup and the reference to the very same object is returned every time. It is a well known pattern and a very easy way to create multiple objects in the server.

To communicate references between clients (necessary for the mesh) we allow registration, deregistration and lookup of references.

Most tests use the main RTT (Round Trip Time) object (see below for description), but for the number of methods in an object test (see *Number of Methods*) we need separate objects. They can be created with the `createMxxx()` methods, where xxx is the number of methods.

Figure 3-3. IDL for the Object Factory

```
module Perf {
  interface ObjFactory {
    Barrier getBarrier();  // Singleton
    RTT getRTT();  // Singleton
    RTT createRTT();

    void registerRTT(in long id, in RTT obj);
    void deregisterRTT(in long id);
    RTT lookupRTT(in long id);

    RTTMethods1 createM1();
    ...
}
```

The Barrier

The Barrier is used to synchronize the clients at the beginning of the tests. At the synchronization point, each client calls the `barrier()` with the total number of clients that will call it. The call returns as soon as the given number of clients reach this call.

Figure 3-4. IDL for the Barrier

```
module Perf {
  interface Barrier {
    void barrier(in long howMany);
  };
}
```

RTT Object

The RTT object is used for round trip time measurements of various data types. This includes the basic data types (void, long, float, boolean) and the compound data types (array, struct). For each test we have a synchronous and an asynchronous version. The object is implemented with empty function bodies, since we are only measuring round trip times.

Figure 3-5. IDL for the RTT Object

```
module Perf {
  interface RTT {
    void sync_VV();
    oneway void async_V();

    void sync_LL(in long din, out long dout);
    oneway void async_L(in long din);
    ...
    void sync_aLaL1(in RTTArrays::longArray1 din,
                    out RTTArrays::longArray1 dout);
    oneway void async_aL1(in RTTArrays::longArray1 din);
    ...
    void sync_sLsL1(in RTTStructs::longStruct1 din,
                    out RTTStructs::longStruct1 dout);
```

```
...}}
```

Chapter 4. Simple Interaction

“Simple Interaction” refers to the client/server architecture: A single server and a single client are used. These results are necessary to understand and predict the more complex interactions.

Basic Data Types

The first analysis concerns the basic data types, namely void, boolean, int and float. Both, synchronous and asynchronous performance is tested.

Method

For this we simply call the method for the corresponding data type 1000 times.

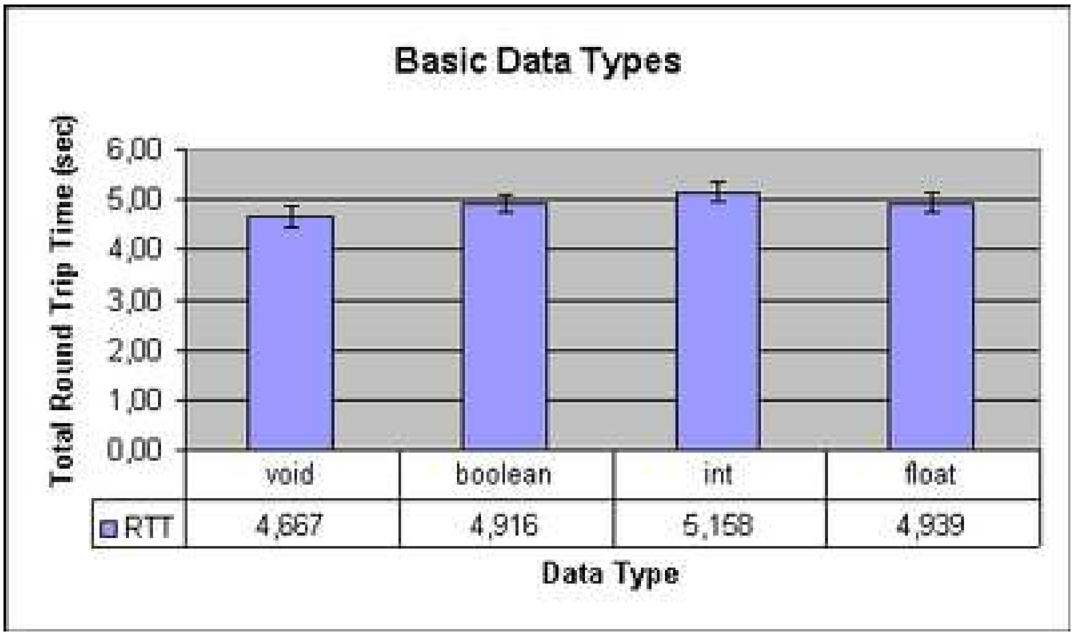
Figure 4-1. RTT Calls for the Basic Data Types

```
...
long t1 = 0;
long t2 = 0;
if (method.equals("sync_VV")) {
    t1 = System.currentTimeMillis();

    for (int i=0; i<1000; i++)
        rtt.sync_VV();

    t2 = System.currentTimeMillis();
}
...
else if (method.equals("sync_LL")) {
    org.omg.CORBA.IntHolder tmp_L =
        new org.omg.CORBA.IntHolder();

    t1 = System.currentTimeMillis();

    for (int i=0; i<1000; i++)
        rtt.sync_LL(42, tmp_L);

    t2 = System.currentTimeMillis();
```

```
}
...
else {
    System.out.println("Unknown method: " + method);
}

System.out.println(t2-t1);
```

Results

Figure 4-2. Basic Data Types RTT (Synchronous Calls)



As can be seen in Figure 4-2 there are some differences in transferring the basic data types. First we should take a look at the transferred data.

void

No data is transferred. This is the minimum call time.

boolean

Transfer of 1 byte, as specified in the GIOP (General Inter-ORB Protocol, [5]).

int

Transfer of 4 bytes, as specified in the GIOP.

float

Transfer of 4 bytes, as specified in the GIOP.

The transfer of void, boolean and int work as expected: Marshaling and transfer of more data takes more time. The difference between int and float remains unexplained because there is no obvious difference in GIOP. Most likely, the byte access to ints is more expensive than to floats, but this is a speculation.

Figure 4-3. Basic Data Types RTT

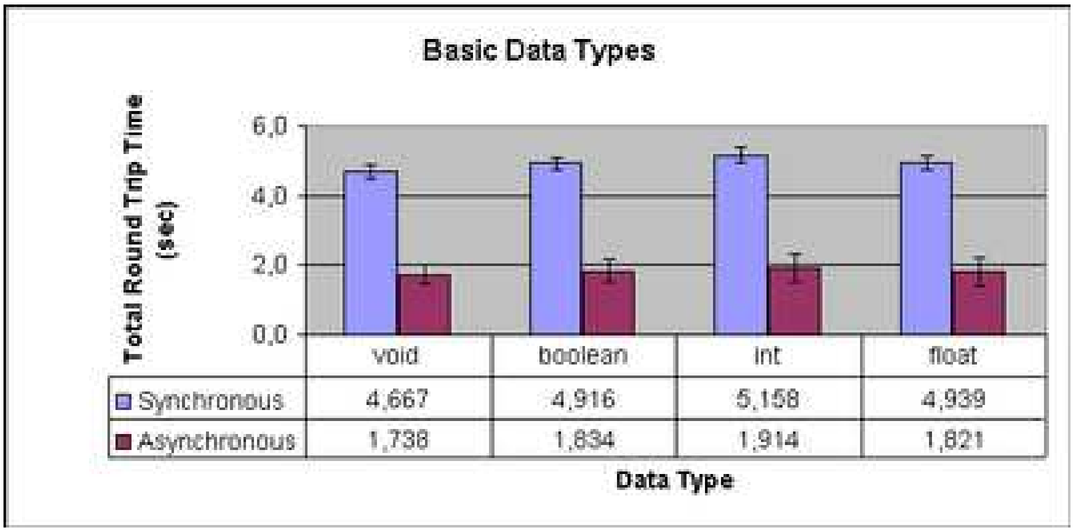


Figure 4-3 shows a comparison between synchronous and asynchrounous calls. Not surprisingly, the asynchronous calls are quite faster than their synchronous counterparts.

The standard deviation is between 200ms and 300ms for all tests.

> **Note:** This comparison is not absolutely fair since we return the same amout of data in the synchronous version and nothing in the asynchronous.

Returning something in the synchronous version seemed more practical.

Compound Data Types

In this section we will take a closer look at the compound data types array and structure. Note that from this point forward we will only deal with synchronous calls.

Array

Method

Arrays will only be tested with ints. As for the simple data types, we simply call a method 1000 times, with an increasing number of array elements. Note that again, we transfer the same amount of data forth and back.

Figure 4-4. RTT Calls for Arrays

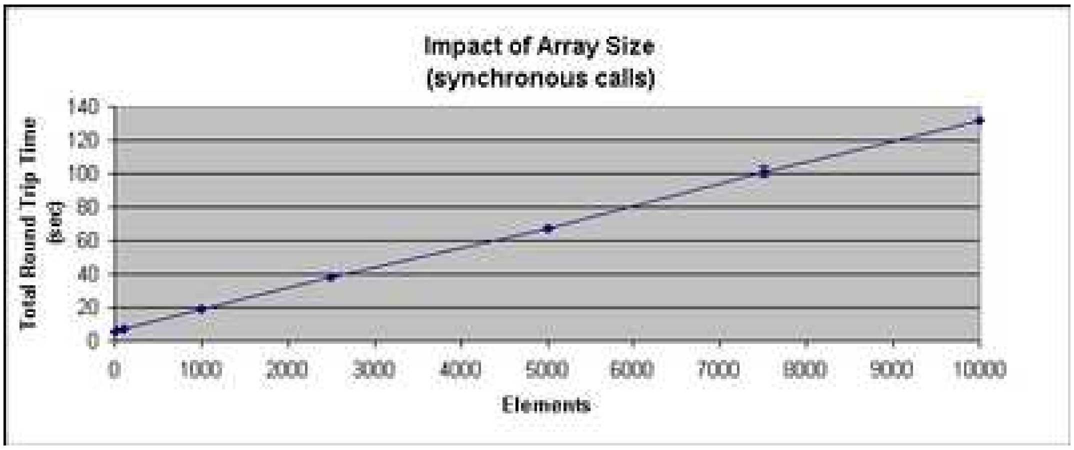
```
...
else if (method.equals("sync_aLaL1")) {
    int[] din = new int[1];
    Perf.RTTArraysPackage.longArray1Holder dout =
        new Perf.RTTArraysPackage.longArray1Holder();
    t1 = System.currentTimeMillis();

    for (int i=0; i<1000; i++)
        rtt.sync_aLaL1(din, dout);

    t2 = System.currentTimeMillis();
}
else if (method.equals("async_aL1")) {
    int[] din = new int[1];

    t1 = System.currentTimeMillis();

    for (int i=0; i<1000; i++)
```

```
            rtt.async_aL1(din);

        t2 = System.currentTimeMillis();
    }
    ...
```

Results

Arrays contain just the plain blocks attached to each other, as specified by the GIOP (General Inter-ORB Protocol, [5]). Thus, Figure 4-5 shows exactly what was expected: A linear correlation between the number of elements and the round trip time.

Figure 4-5. Size of Arrays



Structure

Method

The calls look as before. We only put ints into the structure.

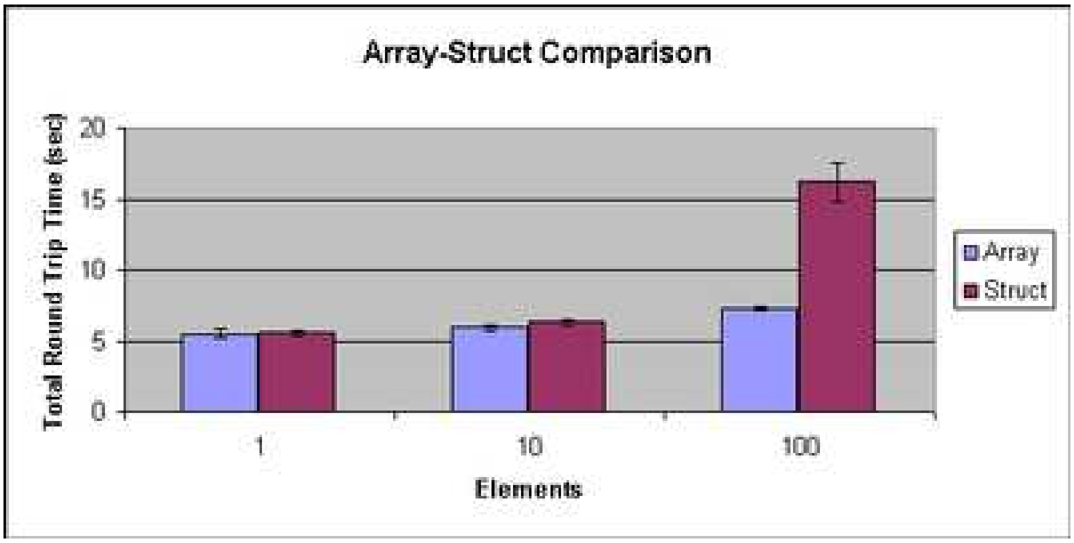
Figure 4-6. RTT Calls for Structures

```
...
else if (method.equals("sync_sLsL1")) {
    Perf.RTTStructsPackage.longStruct1 din =
        new Perf.RTTStructsPackage.longStruct1();
    Perf.RTTStructsPackage.longStruct1Holder dout =
        new Perf.RTTStructsPackage.longStruct1Holder();
    t1 = System.currentTimeMillis();

    for (int i=0; i<1000; i++)
        rtt.sync_sLsL1(din, dout);

    t2 = System.currentTimeMillis();
}
...
```

Results

My intuition suggested that structures are encoded similar to arrays, simply each member after the other. Taking a look at Figure 4-7 is therefore a little surprising. At 100 elements we see a large stray between arrays and structures.

Figure 4-7. Array-Struct Comparison



First, a look at the standard might provide some clue. It says the following.

> The components of a structure are encoded in the order of their declaration in the structure. Each component is encoded as defined for its data type.

It seems we have to dig a little bit deeper, into the source code, to find out what is going on. The following code is taken from a `Helper` class for structures, generated by the IDL compiler.

```
final public class longStruct10Helper
{
    ...
    private static org.omg.CORBA.TypeCode typeCode_;

    public static org.omg.CORBA.TypeCode
    type()
    {
        if(typeCode_ == null)
        {
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
            org.omg.CORBA.StructMember[] members =
              new org.omg.CORBA.StructMember[10];

            members[0] = new org.omg.CORBA.StructMember();
            members[0].name = "d0";
            members[0].type =
              orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_long);

            members[1] = new org.omg.CORBA.StructMember();
            members[1].name = "d1";
            members[1].type =
              orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_long);

            ...

            type-
Code_ = orb.create_struct_tc(id(), "longStruct10",
                                             members);
        }

        return typeCode_;
    }
    ...
```

```
}
```

And below the counterpart for arrays.

```
final public class longArray10Helper
{
    ...
    private static org.omg.CORBA.TypeCode typeCode_;

    public static org.omg.CORBA.TypeCode
    type()
    {
        if(typeCode_ == null)
        {
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
            typeCode_ = orb.create_alias_tc(id(), "longAr-
ray10", orb.create_arr\
ay_tc(10, orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_long)));
        }

        return typeCode_;
    }
    ...
}
```

This type code creation introduces some overhead, but the result is cached in a static variable, which means that it is necessary only once.

Just as the difference between ints and floats, this effect remains unexplained. However, we can speculate about the cause. Below you see the `read()` methods in the `Helper` classes for structures and arrays, respectively. Structures are read element by element, whereas arrays are read as a whole. This might be the cause for the difference, but is *not* verified.

```
    public static longStruct10
    read(org.omg.CORBA.portable.InputStream in)
    {
        longStruct10 val = new longStruct10();
        val.d0 = in.read_long();
        val.d1 = in.read_long();
        val.d2 = in.read_long();
```

```
        val.d3 = in.read_long();
        val.d4 = in.read_long();
        val.d5 = in.read_long();
        val.d6 = in.read_long();
        val.d7 = in.read_long();
        val.d8 = in.read_long();
        val.d9 = in.read_long();
        return val;
    }

    public static int[]
    read(org.omg.CORBA.portable.InputStream in)
    {
        int[] val;
        int len0 = 10;
        val = new int[len0];
        in.read_long_array(val, 0, len0);
        return val;
    }
```

However, this has very little effect on structures with few, say 10, members. As structure with many members, say 100, performs quite poorly, but a structure with that many members is not likely to occur at all.

Number of Methods

Now we will take a closer look at the influence of the number of methods in an object.

Method

Here we have quite a number of methods to call. Instead of hardcoding them we use the Java Reflection API to create references to the methods and then call them.

Figure 4-8. MethodClient.java

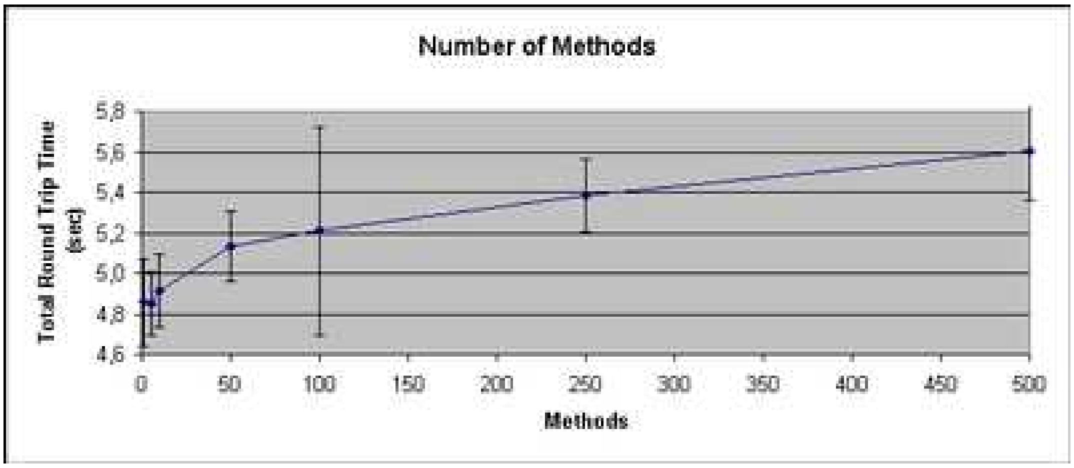
```
        ...
```

```
        long t1 = 0;
        long t2 = 0;
        if (method.equals("sync_VV_1")) {
            RTTMethods1 methods = factory.createM1();
            int num = 1;
            java.lang.reflect.Method[] m =
                        new java.lang.reflect.Method[num];

            for (int j=0; j<num; j++) {
                try {
                    m[j] =
                        meth-
ods.getClass().getDeclaredMethod("sync_VV_"+j, null);
                }
                catch(Exception e) {
                    System.out.println("Error while invok-
ing: "+e);
                }
            }

            try {
                t1 = System.currentTimeMillis();
                for (int i=0; i<loops; i++) {
                    for (int j=0; j<num; j++) {
                        m[j].invoke(methods, null);
                    }
                }
                t2 = System.currentTimeMillis();

            }
            catch(Exception e) {
                System.out.println("Error while invok-
ing: "+e);
            }
        }
        ...
        System.out.println(t2-t1);
```

Results

The number of methods in an object should have very little influence. But let's take a look at the actual data first.

Figure 4-9. Number of Methods RTT



As can be seen above, the round trip time increases about 7% for 100 methods, and about 15% for 500 methods. Why does this happen?

Let's take a look at the GIOP (General Inter-ORB Protocol, [5]) again.

```
module GIOP { // IDL extended for version 1.1 and 1.2
    struct RequestHeader_1_2 {
        unsigned long request_id;
        octet response_flags;
        octet reserved[3];
        TargetAddress target;
        string operation;
        IOP::ServiceContextList service_context;
        // Principal not in GIOP 1.2
    };
};
```

Not surprisingly, the operation is encoded as a string. This string is then looked up in a table, via binary search, and the index is used for a switch, in which each case invokes the correct method. The actual code for an interface with one method is shown below.

```
    final public void
```

```
invoke(org.omg.CORBA.ServerRequest _ob_req)
{
    String _ob_op = _ob_req.op_name();
    final String[] _ob_names =
    {
        "sync_VV_0"
    };

    int _ob_left = 0;
    int _ob_right = _ob_names.length;
    int _ob_index = -1;

    while(_ob_left < _ob_right)
    {
        int _ob_m = (_ob_left + _ob_right) / 2;
        int _ob_res = _ob_names[_ob_m].compareTo(_ob_op);
        if(_ob_res == 0)
        {
            _ob_index = _ob_m;
            break;
        }
        else if(_ob_res > 0)
            _ob_right = _ob_m;
        else
            _ob_left = _ob_m + 1;
    }

    switch(_ob_index)
    {
    case 0: // sync_VV_0
        _OB_op_sync_VV_0(_ob_req);
        return;
    }

    throw new org.omg.CORBA.BAD_OPERATION();
}
```

The above code segment (binary search with 1 element) takes about 55μs to execute. The 500 methods equivalent (binary search with 500 elements) about 750μs. A single method call takes 4.9ms on average for an object with 1 method (see Figure 4-9). With these numbers we can now try to predict the time to call an object with 500 methods. We take the single method call time, substract the lookup time for the method and add the lookup time for 500 methods, i.e. 4.9ms - 50μs + 750μs = 5.6ms. The actual measured value was 5.6ms, as can be seen in Figure 4-9.

In conclusion, the additional time needed seems to come from the binary search of the called methods in the array of all possible methods. Unless other system limitations kick in, the required time will continue to grow with O(ld n) of the binary search.

As discussed for structures before, an object with 500+ methods is not likely to occur and therefore should be a small issue in real life.

Number of Objects

This section deals with the influence of the number of objects in a server.

Method

First, we create a number of objects in the server. Then we iterate through calling them.

Figure 4-10. ObjClient.java

```
...
long t1 = 0;
long t2 = 0;
if (method.equals("sync_VV")) {
    RTT[] rtts = new RTT[objects];

    for (int i=0; i<objects; i++) {
        rtts[i] = factory.createRTT();
    }

    t1 = System.currentTimeMillis();
    for (int i=0; i<loops; i++) {
        // rotate through objects
        for (int j=0; j<objects; j++) {
            rtts[j].sync_VV();
        }
```

```
        }
        t2 = System.currentTimeMillis();
    }
    ...
    System.out.println(t2-t1);
```

Results

The standard ([5]) specifies the following for object references:

> In general, GIOP profiles include at least these three elements:
>
> 1. The version number of the transport-specific protocol specification that the server supports.
> 2. The address of an endpoint for the transport protocol being used.
> 3. An opaque datum (an *object_key*, in the form of an octet sequence) used exclusively by the agent at the specified endpoint address to identify the object.

Important is that the server may freely specify the object key, which must only be understood by him to access the object.

A simple implementation might even take the pointer to the object as the object key, thus nullify the influence of accessing a big number of it (cf the section called *Number of Methods*).

My findings are shown below.

Figure 4-11. Number of Objects RTT (1-1000)

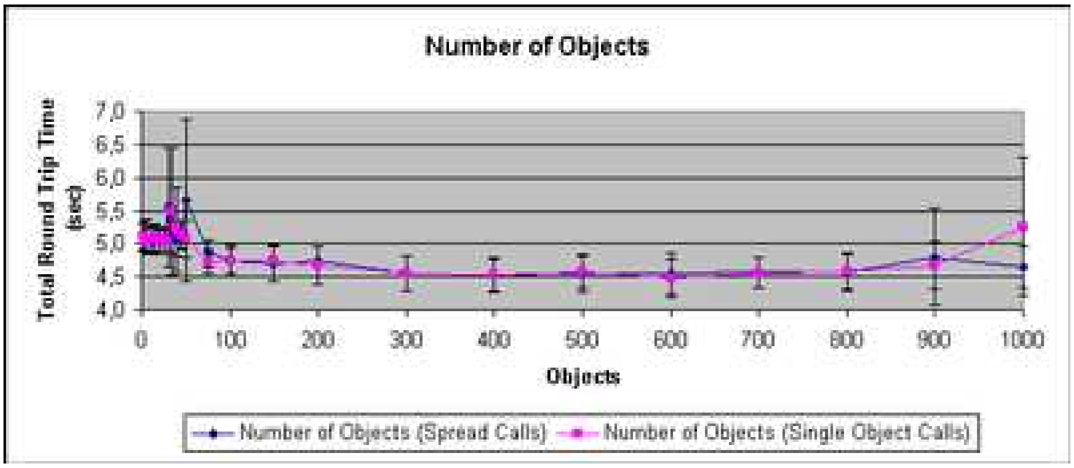
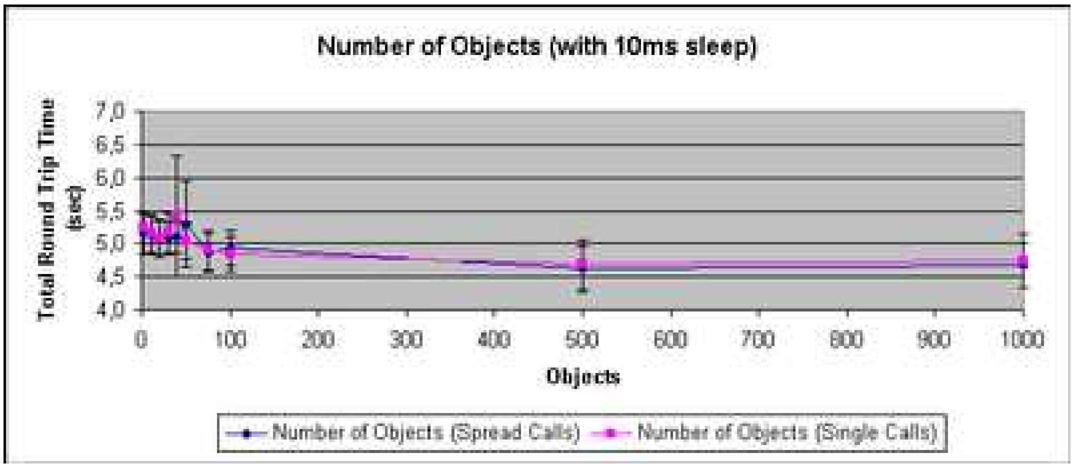


Figure 4-11 shows some interesting instability between 1 and 100 objects. I initially thought that this may be due to some race condition, but Figure 4-12 seems to proof that wrong.

Figure 4-12. Number of Objects RTT (1-1000, 10ms sleep)



Even with a sleep of 10ms between each call (which is not included in the data) we see the anomaly. This effect is subject to further explorations.

Figure 4-13. Number of Objects RTT (1000-44000)[1]

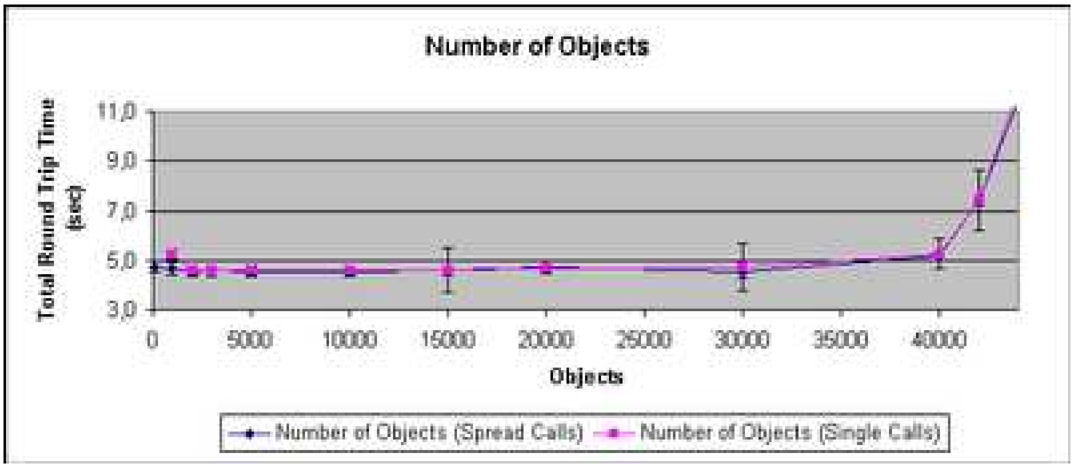


Figure 4-13 shows the stress test. For more than 40000 objects the round trip time increases substantially. Between 45500 and 46000 objects the server ran out of memory.

In summary, the number of objects in the server had no obvious influence to the round trip time.

> **Note:** The tests only include the round trip time. However, the server needs to be contacted for the creation of the objects too. For example, it took approximately 2 server minutes and 1 client minute to create 25000 objects (which is consistent with calling the server that many times). This is, as mentioned, not measured in our tests.

Notes

1. Usually, the server is called 1000 times for each test. In order to call each object at least once, I increased the number for this test to 50000 and normalized the results back to 1000 calls.

Chapter 5. Complex Interaction

In this chapter I will deal with multiple servers and/or multiple clients. Understanding the findings of Chapter 4 should lead to little surprises here.

In order to be somewhat comparable to Glade (Ada95/AnnexE implementation) I decided for a thread pool with 512 threads, just as Glade does.

Star Architecture

In a star architecture we have a single server and multiple clients.

Since only 4 Sparc 20's are available in the test environment I decided for the following configuration for this section.

Server (center of the star)

Sparc 20, 128 Megabytes TMS390Z55

Clients

Ultra 1, 256 Megabytes UltraSPARC

Method

Two situations are evaluated: A single object for all clients and one object per client. In the first case an object is created at server startup and all clients request the reference to this object and call it. In the latter case a new object is requested by each client and then called. Note that we synchronize right before entering the call loop and allow concurrent access to the objects. As usual, the time needed to create the objects is not included.

Figure 5-1. StarClient.java

```
...
long t1 = 0;
long t2 = 0;
if (method.equals("sync_VV")) {
    RTT rtt = factory.createRTT();
```

```
        b.barrier(clients);  // wait for others
        t1 = System.currentTimeMillis();
        for (int i=0; i<loops; i++) {
            rtt.sync_VV();
        }
        t2 = System.currentTimeMillis();
    }
    ...
    System.out.println(t2-t1);
```

Results

Two different time values can be identified for this: The completion time and the (accumulated) round trip time. The former is the wall clock time needed to process all calls, the latter is the sum of every client's round trip time.

As can be seen in Figure 5-2, the completion round trip time raises. We would have expected a steady or slightly decreasing curve because of interleaving effects. It is likely that there is always one client waiting to be served. But on the other hand, they are fighting for the servers attention, and absolute homogeneity of the clients is not guaranteed (e.g. different system load). In short, some unknown random effects.

Note that we are keeping the number of total calls constant (1000). Thus, for four clients we have 250 calls for each. Since our server is threaded we might get the idea that the calls take only a quarter of the time. However, the dispatcher only accepts one connection at a time and then executes the request in a thread. For these tests the objects contain no code, which nullifies the advantage of the threads.[1]

Figure 5-3 shows the (accumulated) round trip time, that is the sum of the round trip times of all clients. As can be seen it is quite linear. For 2 clients each one has its own processing time plus the wait time for the other client, thus doubling the total time, which indicates that the calls are served in a fair manner.

For four clients the completion round trip time is about 5.5ms (Figure 5-2) which, if all 4 are served the same way, multiplies to a total of 22ms. The measured cumulative round trip time is about 20ms. All in all, the calls seem evenly spread.

Using the same object for all clients, or a different one for each client makes no significant difference.

Figure 5-2. Star Architecture (Completion RTT)

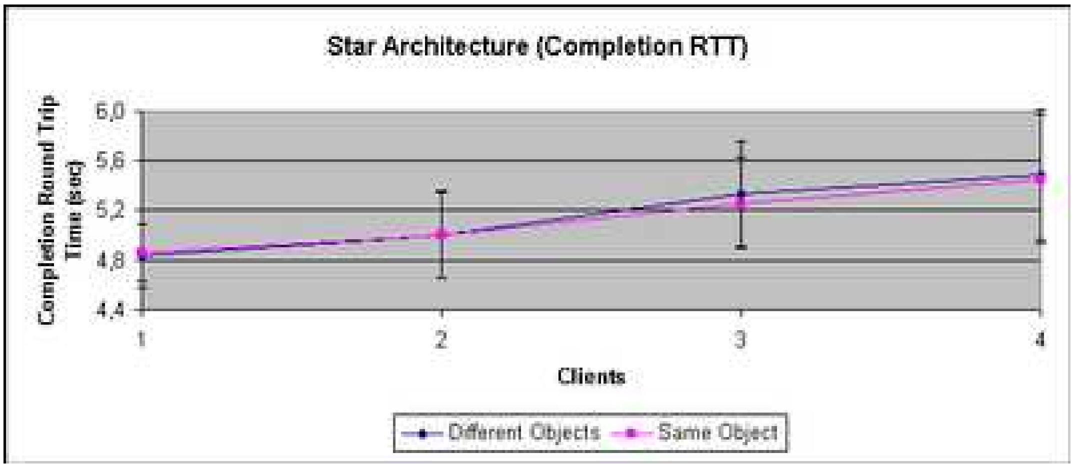
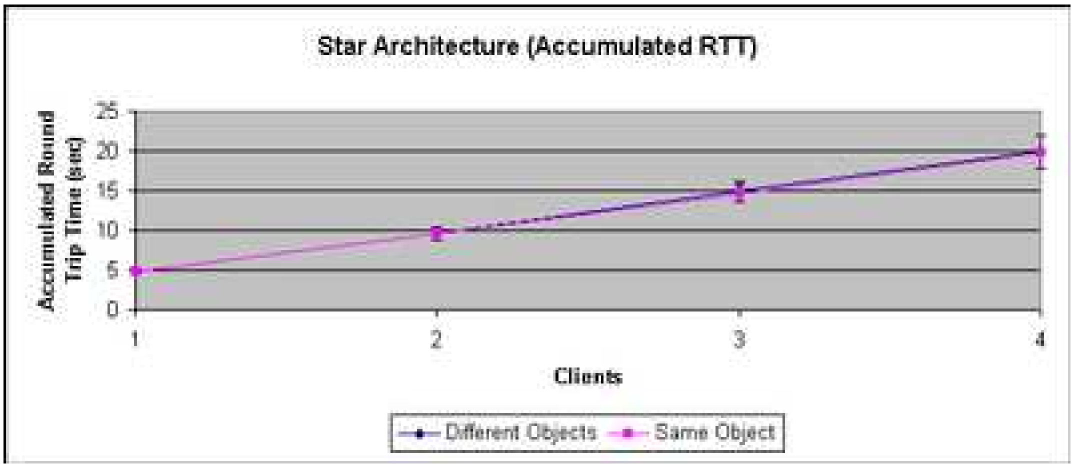


Figure 5-3. Star Architecture (Accumulated RTT)



Mesh

For this test of a mesh configuration we have a complete graph: Each node is both server and client and calls all others. A server is used for synchronization and to distribute the object references.

Again, since the test environment only has 4 Sparc 20's I decided for the following configuration.

Server (synchronization, object references)

Sparc 20, 128 Megabytes TMS390Z55

Nodes

Ultra 1, 256 Megabytes UltraSPARC

Method

First of all, we have to run the server in a separate thread since we use the main thread for calling the other nodes.

Figure 5-4. MeshThread.java

```
class MeshThread implements Runnable {
    org.omg.CORBA.BOA boa;

    public MeshThread(org.omg.CORBA.BOA boa) {
        this.boa = boa;
    }

    public void run() {
        boa.impl_is_ready(null);
    }
}
```

In each node we incarnate a new object, activate it and announce the object reference to the server. Then we wait for the other nodes to complete the same. After that, each node retrieves the references to all other nodes from the server. The clients synchronize again and then start calling each others objects in turn.

Figure 5-5. MeshClient.java

```
        ...
        // register an RTT
        RTT_impl rtt = new RTT_impl();
        factory.registerRTT(id, rtt);
```

```
Thread srv = new Thread(new MeshThread(boa));
srv.start();

Barrier b = factory.getBarrier();
b.barrier(clients);  // wait for others

// get references to the other RTTs
RTT[] otherRTTs = new RTT[clients-1];
{
    int j = 0;
    for(int i = 0; i<clients; i++) {
        if (i != id) {
            otherRTTs[j] = factory.lookupRTT(i);
            j++;
        }
    }
}

long t1 = 0;
long t2 = 0;
if (method.equals("sync_VV")) {
    b.barrier(clients);  // wait for others

    t1 = System.currentTimeMillis();
    for (int i=0; i<loops; i++) {
        for (int j=0; j<otherRTTs.length; j++) {
            otherRTTs[j].sync_VV();
        }
    }
    t2 = System.currentTimeMillis();
}
...
System.out.println(t2-t1);
```

Results

As for the star architecture, we distinguish between completion time and (accumulated) round trip time. The former is the wall clock time needed to process all calls, the latter is the sum of every client's round trip time.

There are a total of 1000 calls in the mesh, no matter how many clients (actually, we may have more but scale back to 1000 calls, as the number of calls is not always evenly divisable).

In Figure 5-6, we see the completion round trip time falling. Ideally it would fall with a rate of 1/n, with n nodes in the mesh. With busy nodes and some requests colliding, i.e. two nodes call the same node, this is not achievable.

Figure 5-7 shows an increase in the (accumulated) round trip time. Ideally, it should stay the same, as each node takes its share of the total calls and processes it. I.e., the total processing cost should be the same. But again, busy nodes and colliding requests prevent this.

Figure 5-6. Mesh Architecture (Completion RTT)

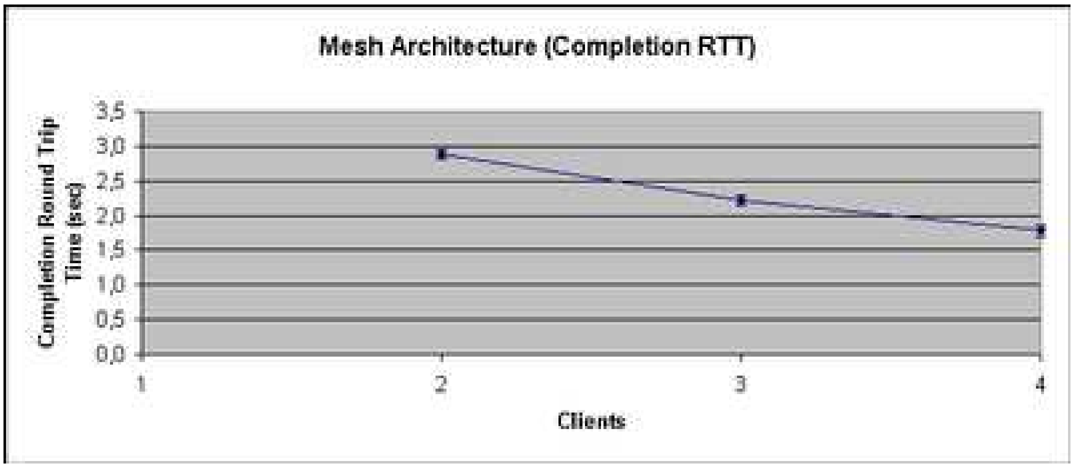
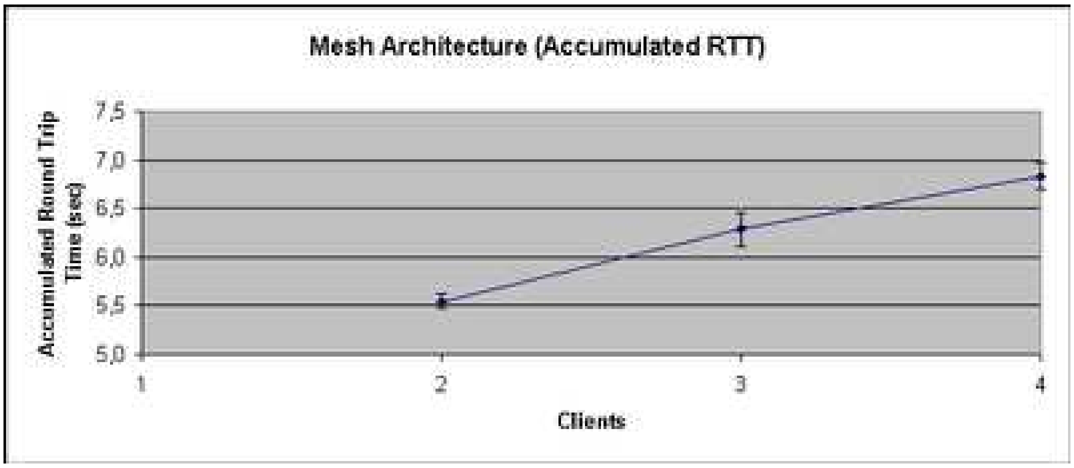


Figure 5-7. Mesh Architecture (Accumulated RTT)



Notes

1. Actually, we add some thread management overhead.

Chapter 6. Conclusions

The used implementation of ORBacus from Object Oriented Concepts ([11]) proofs to be very stable. When this project was close to its finish the new version ORBacus 4.0 became available, which is fully CORBA 2.3 compliant (ORBacus 3.2 is only 2.0 compliant). A major change was the introduction of the POA (Portable Object Adapter). There are several differences to the BOA, but has presumable about the same performance for remote calls.

CORBA References

[1] Michi Henning and Steve Vinoski, *Advanced CORBA programming with C++*, 1999, 1083 pages, 0-201-37927-9, Addison-Wesley Publishing Company.

[2] Alan Pope, *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*, 1998, 407 pages, 0-201-63386-8, Addison-Wesley Publishing Company.

[3] Object Oriented Concepts, Inc., *ORBacus For C++ and Java*, 2000, 310 pages, Object Oriented Concepts, Inc., ftp://ftp.ooc.com/pub/OB/3.3/OB-3.3.pdf .

[4] *The Object Management Group (OMG)*, http://www.omg.org/ .

[5] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 1999, 712 pages, Object Management Group, ftp://ftp.omg.org/pub/docs/formal/99-10-07.pdf .

[6] Object Management Group, *Ada Language Mapping Specification*, 1999, 79 pages, Object Management Group, ftp://ftp.omg.org/pub/docs/formal/99-07-53.pdf .

[7] Object Management Group, *IDL to Java Language Mapping Specification*, 1999, 144 pages, Object Management Group, ftp://ftp.omg.org/pub/docs/formal/99-07-53.pdf .

[8] *Communications of the ACM*, Association for Computing Machinery, October 1999, vol. 42, no. 10, “Comparing Java vs. C/C++ Efficiency Differences to Interpersonal Differences”, Lutz Prechelt, 1999, pages 109-112.

[9] *AdaBroker*, http://adabroker.eu.org/ .

[10] *GNACK: The GNU Ada CORBA Kit*, http://www.adapower.com/gnack/ .

[11] *ORBacus*, http://www.orbacus.com/ .

[12] *CORBA Comparison Project*, , http://www.kav.cas.cz/~buble/corba/comp/ .

[13] Robert Hogg and Elliot Tanis, *Probability and Statistical Inference*, 1997, 722 pages, 0-13-254608-6, Prentice-Hall, Inc..

[14] *JacOrb*, http://www.inf.fu-berlin.de/~brose/jacorb/ .

[15] *JavaORB*,
http://www.multimania.com/dogweb/Download/JavaORB_Download/javaorb_download.html
.

[16] *The CORBA FAQ*, http://www.aurora-tech.com/corba-faq/ .

CORBA Glossary

B

Basic Object Adapter

Adapter between CORBA object and its incarnation. Due to the lack of precision of the standard most implementations are incompatible. The POA should resolve this issue and should be used instead.

See Also: Portable Object Adapter.

BOA

See: Basic Object Adapter

C

Common Object Request Broker Architecture

Standard Object Architecture, managed by the OMG.

CORBA

See: Common Object Request Broker Architecture

G

General Inter-ORB Protocol

Protocol for ORB interoperability.

GIOP

See: General Inter-ORB Protocol

I

IDL

See: Interface Definition Language

IIOP

See: Internet Inter-ORB Protocol

Interface Definition Language

Implementation language-independent interface specification language.

Internet Inter-ORB Protocol

Mapping of GIOP which runs directly over TCP/IP.

Interoperable Object Reference

Reference used when crossing object reference domain boundaries, within bridges.

IOR

See: Interoperable Object Reference

O

Object Management Group

Consortium of 800 members for vendor independent specifications for the software industry.

OMG

See: Object Management Group

Object Request Broker

Provides the means by which clients make and receive requests and responses.

ORB

See: Object Request Broker

P

Portable Object Adapter

Basically a tighter specified and extended BOA. The POA describes a full set of models and policies for managing object life-cycles.

See Also: Basic Object Adapter.

POA

See: Portable Object Adapter

R

Round Trip Time

Time for a message to travel to a destination and back.

RTT

See: Round Trip Time