• Orbix 5.1
• VisiBroker 5.1

Version: Final release
Date: January 30, 2003

Distributed Systems Research Group
Department of Software Engineering
Faculty of Mathematics and Physics
Charles University, Prague, Czech Republic
http://nenya.ms.mff.cuni.cz
Disclaimer
This report is provided "as is", without warranty of any kind, either expressed or implied. This includes, but is not limited to, the implied warranties of merchantability and fitness for a particular purpose. In no event shall any author or copyright holder be liable for damages arising out of the use or the inability to use the report.

Both the results of the measurements in this report and their interpretation depend on a number of technical and circumstantial factors. It is the sole responsibility of the user of this report to assess the applicability of the results and their interpretation.

Authors
The Distributed Systems Research Group is a part of the Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic. The activities of the group include performance evaluation of CORBA broker platforms under the heading of the CORBA Comparison Project. The CORBA Comparison Project commenced in 1998 and has delivered output for partners such as MLC Systeme GmbH (now Deutsche Post Com GmbH), Iona Technologies, Borland International. The group also contributed to the OMG White Paper on Benchmarking.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>3</td>
</tr>
<tr>
<td>Introduction</td>
<td>4</td>
</tr>
<tr>
<td>Platform Configuration</td>
<td>4</td>
</tr>
<tr>
<td>Measurement Technology</td>
<td>4</td>
</tr>
<tr>
<td>Measurement Framework</td>
<td>4</td>
</tr>
<tr>
<td>Acquiring Time Information</td>
<td>4</td>
</tr>
<tr>
<td>Acquiring Resource Usage Information</td>
<td>4</td>
</tr>
<tr>
<td>Measurement Results</td>
<td>5</td>
</tr>
<tr>
<td>Time Distribution Graphs</td>
<td>5</td>
</tr>
<tr>
<td>Time Pattern Graphs</td>
<td>6</td>
</tr>
<tr>
<td>Box And Whiskers Graphs</td>
<td>6</td>
</tr>
<tr>
<td>Resource Usage Graphs</td>
<td>7</td>
</tr>
<tr>
<td>Initial</td>
<td>8</td>
</tr>
<tr>
<td>Ping</td>
<td>8</td>
</tr>
<tr>
<td>Orbix 5.1</td>
<td>9</td>
</tr>
<tr>
<td>VisiBroker 5.1</td>
<td>11</td>
</tr>
<tr>
<td>Summary</td>
<td>13</td>
</tr>
<tr>
<td>Invocation</td>
<td>14</td>
</tr>
<tr>
<td>Static Invocation</td>
<td>14</td>
</tr>
<tr>
<td>Orbix 5.1</td>
<td>15</td>
</tr>
<tr>
<td>VisiBroker 5.1</td>
<td>21</td>
</tr>
<tr>
<td>Summary</td>
<td>27</td>
</tr>
<tr>
<td>Dynamic Invocation</td>
<td>32</td>
</tr>
<tr>
<td>Orbix 5.1</td>
<td>34</td>
</tr>
<tr>
<td>VisiBroker 5.1</td>
<td>40</td>
</tr>
<tr>
<td>Summary</td>
<td>46</td>
</tr>
<tr>
<td>Marshaling</td>
<td>51</td>
</tr>
<tr>
<td>Marshaling Basic Types</td>
<td>51</td>
</tr>
<tr>
<td>Orbix 5.1</td>
<td>53</td>
</tr>
<tr>
<td>VisiBroker 5.1</td>
<td>58</td>
</tr>
<tr>
<td>Summary</td>
<td>63</td>
</tr>
<tr>
<td>Marshaling Arrays Of Basic Types</td>
<td>71</td>
</tr>
<tr>
<td>Orbix 5.1</td>
<td>73</td>
</tr>
<tr>
<td>VisiBroker 5.1</td>
<td>78</td>
</tr>
<tr>
<td>Summary</td>
<td>83</td>
</tr>
<tr>
<td>Marshaling Sequences Of Octets</td>
<td>91</td>
</tr>
<tr>
<td>Orbix 5.1</td>
<td>93</td>
</tr>
<tr>
<td>VisiBroker 5.1</td>
<td>99</td>
</tr>
<tr>
<td>Summary</td>
<td>105</td>
</tr>
<tr>
<td>Dispatching</td>
<td>113</td>
</tr>
<tr>
<td>Dispatching To Servant Instances</td>
<td>113</td>
</tr>
<tr>
<td>Orbix 5.1</td>
<td>114</td>
</tr>
<tr>
<td>VisiBroker 5.1</td>
<td>117</td>
</tr>
<tr>
<td>Summary</td>
<td>120</td>
</tr>
</tbody>
</table>
Introduction
The purpose of this report is to present results of measurements that evaluate CORBA broker platforms along the three axes of response time, throughput and resource requirements.

Platform Configuration
The Solaris Interoperability platform was configured as follows:

- Sun Ultra 2
- Dual Sun UltraSparc I 200MHz
- 384MB RAM
- No Network
- Sun Solaris 8.0
- Sun WorkShop 6 C++ 5.3
- Sun JDK 1.3.1

The brokers tested on the client side were:
- Orbix 5.1 Java
- VisiBroker 5.1 Java

The brokers tested on the server side were:
- Orbix 5.1 C++
- VisiBroker 5.1 C++

Measurement Technology
Dedicated suites of tests measure individual aspects of broker performance. All suites employ the same measurement framework. To form a specific test, the framework is augmented by an action to be measured on the client side and servants to be instantiated on the server side.

The concept of measurement framework is employed in both the C++ and Java environments. The implementations in both languages are functionally equivalent, except for short explicit garbage collection loop in Java client, which is aimed at ensuring similar starting conditions for each measurement.

Measurement Framework
The measurement framework carries out a measurement in five stages.

Warm-up
In the warm-up stage, the framework instantiates threads that carry out the action to be measured. When the threads start, the framework waits for a random time from 5 to 20 seconds to make sure the system is under a stable load before the measurement progresses.

Client Resource Usage
In the client resource usage stage, the framework collects information about client resource usage in 60 samples taken over the period of 60 seconds. For each resource, the minimum, maximum and average values are kept.

Server Resource Usage
In the server resource usage stage, the framework collects information about server resource usage, same as in the client resource usage stage.

Solo Measurement
In the solo measurement stage, the framework records 5000 individual measurements observed by a randomly selected thread. Information such as statistical distribution and quartiles is calculated from the individual measurements.

Shared Measurement
In the shared measurement stage, the framework records minimum, maximum and average values of measurements observed by all threads over the period of 60 seconds. Information such as overall minima, maxima and averages is calculated from the shared measurements.

Each measurement is repeated 5 times.

Acquiring Time Information
Solaris
On Solaris, the `gethrtime()` system call is used to obtain the time information. The resolution of thus obtained time is 1ns.

Windows
On Windows, the `QueryPerformanceCounter()` system call is used to obtain the time information. The resolution of thus obtained time is the same as the frequency of the processor clock.

Acquiring Resource Usage Information
Solaris
The memory usage information on Solaris is collected from the `/proc/self/psinfo` file, where the resident and swapped application memory sizes are stored, from the `kstat` structures, where the kernel memory size is stored, from the `sysconf()` system call, which reports the total physical memory size, and from the `swapctl()` system call, which reports the total swapped memory size.

The network usage information on Solaris is collected from the `kstat` structures, where the network traffic information is stored. The network usage information does not include the traffic over the loopback interface in bytes, which is not reported by the operating system.
The processor usage information on Solaris is collected by hooking the `thr_create()` and `pthread_create()` system calls, where the application thread count is kept, and from the `kstat` structures, where the total processor usage is stored.

**Windows**

On Windows, most information is collected from performance data accessible under `HKEY_PERFORMANCE_DATA` key in the registry. The data is not actually stored in the registry; the registry functions cause the system to collect the data from its source. The data come in form of performance objects, which represent various resources found in a system. Each object has a name and a set of named counters associated with it. The counters associated with an object can come in several instances for entities where it makes sense, e.g. processes, network interfaces, etc.

The memory usage information is collected from `Working Set` and `Page File Bytes` counters of the corresponding instance of `Process` object, where the resident and swapped application memory sizes are stored, from `Pool Paged Bytes` and `Pool Nonpaged Bytes` counters of the `Memory` object, where the kernel memory sizes are stored and from the `GlobalMemoryStatus()` system call, which reports the total physical and total swapped memory sizes.

The network usage information is collected from `Bytes Sent/sec`, `Bytes Received/sec`, `Packets Sent/sec` and `Packets Received/sec` counters of all instances of `Network Interface` object.

The processor usage information is collected from `Thread Count` counter of the corresponding instance of `Process` object, where the application thread count is kept, and from `% User Time`, `% Processor Time` and `% Privileged Time` counters of all instances of `Processor` object, where the total processor usage is stored.

**Java**

In Java, the information is collected using native code methods contained in platform dependent dynamic shared libraries, which utilize the above-mentioned methods. The only exception is the application thread count, which is determined using pure Java methods and represents the number of active threads in the root thread group.

**Measurement Results**

Individual suites of tests are described in dedicated chapters. All chapters begin with a description of the suite accompanied by code fragments that characterize the suite. Except where not applicable, the interface definitions in IDL are present together with the code of the action to be measured on the client side and the code of the servants to be instantiated on the server side.

The results of the tests follow the description in chapters dedicated to individual brokers and platforms. When more than one broker has been tested on a platform, a platform summary is also provided. The results are plotted in graphs accompanied by a description of what the graph shows.

**Time Distribution Graphs**

![Time Distribution Graph](image)

The time distribution graphs are calculated from data collected in the solo measurement stage. On the X axis are intervals of the time being measured, on the Y axis is the percentage of the measured times that fell within the given interval. The bars marked as other denote measured times that were either smaller or bigger than any of the intervals considered.
**Time Pattern Graphs**

The time pattern graphs are calculated from data collected in the solo measurement stage. On the X axis is the sequential number of the measurement, on the Y axis is the measured time.

**Box And Whiskers Graphs**

The box and whiskers graphs are calculated from data collected in the solo measurement stage. The box spans from low to high quartile of the collected data and thus indicates the range within which half of the measured values fall. The whiskers span from minimum to maximum of the collected data and thus indicate the range within which all of the measured values fall. The median of the collected data is marked within the box.
Resource Usage Graphs

The resource usage graphs are calculated from data collected in the client and server resource usage measurement stages. The solid line marks the average of the collected data. When present, a pair of dotted lines marks the minimum and maximum of the collected data.
Initial Ping

Description
The suite measures the time it takes to complete a trivial method invocation. The purpose of the suite is to assess the behavior of the broker on as simple an example as possible to minimize the chance of a systematic error.

IDL Code

```
interface Ping { void Pong (); }
```

C++ Code

```
ulonglong ActionPing (void *pArgs)
{
    pPing->Pong ();
    return (0);
}
void PingServant::Pong () {}  
```

Java Code

```
public final void execute (long [] times) {
    pingServer.Pong ();
}
public final void Pong () {}  
```

Results
The time distribution graphs should display a cluster with small variance. The observed times should correspond with the observed times of other suites. The time pattern graphs should display a sequence with small variance.
**Orbix 5.1**

**Time Distribution (1 Thread)**
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.

![Time Distribution (1 Thread)](image)

**Time Pattern (1 Thread)**
The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.

![Time Pattern (1 Thread)](image)

**Time Pattern (1 Thread, Zoom)**
The graph is a zoom in of the previous graph.
VisiBroker 5.1

**Time Distribution (1 Thread)**
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.

![](image1.png)

**Time Pattern (1 Thread)**
The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.

![](image2.png)

**Time Pattern (1 Thread, Zoom)**
The graph is a zoom in of the previous graph.
Summary

Time Distribution (1 Thread)
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.

Time Pattern (1 Thread)
The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.
## Invocation

### Static Invocation

#### Description
The suite measures the time it takes to complete a static method invocation. The purpose of the suite is to assess the behavior of the static invocation mechanism.

#### IDL Code
```
interface Ping { ulonglong Pong (); }
```

#### C++ Code
```
ulonglong ActionPing (void *pArgs)
{
    return (pPing->Pong ());
}

CORBA::ULongLong PingServant::Pong ()
{
    return (TIMGetTimestamp ());
}
```

#### Java Code
```
public final void execute (long [] times) {
    times [0] = pingServer.Pong ();
}

public final long Pong () {
    return SysInformer.timGetTimestamp ();
}
```

#### Results
The time distribution graphs should display clusters with small variance for both the halfway and the roundtrip times. The clusters should be gradually wider when multiple threads are used.

The time pattern graphs should display a sequence with small variance for both the halfway and the roundtrip times. The sequence should be gradually less uniform when multiple threads are used.

The dependency of both the halfway and the roundtrip times on the number of threads should display constant minimum values and at least linearly increasing median, average and maximum values. The increase over the linear dependency represents the overhead introduced by the concurrent execution.

The client throughput should be constant or decreasing, except on multiprocessor systems, where the throughput should increase initially until the number of threads reaches the number of processors. The decrease under the constant level represents the overhead introduced by the concurrent execution.

The memory usage should be constant or very slightly increasing. The network usage should be constant or very slightly decreasing. The thread usage should grow linearly on the client and reflect the threading model on the server. The processor usage should follow the throughput.
**Orbix 5.1**

**Time Distribution (1 Thread)**
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.

![Time Distribution (1 Thread)](image)

**Time Distribution (10 Threads)**
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.

![Time Distribution (10 Threads)](image)

**Time Distribution (100 Threads)**
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.

![Time Distribution (100 Threads)](image)
**Time Pattern (1 Thread)**

The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.

**Time Pattern (1 Thread, Zoom)**

The graph is a zoom in of the previous graph.
**Time Pattern (10 Threads)**
The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.

**Time Pattern (10 Threads, Zoom)**
The graph is a zoom in of the previous graph.
Time Pattern (100 Threads)
The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.

Time Pattern (100 Threads, Zoom)
The graph is a zoom in of the previous graph.
Dependency On Number Of Threads

The graph depicts the dependency of the time it takes to complete the invocation on the number of threads issuing the invocations concurrently. On the X axis is the number of threads, on the Y axis is the time it took to complete the invocation.
CORBA Comparison Project, Solaris Interoperability Platform

Static Invocation/Orbix 5.1

Confidential Page 20 Copyright © 2003 Distributed Systems Research Group, Charles University
**VisiBroker 5.1**

**Time Distribution (1 Thread)**
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.

![Time Distribution (1 Thread)](image1)

**Time Distribution (10 Threads)**
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.

![Time Distribution (10 Threads)](image2)

**Time Distribution (100 Threads)**
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.

![Time Distribution (100 Threads)](image3)
**Time Pattern (1 Thread)**

The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.

**Time Pattern (1 Thread, Zoom)**

The graph is a zoom in of the previous graph.
The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.

The graph is a zoom in of the previous graph.
**Time Pattern (100 Threads)**

The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.

**Time Pattern (100 Threads, Zoom)**

The graph is a zoom in of the previous graph.
Dependency On Number Of Threads
The graph depicts the dependency of the time it takes to complete the invocation on the number of threads issuing the invocations concurrently. On the X axis is the number of threads, on the Y axis is the time it took to complete the invocation.
Summary

Time Distribution (1 Thread)
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.

Time Distribution (10 Threads)
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.

Time Distribution (100 Threads)
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.
Time Pattern (1 Thread)
The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.

Time Pattern (10 Threads)
The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.
**Time Pattern (100 Threads)**

The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.

**Dependency On Number Of Threads**

The graph depicts the dependency of the time it takes to complete the invocation on the number of threads issuing the invocations concurrently. On the X axis is the number of threads, on the Y axis is the time it took to complete the invocation.
Dynamic Invocation

Description

The suite measures the time it takes to complete a dynamic method invocation. The purpose of the suite is to assess the behavior of the dynamic invocation mechanism.

C++ Code

```cpp
ulonglong ActionPing (void *pArgs)
{
    CORBA::Request_var vRequest;
    CORBA::ULongLong iResult;

    vRequest = pPing->_request ("Pong");
    vRequest->set_return_type (CORBA::_tc_ulonglong);
    vRequest->invoke (ORBArgOnly);
    *(vRequest->result ()->value ()) >>= iResult;

    return (iResult);
}

void PingServant::invoke (CORBA::ServerRequest_ptr pRequest)
{
    // Note that calling arguments is mandated by the
    // standard even when the function has none.
    CORBA::NVList_ptr pArguments;
    CORBA::Any oResult;

    pORB->create_list (0, pArguments);
    pRequest->arguments (pArguments);
    oResult <<= (CORBA::ULongLong) TIMGetTimestamp ();
    pRequest->set_result (oResult);
}
```

Java Code

```java
public final void execute (long [] times)
{
    Request request = pingServer._request ("Pong");

    request.set_return_type (
        BenchBroker.theORB.get_primitive_tc (TCKind.tk_ulonglong));
    request.invoke ();

    times [0] = request.result ().value ().extract_ulonglong ();
}

public final void invoke (ServerRequest request)
{
    if (! request.operation ().equals ("Pong"))
    {
        throw new org.omg.CORBA.BAD_OPERATION ();
    }

    /*
     * Note that calling arguments is mandated by the
     * standard even when the function has none.
     */
    org.omg.CORBA.NVList args = BenchBroker.theORB.create_list (0);
    request.arguments (args);

    org.omg.CORBA.Any result = BenchBroker.theORB.create_any ();
    request.type (BenchBroker.theORB.get_primitive_tc (TCKind.tk_ulonglong));
    result.insert_ulonglong (SysInformer.timGetTimestamp ());
    request.set_result (result);
}
Results

The time distribution graphs should display clusters with small variance for both the halfway and the roundtrip times. The clusters should be gradually wider when multiple threads are used.

The time pattern graphs should display a sequence with small variance for both the halfway and the roundtrip times. The sequence should be gradually less uniform when multiple threads are used.

The dependency of both the halfway and the roundtrip times on the number of threads should display constant minimum values and at least linearly increasing median, average and maximum values. The increase over the linear dependency represents the overhead introduced by the concurrent execution.

The client throughput should be constant or decreasing, except on multiprocessor systems, where the throughput should increase initially until the number of threads reaches the number of processors. The decrease under the constant level represents the overhead introduced by the concurrent execution.

The memory usage should be constant or very slightly increasing. The network usage should be constant or very slightly decreasing. The thread usage should grow linearly on the client and reflect the threading model on the server. The processor usage should follow the throughput.
**Orbix 5.1**

**Time Distribution (1 Thread)**
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.

![Time Distribution (1 Thread)](image)

**Time Distribution (10 Threads)**
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.

![Time Distribution (10 Threads)](image)

**Time Distribution (100 Threads)**
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.

![Time Distribution (100 Threads)](image)
Time Pattern (1 Thread)
The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.

Time Pattern (1 Thread, Zoom)
The graph is a zoom in of the previous graph.
Time Pattern (10 Threads)
The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.

Time Pattern (10 Threads, Zoom)
The graph is a zoom in of the previous graph.
**Time Pattern (100 Threads)**

The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.

**Time Pattern (100 Threads, Zoom)**

The graph is a zoom in of the previous graph.
Dependency On Number Of Threads
The graph depicts the dependency of the time it takes to complete the invocation on the number of threads issuing the invocations concurrently. On the X axis is the number of threads, on the Y axis is the time it took to complete the invocation.
**VisiBroker 5.1**

**Time Distribution (1 Thread)**
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.

![Time Distribution (1 Thread)](image)

**Time Distribution (10 Threads)**
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.

![Time Distribution (10 Threads)](image)

**Time Distribution (100 Threads)**
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.

![Time Distribution (100 Threads)](image)
**Time Pattern (1 Thread)**

The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.

**Time Pattern (1 Thread, Zoom)**

The graph is a zoom in of the previous graph.
**Time Pattern (10 Threads)**

The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.

**Time Pattern (10 Threads, Zoom)**

The graph is a zoom in of the previous graph.
The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.

The graph is a zoom in of the previous graph.
Dependency On Number Of Threads

The graph depicts the dependency of the time it takes to complete the invocation on the number of threads issuing the invocations concurrently. On the X axis is the number of threads, on the Y axis is the time it took to complete the invocation.
Summary

Time Distribution (1 Thread)
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.

Time Distribution (10 Threads)
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.

Time Distribution (100 Threads)
The graph depicts the statistical distribution of the time it takes to complete the method invocation. On the X axis are the time intervals, on the Y axis is the percentage of invocations completed within the given time interval.
Time Pattern (1 Thread)

The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.

Time Pattern (10 Threads)

The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.
Time Pattern (100 Threads)
The graph depicts the pattern of changes of the time it takes to complete the method invocation. On the X axis is the sequential number of the measurement, on the Y axis is the time it took to complete the invocation.

Dependency On Number Of Threads
The graph depicts the dependency of the time it takes to complete the invocation on the number of threads issuing the invocations concurrently. On the X axis is the number of threads, on the Y axis is the time it took to complete the invocation.
Marshaling Basic Types

Description
The suite measures the time it takes to complete a method invocation depending on the type and direction of the argument. The purpose of the suite is to assess the behavior of the marshaling mechanism.

Direction IN

IDL Code

typedef <type> tPingArg;
interface Ping { void Pong (in tPingArg xArg); }  

C++ Code

tPingArg xPingArg;
ulonglong ActionPing (void *pArgs) {
    pPing->Pong (xPingArg);
    return (0);
}
void PingServant::Pong (tPingArg xArg) { }

Java Code

private <type> pingArg;
public final void execute (long [] times) {
    pingServer.Pong (pingArg);
}
public final void Pong (<type> xArg) { }

Direction OUT

IDL Code

typedef <type> tPingArg;
interface Ping { void Pong (out tPingArg xArg); }  

C++ Code

tPingArg xPingArg;
ulonglong ActionPing (void *pArgs) {
    pPing->Pong (xPingArg);
    return (0);
}
void PingServant::Pong (tPingArg_out xArg) { }

Java Code

private <type> pingArg;
public final void execute (long [] times) {
    pingServer.Pong (pingArg);
}
public final void Pong (<type> xArg) { }

Direction INOUT

IDL Code

typedef <type> tPingArg;
interface Ping { void Pong (inout tPingArg xArg); }  

C++ Code
tPingArg xPingArg;
ulonglong ActionPing (void *pArgs)
{
    pPing->Pong (xPingArg);
    return (0);
}

void PingServant::Pong (tPingArg &xArg) { }

Java Code

private <type> pingArg;
public final void execute (long [] times) {
    pingServer.Pong (pingArg);
}

public final void Pong (<type> xArg) { }

Direction RETURN

IDL Code

typedef <type> tPingArg;
interface Ping { tPingArg Pong (); }

C++ Code

ulonglong ActionPing (void *pArgs)
{
    pPing->Pong ();
    return (0);
}

tPingArg xPingArg;
tPingArg PingServant::Pong ()
{
    return (xPingArg);
}

Java Code

public final void execute (long [] times) {
    pingServer.Pong ();
}

private <type> pingArg;
public final <type> Pong () {
    return (pingArg);
}

Results

The roundtrip time should be constant for all the types and directions of the argument, except for the char and wchar types, where the roundtrip time may be affected by the character set conversion, and the long long, unsigned long long and long double types, where the roundtrip time may be affected by the handling of non native types. The quartile range of the values should be small.

The memory usage, network usage and processor usage should be constant, except for the char and wchar types, where the usage may be affected by the character set conversion, and the long long, unsigned long long and long double types, where the usage may be affected by the handling of non native types.
Orbix 5.1

Dependency On Type (Direction IN, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
Dependency On Type (Direction OUT, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
Dependency On Type (Direction INOUT, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
Dependency On Type (Direction RETURN, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
VisiBroker 5.1

Dependency On Type (Direction IN, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
**Dependency On Type (Direction OUT, 1 Thread)**

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
Dependency On Type (Direction INOUT, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
Dependency On Type (Direction RETURN, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
**Summary**

**Dependency On Type (Direction IN, 1 Thread)**

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.

![Graph showing dependency on type (Direction IN, 1 Thread)](image)
Dependency On Type (Direction OUT, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
Dependency On Type (Direction INOUT, 1 Thread)
The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
Dependency On Type (Direction RETURN, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
Marshaling Arrays Of Basic Types

Description
The suite measures the time it takes to complete a method invocation depending on the type and direction of the argument. The purpose of the suite is to assess the behavior of the marshaling mechanism.

Direction IN

IDL Code

typedef <type> tPingArg [size];
interface Ping { void Pong (in tPingArg xArg); }

C++ Code

```cpp
tPingArg xPingArg;
ulonglong ActionPing (void *pArgs)
{
    pPing->Pong (xPingArg);
    return (0);
}
void PingServant::Pong (const tPingArg xArg) { }
```

Java Code

```java
private tPingArgHolder pingArg;
public final void execute (long [] times)
    { pingServer.Pong (pingArg.value); }
public final void Pong (<type> [] xArg) { }
```

Direction OUT

IDL Code

typedef <type> tPingArg [size];
interface Ping { void Pong (out tPingArg xArg); }

C++ Code

```cpp
tPingArg xPingArg;
ulonglong ActionPing (void *pArgs)
{
    pPing->Pong (xPingArg);
    return (0);
}
void PingServant::Pong (tPingArg_out xArg) { }
```

Java Code

```java
private tPingArgHolder pingArg;
public final void execute (long [] times)
    { pingServer.Pong (pingArg.value); }
private <type> [] pingArg = new <type> [size];
public final void Pong (tPingArgHolder xArg)
    { xArg.value = pingArg; }
```

Direction INOUT

IDL Code

typedef <type> tPingArg [size];
interface Ping { void Pong (inout tPingArg xArg); }

C++ Code

```cpp
tPingArg xPingArg;
ulonglong ActionPing (void *pArgs)
{
    pPing->Pong (xPingArg);
    return (0);
}
void PingServant::Pong (tPingArg_out xArg) { }
```
tPingArg xPingArg;
ulonglong ActionPing (void *pArgs)
{
    pPing->Pong (xPingArg);
    return (0);
}

void PingServant::Pong (tPingArg &xArg) { }

Java Code

private tPingArgHolder pingArg;
public final void execute (long [] times) {
    pingServer.Pong (pingArg);
}

public final void Pong (tPingArgHolder xArg) { }

Results

The roundtrip time should depend predominantly on the total size of the argument for all the types and directions of the argument, except for the char and wchar types, where the roundtrip time may be affected by the character set conversion, and the long long, unsigned long long and long double types, where the roundtrip time may be affected by the handling of non native types. The quartile range of the values should be small.

The memory usage, network usage and processor usage should depend predominantly on the total size of the argument, except for the char and wchar types, where the usage may be affected by the character set conversion, and the long long, unsigned long long and long double types, where the usage may be affected by the handling of non native types.
**Orbix 5.1**

**Dependency On Type (Direction IN, 1024 Octets, 1 Thread)**

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
Dependency On Type (Direction OUT, 1024 Octets, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
Dependency On Type (Direction INOUT, 1024 Octets, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.

**Dependency On Type (Direction RETURN, 1024 Octets, 1 Thread)**

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
**VisiBroker 5.1**

**Dependency On Type (Direction IN, 1024 Octets, 1 Thread)**

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.

![Graph showing dependency on type](image)
Dependency On Type (Direction OUT, 1024 Octets, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
Dependency On Type (Direction INOUT, 1024 Octets, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
Dependency On Type (Direction RETURN, 1024 Octets, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
### Summary

**Dependency On Type (Direction IN, 1024 Octets, 1 Thread)**

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
Dependency On Type (Direction OUT, 1024 Octets, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
Dependency On Type (Direction INOUT, 1024 Octets, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
Dependency On Type (Direction RETURN, 1024 Octets, 1 Thread)
The graph depicts the dependency of the time it takes to complete the invocation on the type and direction of the argument. On the X axis is the type of the argument, on the Y axis is the time it took to complete the invocation.
Marshaling Sequences Of Octets

Description
The suite measures the time it takes to complete a method invocation depending on the size of the octet sequence passed as argument. The purpose of the suite is to assess the behavior of the marshaling mechanism.

Direction IN

IDL Code

typedef sequence<octet> tOctetSequence;
interface Ping { void Pong (in tOctetSequence sSequence); } 

C++ Code

tOctetSequence *pPingOctetSequence;
ulonglong ActionPing (void *pArgs)
{
    pPing->Pong (*pPingOctetSequence);
}
void PingServant::Pong (const tOctetSequence &sSequence) { }

Java Code

private tOctetSequenceHolder sequenceHolder;

public final void execute (long [] times) {
    pingServer.Pong (sequenceHolder.value); 
}
public final void Pong (byte [] sSequence) { }

Direction OUT

IDL Code

typedef sequence<octet> tOctetSequence;
interface Ping
{
    void Pong (in unsigned long iLength, out tOctetSequence sSequence);
}

C++ Code

tOctetSequence *pPingOctetSequence;
CORBA::ULong iPingOctetSequenceLength;
ulonglong ActionPing (void *pArgs)
{
    pPing->Pong (iPingOctetSequenceLength, pPingOctetSequence);
    delete (pPingOctetSequence);
}
void PingServant::Pong (CORBA::ULong iLength, tOctetSequence_out pSequence)
{
    pSequence = new tOctetSequence (iLength);
    pSequence->length (iLength);
}

Java Code

public static int sequenceLength;
private tOctetSequenceHolder sequenceHolder;
public final void execute (long [] times) {
    pingServer.Pong (sequenceLength, sequenceHolder);
    sequenceHolder.value = null;
}
public final void Pong (int iLength, tOctetSequenceHolder sSequence) {
    sSequence.value = new byte [iLength];
}
**Direction INOUT**

**IDL Code**

typedef sequence<octet> tOctetSequence;
interface Ping { void Pong (inout tOctetSequence sSequence); }  

**C++ Code**

tOctetSequence *pPingOctetSequence;
ulonglong ActionPing (void *pArgs)
{
    pPing->Pong (*pPingOctetSequence);
}

void PingServant::Pong (tOctetSequence &sSequence) { }

**Java Code**

private tOctetSequenceHolder sequenceHolder;
public final void execute (long [] times) {
    pingServer.Pong (sequenceHolder);
}

public final void Pong (tOctetSequenceHolder sSequence) { }

**Direction RETURN**

**IDL Code**

typedef sequence<octet> tOctetSequence;
interface Ping { tOctetSequence Pong (in unsigned long iLength); }  

**C++ Code**

CORBA::ULong iPingOctetSequenceLength;
ulonglong ActionPing (void *pArgs)
{
    delete (pPing->Pong (iPingOctetSequenceLength);
}

tOctetSequence *PingServant::Pong (CORBA::ULong iLength)
{
    tOctetSequence *pSequence;
    pSequence = new tOctetSequence (iLength);
    pSequence->length (iLength);
    return (pSequence);
}

**Java Code**

public static int sequenceLength;
public final void execute (long [] times) {
    pingServer.Pong (sequenceLength);
}

public final byte [] Pong (int iLength) {
    return new byte [iLength];
}

**Results**

The dependency of the roundtrip time on the size of the sequence should display linearly increasing minimum, median and average values and reasonably small maximum values.

The memory usage should be constant or very slightly increasing. The network usage should be asymptotically increasing towards the maximum capacity. The processor usage should follow the network usage.
Orbix 5.1

Dependency On Size Of Sequence (Direction IN, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the size of the octet sequence passed as argument. On the X axis is the size of the sequence, on the Y axis is the time it took to complete the invocation.
Dependency On Size Of Sequence (Direction OUT, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the size of the octet sequence passed as argument. On the X axis is the size of the sequence, on the Y axis is the time it took to complete the invocation.
Dependency On Size Of Sequence (Direction INOUT, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the size of the octet sequence passed as argument. On the X axis is the size of the sequence, on the Y axis is the time it took to complete the invocation.
Dependency On Size Of Sequence (Direction RETURN, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the size of the octet sequence passed as argument. On the X axis is the size of the sequence, on the Y axis is the time it took to complete the invocation.
VisiBroker 5.1

Dependency On Size Of Sequence (Direction IN, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the size of the octet sequence passed as argument. On the X axis is the size of the sequence, on the Y axis is the time it took to complete the invocation.
Dependency On Size Of Sequence (Direction OUT, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the size of the octet sequence passed as argument. On the X axis is the size of the sequence, on the Y axis is the time it took to complete the invocation.
Dependency On Size Of Sequence (Direction INOUT, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the size of the octet sequence passed as argument. On the X axis is the size of the sequence, on the Y axis is the time it took to complete the invocation.
Dependency On Size Of Sequence (Direction RETURN, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the size of the octet sequence passed as argument. On the X axis is the size of the sequence, on the Y axis is the time it took to complete the invocation.
Summary

Dependency On Size Of Sequence (Direction IN, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the size of the octet sequence passed as argument. On the X axis is the size of the sequence, on the Y axis is the time it took to complete the invocation.
Dependency On Size Of Sequence (Direction OUT, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the size of the octet sequence passed as argument. On the X axis is the size of the sequence, on the Y axis is the time it took to complete the invocation.
Dependency On Size Of Sequence (Direction INOUT, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the size of the octet sequence passed as argument. On the X axis is the size of the sequence, on the Y axis is the time it took to complete the invocation.
Dependency On Size Of Sequence (Direction RETURN, 1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the size of the octet sequence passed as argument. On the X axis is the size of the sequence, on the Y axis is the time it took to complete the invocation.
Dispatching

Dispatching To Servant Instances

Description
The suite measures the time it takes to complete a method invocation depending on the number of active servants hosted by the server. The purpose of the suite is to assess the behavior of the dispatching mechanism.

IDL Code

```idl
interface Ping { void Pong (); }
typedef sequence<Ping> tPingSequence;
```

C++ Code

```cpp
tPingSequence *pPingSequence;
CORBA::ULong iPingSequenceLength;
ulonglong ActionPing (void *pArgs)
{
   (*pPingSequence) [RNDGetRandom (iPingSequenceLength)]->Pong ();
   return (0);
}
void PingServant::Pong () { }
```

Java Code

```java
public static int pingSequenceLength;
public static Ping [] pingSequence;
public final void execute (long [] times) {
   pingSequence [(int) BenchGlobal.rand (pingSequenceLength)].Pong ();
}
public final void Pong () { }
```

Results
The dependency of the roundtrip time on the number of servants should display constant or less than linearly increasing minimum, median and average values and reasonably small maximum values.

The memory usage should be linearly increasing and reflect the size of the servants on the server and the proxies on the client. The network usage should be constant or very slightly decreasing. The processor usage should be monotonous.
**Orbix 5.1**

**Dependency On Number Of Servants (1 Thread)**

The graph depicts the dependency of the time it takes to complete the invocation on the number of active servants hosted by the server. On the X axis is the number of servants, on the Y axis is the time it took to complete the invocation.
Dependency On Number Of Servants (1 Thread, Zoom)

The graph is a zoom in of the previous graph.
**VisiBroker 5.1**

**Dependency On Number Of Servants (1 Thread)**

The graph depicts the dependency of the time it takes to complete the invocation on the number of active servants hosted by the server. On the X axis is the number of servants, on the Y axis is the time it took to complete the invocation.

![Graph showing dependency of time on number of servants](image-url)
Dependency On Number Of Servants (1 Thread, Zoom)

The graph is a zoom in of the previous graph.
Summary

Dependency On Number Of Servants (1 Thread)

The graph depicts the dependency of the time it takes to complete the invocation on the number of active servants hosted by the server. On the X axis is the number of servants, on the Y axis is the time it took to complete the invocation.
Dependency On Number Of Servants (1 Thread, Zoom)

The graph is a zoom in of the previous graph.
CORBA Comparison Project, Solaris Interoperability Platform

Marshaling Sequences Of Octets/Summary

% Server Network Packets Received

% Client Processor Kernel

% Client Processor Wait

% Client Processor User

% Client Processor Idle

% Server Processor Kernel

% Server Processor Wait

% Server Processor User

% Server Processor Idle

© 2003 Distributed Systems Research Group, Charles University