IDA

INSTITUTE FOR DEFENSE ANALYSES

Predicting CORBA Performance Through Prototyping

Clyde G. Roby, Task Leader

Edward A. Feustel

DTIC QUALITY INSPECTED 4

February 1997

Approved for public release;
distribution unlimited.

IDA Paper P-3327

Log: H 97-000587

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

19971002 074

This work was conducted under contract DASW01 94 C 0054, Task T-S5-1446, for the Defense Information Systems Agency. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

© 1997 Institute for Defense Analyses, 1801 N. Beauregard Street, Alexandria, Virginia 22311-1772 • (703) 845-2000.

This material may be reproduced by or for the U.S. Government pursuant to the copyright license under the clause at DFARS 252.227-7013 (10/88).

PREFACE

This document was prepared by the Institute for Defense Analyses (IDA) for the Software Data Architecture Engineering Division, Center for Computer Systems Engineering, Defense Information Systems Agency (DISA), under the task entitled "Common Operating Environment Architecture". This document fulfills these task objectives:

- To provide a first-order performance model and associated analysis of distributed computing in a Client/Server application.
- To provide guidance to technical designers and developers about how to analyze the performance of new or legacy applications in order to engineer or re-engineer them for implementation in a distributed environment.

The following IDA research staff members were reviewers of this document: Dr. Alfred E. Brenner, Dr. Norman R. Howes and Dr. Richard J. Ivanetich. The contributions of Mr. David Diskin, Ms. Sherrie Chubin, and Mr. Steve Stefanini, all of DISA, are gratefully acknowledged.

Table of Contents

SUMMARY S-1

CHAPTER 1. INTRODUCTION 1

1.1 PURPOSE 1
1.2 AUDIENCE 2
1.3 OUTLINE OF PAPER 2
1.4 OBJECTIVES OF THE EXPERIMENTS 2
1.5 BENEFITS TO DII COE DEVELOPERS AND COMMUNITY 3

CHAPTER 2. METHODOLOGY 5

2.1 INTRODUCTION TO DISTRIBUTED COMPUTING 5
2.2 BUILDING THE TIMING AND DATA MODELS 11
2.3 CONFOUNDING FACTORS 14
2.4 MITIGATION OF CONFOUNDING FACTORS 16
2.5 EXPERIMENTS TO BE CONDUCTED 17
2.5.1 Methodology of Experiments 17
2.5.2 Equipment and Software Used 18
2.5.3 Variations to be Performed 20
2.6 DATA WHICH WAS CAPTURED 21

CHAPTER 3. RESULTS OF MEASUREMENT 23

3.1 EXTRA EXPERIMENTS 23
3.2 DOUBLE PRECISION TIME RETURNED (MAIN EXPERIMENT 1) 23
3.2.1 Client and Server on Same Platform 24
3.2.2 Client and Server on Different SPARC20s 24
3.2.3 Server on SPARC20 and Client on Pentium 90 25
3.3 938 BYTE STRING RETURNED (MAIN EXPERIMENT 2) 26
3.3.1 Client and Server on Same Platform 26
3.3.2 Client and Server on Different SPARC20s 27
3.3.3 Server on SPARC20 and Client on Pentium 90 27
3.4 VTPLATFORM TRACK RETURNED (MAIN EXPERIMENT 3) 27
3.4.1 Client and Server on Same Platform 28
3.4.2 Client and Server on Different SPARC20s 28
3.4.3 Server on SPARC20 and Client on Pentium 90 28
3.5 ADDITIONAL INFORMATION 29

CHAPTER 4. SUMMARY OF OBSERVATIONS 31

CHAPTER 5. RECOMMENDATIONS FOR DESIGN 33

5.1 BUILD A PROTOTYPE TESTING CAPACITY 34
5.2 FINAL COMMENTS 37

LIST OF ACRONYMS 39

APPENDIX A. Glossary A-1

APPENDIX B. Attachments B-1

APPENDIX C. Source Code for Tests C-1

C.1 Double Precision Timer Source Routines C-1
C.1.1 Client C-1
C.1.2 Server C-2
C.1.3 Filters C-4
C.1.4 OBJECT INCLUDES C-6
C.1.5 Object Methods C-6
C.1.6 Interface Definition C-6
C.2 Character Sequence Source Routines C-7
C.2.1 Client C-7
C.2.2 Server C-8
C.2.3 Object Includes C-10
C.2.4 Object Methods C-11
C.2.5 Interface Definitions C-11
C.3 VtPlatform Source Routines C-11
C.3.1 Client C-11
C.3.2 Server C-13
C.3.3 Object Includes C-14
C.3.4 Object Methods C-15
C.3.5 VtTrack Interface Definitions C-15
C.4 Procedure Call Timing C-38
C.5 Null Procedure Timing C-39
C.6 Null Loop Timing C-39

List of Figures

Figure 1. One Client S4
Figure 2. Multiple Clients S5
Figure 3. Client/Server Communication Chart 7
Figure 4. Local Procedure Call 8
Figure 5. Remote Procedure Call 9
Figure 6. Experimental Monitoring Points 12
Figure 7. Client and Server on Same Platform 24
Figure 8. Client and Server on Different Sparc20s 25
Figure 9. Client and Server on Different Platforms 25
Figure 10. Experimental Monitoring Points 35

SUMMARY

Purpose

We performed a series of experiments to provide guidance to technical designers and developers about how to analyze the performance of new or legacy applications in order to engineer or re-engineer them for implementation in a distributed environment. We used IONA's Orbix, a commercially available Object Request Broker (ORB) that is an implementation of the Common Object Request Broker Architecture (CORBA), Version 2.0, developed by the Object Management Group (OMG)[1]. The main purpose of our experiments is threefold:

1. To understand resource expenditures associated with distributing objects between clients and servers that communicate using remote procedure calls.
2. To understand performance characteristics of distributed object computing using a simple example based on splitting the Global Command and Control System (GCCS) Track Correlation Application (TCA) into a Track Correlation Service (TCS) and a display client.
3. To provide a generic process for evaluating a potential design of a distributed application through the use of experimental measurements.

Background and Scope

The Defense Information Infrastructure (DII) Common Operating Environment (COE) project office will add distributed object technology products to future releases of the DII COE. This technology will implement the Object Management Group (OMG)'s Common Object Request Broker Architecture (CORBA), Version 2.0, and Object Management Architecture (OMA). This technology will lead to new design considerations which will have a large potential impact on the efficiency of applications.

Our set of experiments will enable designers and developers to:

---

[1] Some timing properties will be similar to those faced when using other distribution products based on the Open Software Foundation's Distributed Computing Environment (DCE) or Microsoft's Distributed Common Object Model (DCOM).

- Understand the feasibility of using OMG's OMA and CORBA as the software infrastructure for DII COE-based applications.
- Understand the effect of object distribution on the performance of DII COE based applications and services due to changing from a local procedure call paradigm to a remote procedure call paradigm.

By using a similar set of experiments based upon their own applications' requirements, designers and developers will be able to:

- *Develop a notion of granularity using the ratio of computation time to communication time.*
- *Develop a preliminary understanding of potential bottlenecks in the systems they design.*
- *Develop insight into timing latencies and dispersion[2] inherent in distributed systems.*
- *Establish a checklist of items which should be considered in the design and development of applications.*

Model and the Experiments

Distributed computing technology has been maturing over the past 15 years. In the distributed computing environment, Client/Server computing has become the dominant paradigm of developing distributed applications. Developers are responsible for providing client applications with a method for locating appropriate services hosting objects acting as servers. Using a CORBA Version 2.0 ORB, client objects specify the desired server object to the ORB and the ORB determines the location of the platforms hosting the server object, ascertains that the server object exists and is running, and causes the parameters issued by the client to be presented to the server hosting the service and then to the desired server object. Details of these conveniences are left to the implementors of the ORB instead of the developers of the application. Thus, in the model of computing that we investigated, there are a client object, a server object, and an ORB.

We defined a timing model of client to server communications for both a local procedure call (LPC) paradigm and a remote procedure call (RPC) paradigm, the latter based upon IONA's Orbix. Orbix offers eight points where we can measure time on a per-process basis

---

[2] Variation in the latency of remote communication.

within Orbix library code in a convenient fashion; in addition there are other points where we can measure time in the clients and the servers.

We performed three client to server communications experiments — communicating simple integer values, communicating a 938-byte string (representing a token GCCS track), and communicating a VtPlatform track (representing a "real" GCCS track) — on three sets of Client/Server configurations: Client and Server objects on the same Sparc20, Client and Server on different Sparc20s, and Client on Pentium 90 and Server on Sparc20. The set of experiments *assumes* that stable results can be obtained. Several confounding factors that are explained in this report can invalidate our experiments or invalidate their interpretation. These were mitigated to the extent possible.

Summary of Key Observations and Extrapolation

On a Sun Sparc20 hosting both client and server processes, a LPC which does no server work[3] costs about 1.5 *micro*seconds (μs); an Orbix RPC which does no server work takes about 1.5 *milli*seconds (ms), *or about 1000 times longer.* Communication between client and server objects uses about 1.3 ms of that Orbix RPC time.

Let efficiency be defined as the ratio:

(LPC Time + Server Work Time)/(RPC Time + Server Work Time).

In order to achieve 50% efficiency using RPC in this case, the work done by the server would have to take more than the time taken by 1000 LPCs or 1.5 ms, i.e.,

Efficiency = (1+1000) / (1000+1000).

In order to achieve 90% efficiency, the server work would have to take more than 9000 LPCs (9 RPCs) or 13.5 ms. This leads to the conclusion that procedures which are to be executed remotely on a Sparc20 should have sufficiently large granularity (very much greater than 1 ms) in order to amortize the inefficiency of RPC. Even larger granularity may be required to amortize other CORBA/Orbix inefficiencies.

Substantial time is used by communication between client and server objects: 1.3 ms with very small messages in each direction on the Sparc20 when client and server processes are

---

[3] This is the time required to ping the server. To avoid confusion, the term "server work" will be used to refer to the work performed by the called entity whether that entity is called by LPC or RPC. Client work will refer to the work that the client performs on the results provided by the server.

hosted on the same machine. More time is expended if the network is used. Substantial additional time is required for handling heterogeneity, e.g., for assembling and converting strings or structures for transmission.

Based upon our experiments using Orbix 2.0.1 and Solaris 2.5 for casual browsing of a database on a record by record basis, a Sparc20 used to host a server process may provide enough capability, but for intensive updating of information by a number of clients, it is unlikely that the Sparc20 will provide sufficient computing power to accomplish the job, as the following illustrative examples demonstrate.

Examples Derived from Measurements

Suppose we wished to divide the GCCS Track Correlation Application into a client and a server where track updates occur at a rate of 1000 per second. We might naively divide such an application into a display client and a track correlation service (TCS) which accesses a record in a database. Assume that tracks are stored in a memory cache, and that information is disseminated using RPC based on the API used to access the database of the current GCCS Track Correlation Application.

The One Client Case

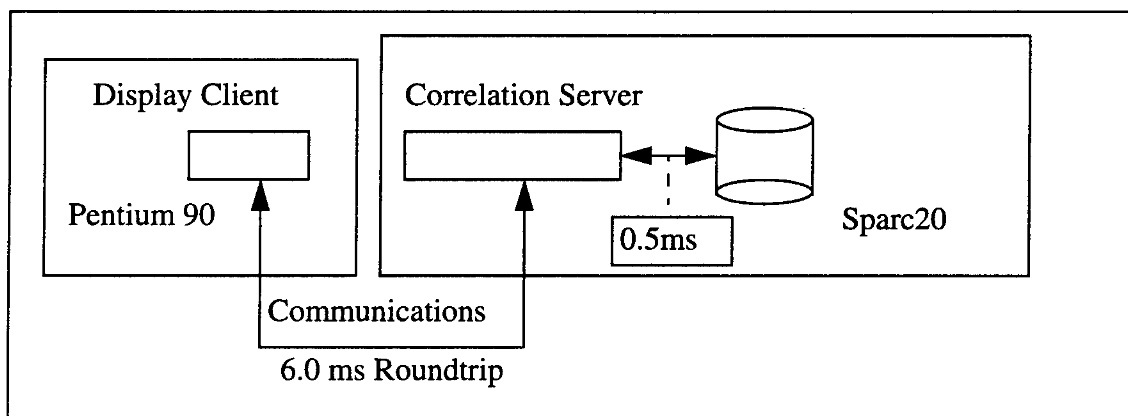


Figure 1. One Client

Suppose that the Sparc20 Server process could retrieve a record of 1000 bytes in 0.5 ms[4] from a database cached in memory for use in the same process. Obtaining that record using a Pentium 90 as client by means of RPC would add an additional 6 ms[5] for a total of 6.5 ms. The maximum number of serial reads per second is (1 second / 0.065 seconds) or about 150 per

[4] A hypothetical time for cached database access.

[5] Drawn from the data of Pentium90-based Client and SPARC-based Server.

second for this client, assuming that the client obtains a record, displays it, then obtains another record, displays it, and so forth. *Using this simple design we cannot achieve the goal of 1000 updates per second.*

The Multi-Client Case

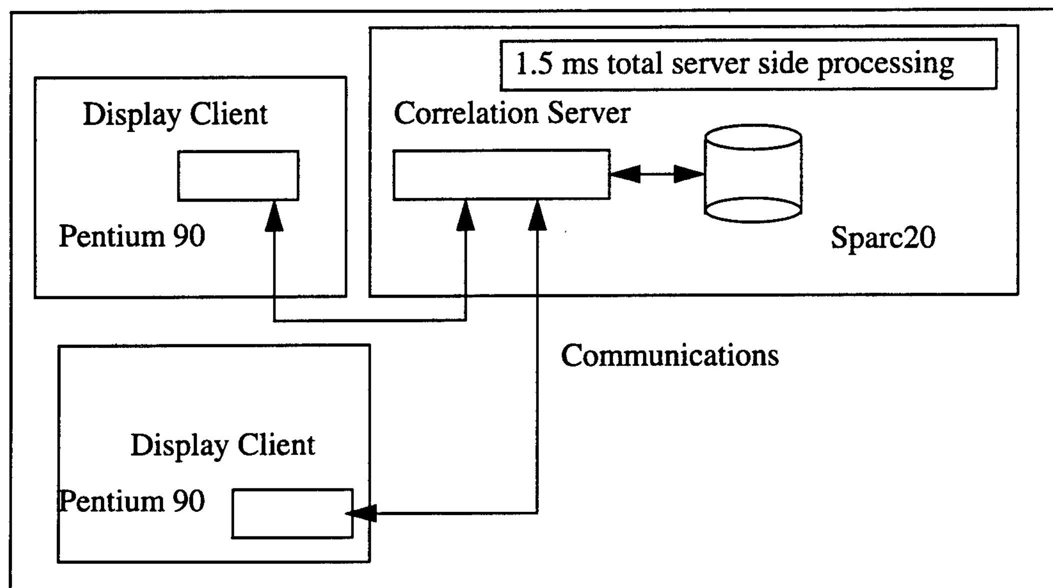


Figure 2. Multiple Clients

A second example: Suppose the total time used by the server's host Sparc20 for each client's request for a record was 1.5 ms including the read.[6] Five ms[7] is spent on the client platform, resulting in a total roundtrip time of 6.5ms. Then the maximum number of records per second which might be obtained in a simple request-response pattern from this server using a cached database would be on the order of (1 second/ 0.0015 second) or approximately 666 requests per second -- thus it could only support four Pentium 90 display clients at approximately 166 requests per second each.

We conclude, based upon our observations, that a naive division of the GCCS Track Correlation Application into a client and database service on Pentium 90s and Sparc20s using Orbix 2.0.1 is unlikely to be successful because of the computing and communication overhead required for distribution of the client and server. It remains to be seen whether a different strategy such as simultaneous transmission of multiple tracks would better utilize the processor so that the server work could be done in the time required. Design strategies must also be recon-

---

[6] Assuming that none of the time is spent waiting and based on timing of the SPARC20-based Server.

[7] Based on a roundtrip time of 6.5 ms including the data access minus 1.5 ms server time.

sidered if you use a higher performance processor, a different or improved ORB/IDL/Library combination featuring a better implementation of communications.

Recommendations

Based upon our experiments, we recommend that designers or developers should carefully consider the timing properties of the *delivery platform*(s) and of the software infrastructure during the design of *any distributed* application:

- Developers should create a *simple prototype* from which they will gain the information required for their design. For the targeted client and server platforms, designers should employ the hardware and software which will support the delivered application. Designers should determine the time to marshal[8] and unmarshal the data structures to be used. In addition they should measure the amount of communications time taken by each different kind of remote procedure call and the amount of time required to service each request. This should lead to a mathematical model of the amount of resources required to perform each kind of request. These times can also be used in simulations of the applications to be designed.
- Designers or developers should give special attention to the following factors:
  - Carefully consider the amount of computation to be done on the server for each request. Too little computation on the server reduces the overall efficiency of computation because of the communications overhead. If the overhead of a remote request is too high, implement the service as a library or collection of classes using LPC instead.
  - Carefully consider the type of argument(s) to be passed to the server or returned to the client. Try to make the argument as simple and as aligned as possible. Attempt to find an implementation of IDL that can perform marshalling, unmarshalling, and service in parallel, using threads. This is particularly valuable if the service performs I/O operations.
  - For computationally intensive services that can be parallelized, consider using multiple servers and distributing the computation (servers) on multiple platforms. Also try to hide communications latencies by using multi-threaded servers to handle multiple requests in a pipelined fashion. Find the sum of times on the paths which must be performed serially. This is the optimum time which

[8] See the body of the paper or glossary for definition of technical terms such as marshalling and unmarshalling.

can be achieved [one of Amdahl's laws]. One may think that using many machines executing a task in parallel pieces will reduce the total computation time, but the overhead introduced by the communication must be taken into account. Verify that the time for using multiple servers is shorter than running the calculation on a single machine using LPC.

- Carefully consider the factors which could render the experimental results invalid such as requests causing communications to other servers on the same host. Determine if they will contribute to your client-server workload. Especially consider the number of requests per second to database servers and the amount of I/O operations these servers are expected to perform.
- Carefully consider the confounding factors. Try to characterize them in order to determine if they will contribute to your client/server workload. If they will, identify mitigation strategies. Especially consider the number of requests per second to database servers and the amount of I/O these servers are expected to do.

As has been suggested in the illustrative examples, results vary depending on many factors. What is important is to experiment and model during design to determine what factors are most significant and what can be done to improve performance.

CHAPTER 1. INTRODUCTION

1.1 PURPOSE

This paper gives guidance to technical designers and developers about how they can analyze the performance of legacy or new applications in order to re-engineer them for implementation in a distributed object environment using the Object Management Group (OMG)'s Common Object Request Broker Architecture (CORBA)[1]. New distributed object- or procedure-oriented applications may also benefit from this methodology which is based upon conducting and utilizing the results of a series of experiments. Our experiments were made using IONA's Orbix, a commercial Object Request Broker (ORB) compliant with CORBA, Version 2.0. The main purpose of our experiments is threefold:

1. To understand resource expenditure required to support distributed computing for the purpose of distributing objects optimally between clients and servers that communicate using remote procedure call.
2. To understand performance characteristics of distributed object computing using a simple example based on splitting the Global Command and Control System (GCCS) Track Correlation Application (TCA) into a Track Correlation Service (TCS) and a display client.
3. To provide a generic process for evaluating a potential design of a distributed application through the use of experimental measurements.

A simple example will be used to show the kind of analyses which must be done before dividing an application into clients and servers and before attempting to decide on which platforms to place client and server. The example: dividing the GCCS Track Correlation Application into display client and track correlation server using the current track correlation database interface, while naive, is typical of the so-called "two-tier" approach where a client directly uses a data base by calling its interface API. It is not the intent of the paper to suggest that this

---

[1] Some of these considerations will be similar to those faced when using other distribution products based on the Open Software Foundations' Distributed Computing Environment (DCE) or Microsoft's Distributed Common Object Model (DCOM).

is the appropriate division of the TCA. It is the intent of the paper to use this example to motivate the experiments which were performed and to sketch the difficulties that such a design would face.

1.2 AUDIENCE

This paper is intended for technical designers and developers who must design distributed object applications using CORBA and OMA or for others who would like to understand the feasibility of applications in a distributed object environment.

1.3 OUTLINE OF PAPER

The paper is divided into four major sections, exclusive of the introduction:

1. Introduction — describes the objectives of the study and the benefits to be derived from it.
2. Methodology — describes the framework for experimentation.
3. Results of Measurement — provides the results of the experiments in detail.
4. Summary of Observations — summarizes what was discovered and attempts to generalize the results.
5. Recommendations for Design — provides a process for developers of distributed applications and a checklist of considerations for designers.

1.4 OBJECTIVES OF THE EXPERIMENTS

The Defense Information Infrastructure (DII) Common Operating Environment (COE) project office intends to add distributed object technology to future releases of the COE which will support an object-oriented distributed computing paradigm. This technology will employ software based on the consortium specifications known as the Object Management Group (OMG)'s Common Object Request Broker Architecture (CORBA), Version 2.0[2] and the OMG's Object Management Architecture[3] (OMA). The technology will lead to new considerations for design which will have a large potential impact on the efficiency of systems developed using this technology.

---

[2] Object Management Group. 1996. Common Object Request Broker Architecture, Revision 2. Object Management Group, Framingham, MA. 01701. July 1996

[3] Object Management Group. 1990. Object Management Architecture Guide, Revision 1.0. Object Management Group, Framingham, MA. 01701. November 1, 1990.

The set of experiments which were conducted are designed to help designers:

- Understand the effect of object distribution on the design of DII COE based applications and services.
- Understand the effect of changing from *local procedure call* (LPC) to *remote procedure call* (RPC) on design of applications.
- Understand the feasibility of using OMG's OMA for specific applications of important subdomains such as the Global Command and Control System (GCCS) including TCS.

1.5 BENEFITS TO DII COE DEVELOPERS AND COMMUNITY

By using a similar set of experiments based upon their own application's requirements and concept of operations, designers and developers of distributed object systems based on CORBA will be able to:

- *Develop a notion of granularity using the ratio of computation time to communication time.*
- *Develop a preliminary understanding of potential bottlenecks in the systems they design.*
- *Develop insight into timing latencies and dispersion inherent in distributed systems.*
- *Establish a checklist of items which should be considered in the design and development of distributed applications.*

CHAPTER 2. METHODOLOGY

2.1 INTRODUCTION TO DISTRIBUTED COMPUTING

Distributed computing technology has been maturing over the last 15 years. Originally, each solution was handcrafted, using whatever communication mechanisms were available. Later, low level communication services were standardized. The Berkeley community chose sockets and the Bell Labs community chose the Transport Layer Interface(TLI). Next, RPC was used to hide the complexity of socket programming and to make distributed programming obey the well-known procedure call semantics. RPC had higher overhead than socket programming but was much easier to use. Application-oriented languages/systems such as Linda[4] permitted programmers to develop applications that were of large granularity and computationally intensive with few data dependencies which could be executed in parallel on multiple machines using RPC.

In the late 1980s, the Open Software Foundation (OSF), now part of the Open Group (OG), developed a framework for distributed computing called the Distributed Computing Environment (DCE) which utilized RPC to access a set of standardized services. These services included security, time, and directory services among others. A complex application programming interface (API) was delivered in the early 1990s which permitted the development of distributed applications of high complexity and which facilitated optimization of those applications. An example of one of those applications is the Distributed File System (DFS) which features replication, access control lists, and distributed management tools. Another is Transarc's Encina®, a transaction manager. DCE utilizes procedural access to services.

In 1990 the Object Management Group was formed to promote distributed computing based upon the premise that all computing would be utilizing an object paradigm. Goals of this new mode were: location transparency; heterogeneity of platform, operating system, and programming language; and access via standardized interface. Where DCE programming was at an inherently lower level of detail, OMG's object programming[5] was at a level much closer to the application: the object request broker (ORB) was to hide many of the small details typical

[4] See **http://www.sca.com/linda.html.**

of DCE programming. Object services were to be specified that would provide a rich set of reusable components

In the DCE world of Client/Server computing, programmers are responsible for determining where Servers for their Clients reside, either using end-point mappings, a local directory service, or a remote directory service. In the ORB world, one specifies the desired service to the ORB using an object reference, and the ORB determines the location of the service, ascertains that it exists and is running, and causes the parameters issued by the Client object to be presented to the desired object within the service that instantiates the object. Details of these conveniences are left to the implementors of the ORB, instead of the developers of the application.

Thus in the model of computing to be investigated, there is an object with role of Client, an object with role of Server, and an ORB. The application we will investigate is a repetitive query in which the Client repeatedly interrogates the Server for the latest data values, waiting for a set of values to be delivered before asking for the next set. In the implementation to be tested: Orbix 2.0.1 for Solaris and Orbix 2.02 for Microsoft NT, the ORB is only involved in a few of the communications. The Orbix ORB determines the location of the Server for the Client, and initializes the service if necessary; then it mediates the selection of connection-oriented communication paths (transmission control protocol on internet protocol (TCP/IP) sockets) from Client to Server (and vice-versa) in communications 1-8 (See Figure 3). Until the Client decides to close the connection to the service, the connection between Client and service is maintained by the communications software. In the event that the Server fails, a Client request will return an exception. In this case the Client must request that the ORB reinitialize the failed server or locate another server and initialize communication with it. Our experiments will assume a failure free situation for simplicity. In our use almost all the traffic, generated by Client method invocations, proceeds from Client to Server to Client over the mediated socket connection provided by the ORB. Since the Server can support multiple objects, a dispatcher provided by the Orbix Library code on the Server platform uses the object reference to determine which object hosted by the server is being queried and which method of the interface to that object is being called and dispatches to it.

---

[5] OMG's object paradigm is interface based, utilizing RPC to invoke methods of objects which "hide behind" their interface.

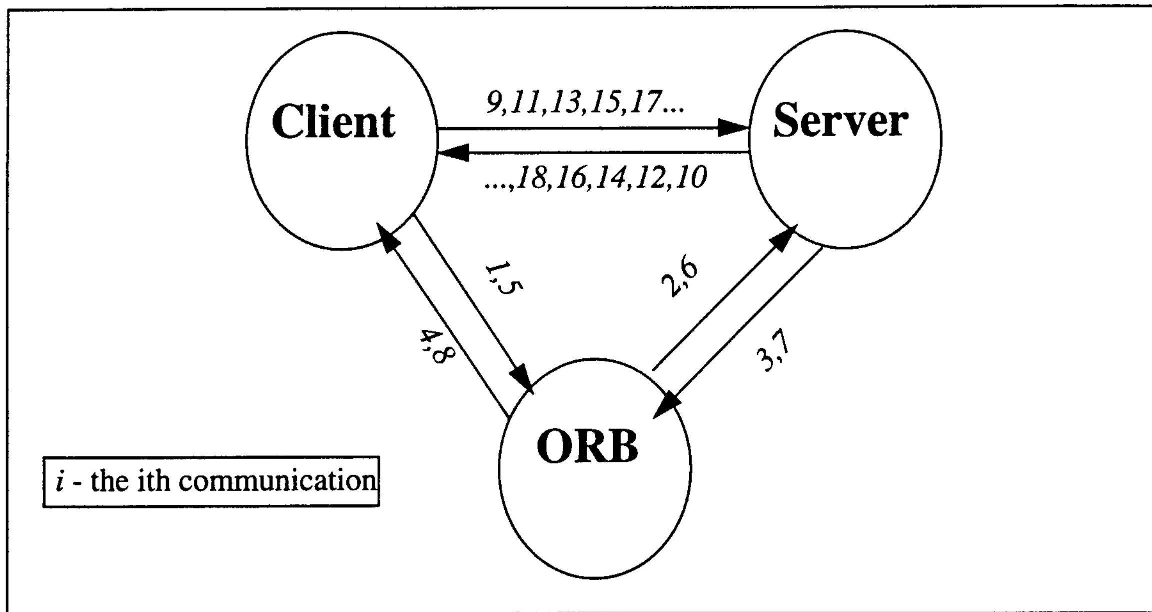


Figure 3. Client/Server Communication Chart

Assuming that there is a long-lived connection between Client and Server, we might wish to focus our attention on the communication between the Client and the Server in communications 9, 10, 11, 12, ... (unless the ORB is exceptionally slow) in the analysis of distributed object computing[6]. If only a few requests are issued for a service, we would focus attention on the time required for mediation by the ORB and on the cost of opening and closing connections. Such would be the case if we were designing a collaborative planning or network browsing application. In that case the nature of the experimental investigation would be different.

Since our example primarily demonstrates the communication between Client and Server, we note that the difference between the distributed and the non-distributed case is the use of Remote Procedure Call (RPC) in the former case and Local Procedure Call (LPC) in the latter case. We can define relative efficiency to be the ratio of the time taken for the RPC + Server Work[7] to that of LPC + Server Work. Figure 4 and Figure 5 illustrate a model of the two cases.

---

[6] Under some circumstances, the Client might bind itself to an object in the Server, request a few invocations, and then disconnect, possibly reconnecting later. In our examples, the Client would connect once and remain connected for a very substantial period.

[7] To avoid confusion, the term "server work" will be used to refer to the work performed by the called entity whether that entity is called by LPC or RPC. Client work will refer to the work that the client performs on the results provided by the server.

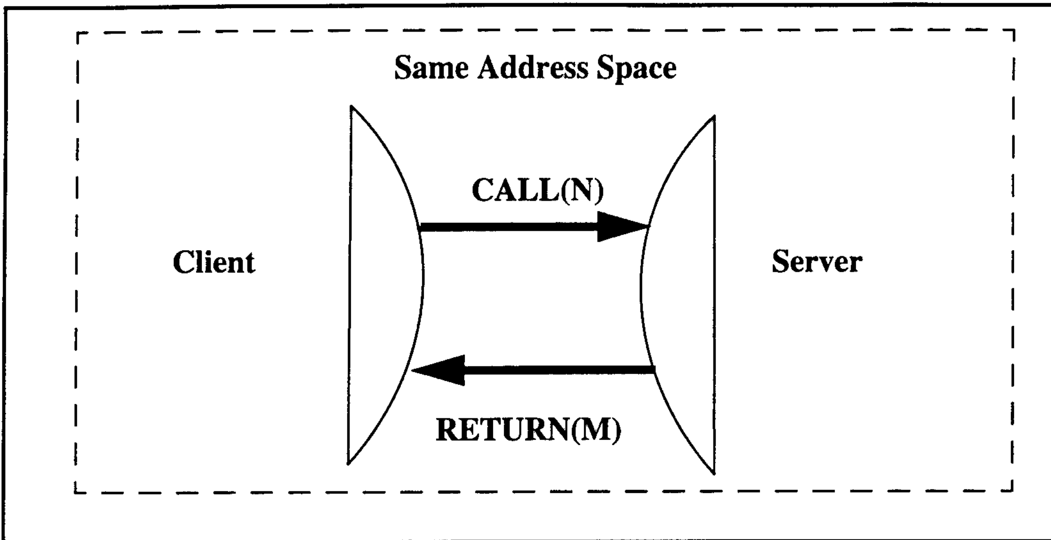


Figure 4. Local Procedure Call

Eqn. 1: $$\tau|_{LPC} = \tau|_{CALL(N)} + \tau|_{SERVICE} + \tau|_{RETURN(M)}$$

Let $\tau|_x$ be the amount of time taken for the part of the local procedure call labeled x, then the time for a local procedure call is given by Equation 1 above, where *N* represents the number of bytes transferred by the caller and *M* represents the number of bytes returned by the service. Figure 4 is incorrect for system calls because the typical operating system call involves the use of different address spaces: the system address space and the application address space. The reason this is important is that "call by reference" can be employed within the same address space, but that "call by value" must be used between two different address spaces. The latter requires the allocation of space and copying of values which may involve significant overhead.[8] It also may require two process exchanges to give the hardware access to the new address space and to return and may result in multiplexing the use of resources. The use of local procedure call within a single address space is the least expensive of all service use in terms of time expended.

---

[8] A typical local procedure call on a Sun SPARC20 (75Mhz) requires 1 microsecond whereas a typical instruction may require but 2 nanoseconds and a process exchange 100 nanoseconds.

Remote Procedure Call is the most expensive use of time. It is illustrated in Figure 5.

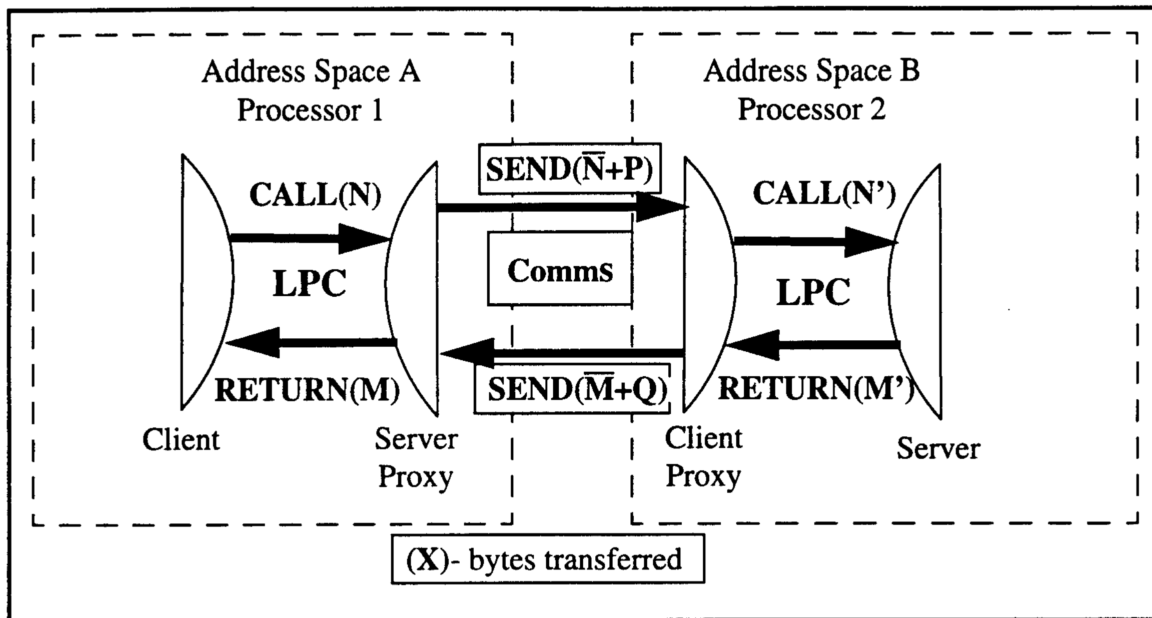


Figure 5. Remote Procedure Call

In this case there are two address spaces involved, potentially on two different machines. The Client and software which acts as proxy for the Server are in one address space and software which acts as a proxy for the Client and the Server are in a second address space, potentially on a different machine. The following is the conceptual sequence by which remote procedure call is accomplished:

- The Client calls the proxy for the Server and provides arguments to it, either by value or by reference because it is in the same address space.
- The proxy for the Server marshals the arguments, i.e., it obtains the values for the N bytes of arguments, converts them to a standard representation, following standard alignment rules which may require padding with null bytes. Then it passes this set of $\overline{N}$ bytes (which probably will be larger than N) to the circuit based communication protocol (in this case TCP/IP sockets) which reliably forwards the message along with P protocol bytes to the socket on the other platform.
- The $\overline{N}$ bytes are received by the proxy for the Client which unmarshalls the arguments: i.e., it converts the bytes from standard form and alignment to N' bytes of the proper form and alignment for the second platform.
- The proxy for the Client then calls the object that the Server has made available to the Client, selects the method specified by the Client, and calls this method using

the N' bytes of arguments supplied by the Client using a local procedure call which may be either by value or by reference.

- The service performs its computation, returning its result of M' bytes to the proxy for the Client.
- The proxy for the Client marshals the returned values into a data structure of $\overline{M}$ bytes which it passes through the communications protocol with Q protocol bytes to the socket on the Client's platform.
- There the proxy for the service unmarshalls the returned value and returns these M bytes to the Client.

The timing equation for RPC is as follows:

Eqn. 2:

$$\tau|_{RPC} = \tau|_{CALL(N)} + \tau|_{Marshal(\overline{N})} + \tau|_{Send(\overline{N}+P)} + \tau|_{Unmarshal(\overline{N})} + \tau|_{CALL(N')} + \tau|_{SERVICE} + \tau|_{RETURN(M')} + \tau|_{Marshal(\overline{M})} + \tau|_{Send(\overline{M}+Q)} + \tau|_{Unmarshal(\overline{M})} + \tau|_{RETURN(M)}$$

It conveniently omits the detail of additional overhead which may be added utilizing the level of security which may be desired. For example, if we desire confidentiality, time for encryption and decryption must be added. If we want to guarantee integrity of messages, integrity bits must be generated and added to the message and must be checked at the receiver. If the Client and/or Server do not trust one another, additional time must be added for the purpose of authentication. These additional times are shown in Equation 3.

Eqn. 3:

$$\begin{aligned}\tau|_{RPC} = {} & \tau|_{CALL(N)} + \tau|_{Marshal(\overline{N})} + \tau|_{Send(\overline{N}+P)} + \tau|_{Unmarshal(\overline{N})} \\ & + \tau|_{CALL(N')} + \tau|_{SERVICE} + \tau|_{RETURN(M')} + \tau|_{Marshal(\overline{M})} + \tau|_{Send(\overline{M}+Q)} + \tau|_{Unmarshal(\overline{M})} \\ & + \tau|_{RETURN(M)} + \tau|_{IntegrityGenerate(\overline{N})} + \tau|_{IntegrityCheck(\overline{N})} + \tau|_{IntegrityGenerate(\overline{M})} \\ & + \tau|_{IntegrityCheck(\overline{M})} + \tau|_{Encrypt(\overline{N})} + \tau|_{Decrypt(\overline{N})} + \tau|_{Encrypt(\overline{M})} + \tau|_{Decrypt(\overline{M})} \\ & + \tau|_{AuthenticateClient} + \tau|_{AuthenticateServer}\end{aligned}$$

We note that in the simple case of RPC (Equation 2) we have two calls and two returns. There are also fixed overheads of two marshals and two unmarshals plus a fixed time component associated with *send* and a variable component associated with *send* but actually related to latencies due to scheduling processes on the platforms which host the Client and Server

objects. The time for *send* is inclusive of both the fixed and the variable times. The figure in Appendix B, Attachment 3 shows the ratio between LPC + Server Work and RPC + Server Work as a function of the time required for the work on the Server. As can be seen, the more time taken in the service, the higher the relative efficiency.

In order to get a better idea of the actual efficiency, it will be necessary to measure actual overheads in several styles of implementations. Ideally, we would determine the actual times for a specific service and there would be no variation. In practice we will not be able to do this, but we will be able to determine "general orders of magnitude" which will help us design applications for which our experiments are relevant.

2.2 BUILDING THE TIMING AND DATA MODELS

In building a timing model, we need to understand where we can measure and the accuracy to which time may be measured. A CORBA compliant ORB, Iona Orbix 2.x has been specified for use in the DII COE. Orbix offers eight places (Points 1-8 in Figure 6) where time can be measured on a per-process basis in a convenient fashion[9]; in addition there are other points (e.g., Points 0 and 9 in Figure 6) where we can measure time in the Client or in the Server. Points 0-9 in Figure 6 are used in our experiments. In addition points 10 and 11 mark places where the number of bytes used in communication are measured. The times at which events are measured and the number of bytes transmitted are denoted as indicated by the following list.

0. In the Client object, τ_0.
1. Prior to the point that the proxy for the Server marshals the arguments, τ_1.
2. After the proxy for the Server has marshalled the arguments, τ_2.
3. Prior to the point that the proxy for the Client unmarshals the arguments, τ_3.
4. After the proxy for the Client has marshalled the arguments, τ_4.
5. Prior to the point that the proxy for the Client marshals the arguments, τ_5.
6. After the proxy for the Client has marshalled the arguments, τ_6.
7. Prior to the point that the proxy for the Server unmarshals the arguments, τ_7.
8. After the proxy for the Server has unmarshalled the arguments, τ_8.

[9] Orbix allows interception on a per object or per process basis. Since our service has only one "object" we will use the per process interception. Using it we also see all requests of the Client process to the ORB and the ORB process to the Server process. These requests result in the initialization and destruction of the communication paths between Client and Server.

9. In the Server object, τ_9.

10. Total number of bytes transmitted by the Server proxy, B1

11. Total number of bytes transmitted by the Client proxy, B2

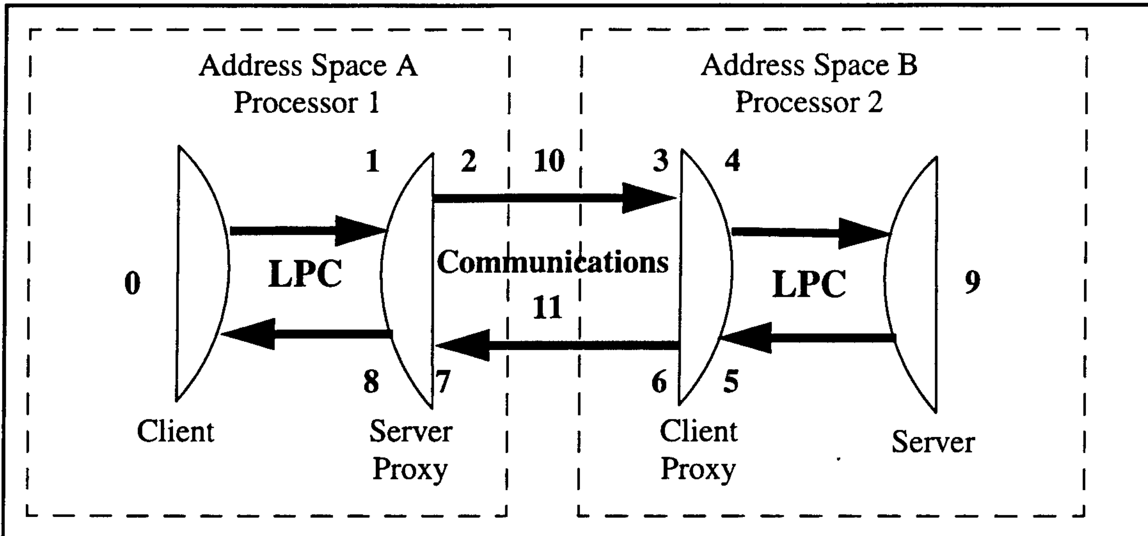


Figure 6. Experimental Monitoring Points

Note that the time for marshalling can be determined using points 1 and 2 as well as 5 and 6. Unmarshalling time can be determined using points 3 and 4 as well as 7 and 8. Service time can be determined by using the difference of the times at points 4 and 5. If the Client process and the Server process are both on the same machine, then points 3 and 4 can be used to measure the transmission of $\overline{N}$ + P bytes and 6 and 7 to measure the transmission of $\overline{M}$ + Q bytes.

This system permits two kinds of time measurements. One is a high resolution measurement in multiples of 500 nanoseconds; the other is in hundredths of a second. The latter time can be used to measure wall clock time or time spent in the (UNIX) User's address space or time spent in the operating system. In order to get the best timing possible, we used the high resolution measurement in the Client and the Server in addition to the points 1-8 per Figure 6, when that was possible. We were not always able to measure individual events, but instead measured the time required for repeated occurrence of the same event and worked in terms of averages.

The introduction of timing measurement can perturb activity in the Client address space or the Server address space. It is important to assure that the timing is reasonably stable and as free of noise due to asynchronous events as possible. Assuming this is done, one should measure the following:

1. Typical cost of local procedure call in same address space and to system address space (See Appendix C.5)
2. Typical cost of loop overhead (See Appendix C.6)
3. Typical cost of measurement activity, i.e. the cost of an interceptor. In our case, we did this by timing the 9-point example and the 5-point example. We subtracted the minimum time for each and assumed that this time resulted from the addition of the interceptor code for four interceptions.
4. Ratio of best-case LPC to best-case RPC, i.e., an LPC which does no server work and an RPC which does no server work. The minimal times expended are best case, because they are the shortest times we can obtain.
5. Typical best-case time for marshalling and unmarshalling of arguments, in our example a string of 938 bytes. See Appendix C.2.
6. Typical best-case time for transmission of arguments, in our case, 938 bytes of payload + protocol and tagging data. See Appendix C.2.

The reason we have selected 938 bytes is that this is the length of the information taken from the data base of the TCS of GCCS. We have selected this service because it seems to present the most difficult challenges to a successful implementation.

Assuming that these times can be obtained, we will be able to determine the effect of RPC on allocation of function in Clients and services utilizing information obtained. *Of course these values are guaranteed only for the experiments made in the environments provided.* By making slight variations in environment, we can determine how sensitive the results are as a function of the environment. Using the results of the experiment is more likely to produce better factoring of our applications into pieces which can be distributed than using our judgment alone.

2.3 CONFOUNDING FACTORS

The set of experiments assumes that stable results can be obtained. The following factors can invalidate the results of these experiments or invalidate their interpretation in a more general setting; our experiments attempted to mitigate these factors as described in Section 2.4.

1. Lack of high precision timer on Pentium 90. A preferred Client for the DII COE is likely to be an Intel based computer running the Microsoft NT 4.0 operating system. Regrettably, there appears to be no way to access a high resolution clock on NT4.0 clients. A 60 Hertz clock is available, but this produces minimum increments of 16.66 ms which are only useful for measurement of averages and give us less information about the distribution of times required for satisfaction of a request.

2. Lack of high accuracy coordination of times between Client and Server. The high resolution clocks on two different SPARC machines are not correlated, each measures its own time, presumably from when the system was booted. Because we do not have an accurate global clock, there are a number of measurements which we cannot make with accuracy. If measurements were totally stable, it might be possible to estimate an offset for one of the clocks from the other. Unfortunately, the measurements are not stable enough for this purpose.

3. The work required for reading a system clock may distort measurements. This can be due to activities involving process exchange which may cause each machine's scheduler to introduce latencies into the desired calculation.

4. In addition to 3, the subroutines required for measuring time at various points in the Client-Server-Client loop may distort measurements in a non-linear manner. Because of the variance in iteration time, it is difficult to subtract out these times precisely. However we can gain an understanding of the magnitude of the time required to take the measurements and verify that this time is a small percentage of the total Client-Server-Client loop.

5. Degree of burstiness and simultaneous applications. The experiments involve running a single application at a time. In the usual case, multiple applications will be running simultaneously. Use of an ethernet as a communications medium produces a latency vs. bandwidth requirement which is very non-linear after 50% of the bandwidth is used. There may be substantial dispersion in times for throughput, especially under heavy load. The results obtained in this experiment are for lightly loaded systems and will not necessarily apply under heavy loading.

6. Security processing overhead. These experiments have ignored all effects of providing security as per Equation 3. In cases where these functions must be applied, the developer must attempt to understand the magnitude of their contribution.

7. Non-linearities in number of IP packets/second processed. Typical work stations reach a point at which they can no longer process packets. On older work stations such as the Sun IPX, this point is around 200 packets per second. On newer ones, this limit is higher, but still may be less than 1500 packets per second (of any size).

8. Non-linearities in bandwidth of IP packets/second required for transmission. Each medium and access method has different characteristics. This set of experiments assumes an ethernet packet whose maximum size is slightly more than 1500 bytes including protocol information. When more than 1500 bytes are to be transmitted, the original messages are segmented and sent as multiple packets, producing a non-linear requirement for bandwidth as a function of message size. This effect will be even more noticeable with Internet Protocol Version 6 with large addresses enabled, since many more bytes of overhead are introduced. If Asynchronous Transfer Mode (ATM) transmission is used, similar non-linearities may be observed. This is because the TCP/IP and ATM error determination and recovery mechanisms are not designed to work with one another and ATM error rate is permitted to be high, often resulting in multiple re-transmissions of ATM packets or error in the IP transmission which can require complete re-transmission of the TCP/IP packet.

9. LAN/WAN interfaces and queuing. This experiment has assumed that both Client and Server are either on the same platform or that alternatively they are on platforms which are on the same Local Area Network (LAN). There is a possibility that they are on different networks with a Wide Area Network (WAN) connecting the LANs. In this case attention must be given to the queueing that takes place at the interface where there is customarily a large bandwidth differential, e.g., 10 mb/s and 9600 bps.

10. The ORBIX IDL Compiler. The interface definition language (IDL) compiler which IONA uses to produce C++ implementations of ORB-usable interfaces is in a state of flux, improving at each new version as does their library of supporting proce-

dures. As noted by Schmidt [10] this translation has a very strong effect on the efficiency of marshalling and unmarshalling, particularly in the case of large records.

11. Possibility of caching. The possibility of caching means that we must ascertain that an appropriate amount of data is transmitted — that the mechanism is not only transmitting changed data. In a production case we would attempt to engineer smart proxies to take advantage of repetitive, unchanged data.

12. Lack of time to try detailed optimizations. Even though the experiments were envisioned as exhaustive, there was not enough time or resources to try all possible optimizations which a developer might perform in order to meet required performance goals or to use proprietary extensions of Orbix other than filters for time measurement or smart proxies.

2.4 MITIGATION OF CONFOUNDING FACTORS

The experiments were conducted in a manner which attempt to mitigate the confounding factors as much as possible.

- Factors 3 and 4. The experiments attempt to account for extraneous times and remove them.
- Factors 7 and 8. The experiments used an isolated subnetwork with minimum extraneous traffic and linear duty cycle; records were made of the network statistics in an effort to substantiate operation in this regime.
- Factor 9. A WAN was not employed; all communications were performed on a single segment of a LAN. To gain an understanding of the interprocess communication, an experiment using loopback TCP/IP was used to establish basic timing for communications and for both Client and Server processing times on the same SPARC20.
- Factor 12. At least one change in returned value was made at each iteration and the number of bytes and messages transmitted and received were checked to ascertain that an intelligent caching scheme was not being used.

In addition the following practices were followed:

[10] Gokhale, Aniruddha and Douglas C. Schmidt. 1996. Measuring the Performance of Communication Middleware on High-Speed Networks. ACM SIGCOMM Conference Proceedings. Association for Computing Machinery, New York, N.Y. August 1996.

- Only the standard Basic Object Adapter defined for all implementations of CORBA Version 2.0 was employed. The interface definitions and Client and Server code in C++ are provided so that an interested party can produce similar results.
- Complete records of each experiment were retained for further examination.

2.5 EXPERIMENTS TO BE CONDUCTED

The goal of the measurements which were made were to try to determine the parameters of Equation 2 using a variety of host platforms for the *Client* and obtaining the most information possible with the fewest experiments: one returning a double precision value, another returning a 938 byte string, and a third returning a VtPlatform track (see glossary) using the API of the Track Correlation Application Database Manager.

2.5.1 Methodology of Experiments

Each experiment was conducted to:

1. Determine the amount of time required and number of bytes of data transmitted for a Client/Server interaction. This was accomplished by using a loop iterator to repeat the RPC over and over again. This showed us the maximum rate that a Client-Server-Client interaction can proceed.
2. Break down the time required per interaction into the time required on the Client platform, the time required on the Server platform, and the communications time (as accurately as possible).
3. Break down the time required on the Client platform into the time required for Client marshalling, Client unmarshalling, and Client processing.
4. Break down the time required on the Server platform into the time required for Server unmarshalling, Server processing, and Server marshalling.
5. If possible, decompose the communication time into time spent on Client platform, time spent on Server platform, and time on the "wire" as a function of the amount of data and kind of data.

Each experiment was performed in two basic varieties. In the first we simply noted the value of the real time clock when the processes reached each measurement point. Of greater interest is an experiment related to the TCS data base. The TCS data base contains a variety of data records from all sorts of data sensors, e.g., radar, sonar, acoustic, etc., which report positions of "elements" at a given instant of time. The purpose of track correlation is to attempt to

fuse the data from various sources such as electronic intelligence, heat sensing, signal intelligence, and human intelligence, deconflicting multiple instances of the same track from multiple sensors. As a result, the TCS data base contains 12 different kinds of records. A simple kind of display Client might request the return of track data so that it could present the information on a display. Instead of determining the type of record and requesting it specifically, we would prefer to use a generic request to return a record which has been appropriately marshalled and unmarshalled so that it is usable on both Intel and Sun Platforms with their different representations of the same data. The marshalling routines must determine which data record format is actually being returned and marshal and unmarshal it correctly. This problem is representative of all generic data base requests. Our experiments were expected to produce different results for the timing of each kind of record and they did. Two kinds of records were actually employed, a record of 938 bytes, and a record containing a VtPlatform record — see Appendix A.2.3.

We performed the above for a Client/Server interaction returning a very small amount of data, a string of data approximating a track from the TCS, and a structure of data which is one of the 12 types of tracks sent by the correlation Server.

2.5.2 Equipment and Software Used

We used equipment available to us at the time, including two Sparc20s, a Sparc IPX, and a Pentium 90. We are fairly certain that the platforms had sufficient memory and processing power to participate in the experiment. *While the Server platforms and Client platforms of the DII COE used in a production environment will have substantially larger configurations and use more capable networks, the results are still representative and scale with processor speed since only a few microseconds are used for transmission of the values on network media. (One would have to test to assure that most of the time is lost due to processing and that this time is inversely proportional to the speed of the processor.).*

The Sparc20 was equipped with a 75Mhz clock and 96 Mb of memory. It was always used to host the Server process and as the host for the Orbix 2.0.1 ORB which is CORBA 2.0 compliant. This Sparc20 was also used to host the Client so that loopback TCP/IP[11] could be timed precisely and so that all 9 points could be measured with the same clock, giving the effect

[11] In loopback TCP/IP, the packets which would have been transmitted over the network are simply reflected by the media control and processed as input without having been transmitted over the network. The timings for this are slightly less than if the data had actually been sent over the network.

of a global clock. It should be noted that both Client and Server were single threaded which means that the Client waits for the Server to respond and vice versa.

A second Sparc20 was equipped with a 75 Mhz clock and 64 Mb of memory. It was used exclusively as a second Client. Although its 4 measurement points did not use the same clock as the platform hosting the Server, its clock was as accurate. Since the cycle of Client-Server-Client was still employed, the two different experiments could be compared for the sake of consistency.

A Sun Sparc IPX with 24 Mb of memory was used as a Client to verify the effects of a slower processor with similar data representation. The results of these experiments are presented in spreadsheet in Appendix B, Attachment 1 for completeness, but are not otherwise discussed.

A Pentium 90 with 16 Mb of memory running Microsoft NT4.0 was used as a Client because it was available equipment. It is not likely that a Pentium with such low clock rate or limited amount of memory will be used as deployed equipment in DII 3.2. *More likely, a 200 Mhz Pentium would be employed; it is about twice as fast.* The ethernet controller on the Pentium 90 was a 3Com 3C509, an unoptimized controller.

The ethernet used consisted of a central hub, a router, and optical cable to the equipment. Equipment on the ethernet was limited to the 2 Sparc20s, an IPX, and the Pentium 90. In order to use files from the author's machine, a file system from that machine was mounted on the Sparc20 using the Network File System of Sun Computer. The use of this convenience may have introduced additional variance in the results since this use results in the Sparc20 pinging the host on an infrequent basis to ascertain whether it is alive. Based on the results presented in Appendix B, Attachment 1, this use did not greatly add variance to the results.

The operating system on all the Sun machines was Sun Solaris Version 2.5. It is expected that DII COE will support this at Revision 3.1. (It supports 2.4 at DII-COE Release 3.0). The version of Orbix on the Sparc was Orbix 2.0.1 which is not a multi-tasking version. A multi-tasking version would have made more sense if the ORB had been heavily loaded or if the Server had managed multiple objects. The Sparc C++ Compiler was version 4.0.1 (SC3.0.1) as required by IONA for operation with Orbix Version 2.0.1. The Sun version of *time* and *netstat* were used to accumulate centisecond time and network statistics.

The operating system on the Pentium 90 was Microsoft NT Version 4.0 with no patches. Orbix 2.0.2 for NT was used as the Client ORB. The compiler/library suite used was Microsoft Visual C++ Compiler, Professional Version 4.2. Minimal changes were made in the source

code of the Clients previously compiled and run on the Sun. These changes, documented in the Orbix documentation, accommodated the compiler's use of macros to handle declarations of nested classes.

2.5.3 Variations to be Performed

Sets of trials were run for each experiment, using four different platforms for Clients to gain a sense of robustness.

First, each basic experiment was run using nine time monitoring points — the eight as mentioned above and an explicit request in the Server for high resolution time. These were run on the Sparc20 platform which hosted both the Client and the Server as well as the pair of Sparc20s. This permitted us to gain a view of the effect of marshalling and unmarshalling services of greater and greater data complexity. While performing these experiments, we also collected *netstat* statistics and Unix centisecond timing as described previously in Section 2.2. This gave us confidence in the facts that

1. Our network was lightly loaded.
2. The number of bytes and the number of messages transmitted were as expected.
3. Our high resolution timing measurements were consistent with the low resolution measurements made over the entire experiment.

Second, we disabled the four collection points in the Client, and ran each experiment again with Client and Server on the same platform. This left five collection points in the Server. We did this because we ran this experiment across the four different Client options and we knew that the Pentium 90 does not have a high resolution clock equivalent. Third we ran the Client compiled for the Sparc20 on the second Sparc20 to see the effect of using loopback versus using the network. Fourth, that same Client was run on the IPX to see the effect of performance of a slower Client. Finally, the source of the Client was transported to the Pentium, modified as required for Microsoft C++, and run as a Client on the Pentium 90 under Microsoft NT4.0. In this last case the Microsoft NT version of *netstat* was used to collect network statistics. This set of results in profile showed consistency across the Sparcs and showed diversity in the Sparc vs. Pentium with respect to marshalling, transmission times, number of messages, etc.

The first experiment involved sending an integer value from Client to Server. See the code in Appendix C.1. The Server returns a double scalar which is the high resolution time. This experiment provides basic performance loop time against which all other results can be

compared. It must be performed enough times so that the experimenter can have relative confidence in the results — in all cases 10,000 times. The number 10,000 was chosen because this number of observations could be collected in random access memory for each of the collection points and sent to disk after the test was complete, minimizing the effect of the printing on the experiment. Capturing each value allowed display of the distribution of round-trip timing values as a histogram. Its results help us to understand the ratio of overheads for LPC versus RPC in the simplest case.

By comparing the nine point experiment running on the pair of Sparc20s to the five point experiment running on the Sparc20s we may determine how much time is used in four calls to the time monitoring routines using the Orbix framework.

The second experiment requires sending an integer value from Client to Server. See the code in Appendix C.2. The Server returns a double scalar which is high resolution time and a 938 byte string to determine marshalling/unmarshalling time and communications time for sending *strings*. This experiment helps us to understand the overhead of using a fixed length string of bytes as the answer to a request. The time for this should be lower than for returning any structured value of the same length.

The third experiment requires sending an integer value from Client to Server. The Server returns a double scalar which is high resolution time and a 938 byte structure to determine marshalling/unmarshalling time and communications time for sending *structures*. This structure is our representation of the VtPlatform Track, a 938 byte complex structure described in Appendix C.3. This experiment tells us the overhead of returning a typical structured value from the data base.

Additional experiments were also conducted to determine the time for:

1. A do loop containing no server work. See Appendix C.6.
2. A loop of subroutine calls to a null routine returning a void result. See Appendix C.5.
3. A loop of subroutine calls to the high resolution clock. See Appendix C.4.

2.6 DATA WHICH WAS CAPTURED

In the three experiments, the following data was captured for each Client/Server pair:

1. High Resolution time for each Call-Service-Return-Process Loop (10,000 values).
2. Real, User, and System Time (Unix System).

3. High-resolution times at each collection point.
4. Network statistics: before and after Client and Server runs.

In the additional experiments, the following data was captured:

1. High-resolution time to perform 1 billion null iterations of a for-loop.
2. High-resolution time to perform 1 billion subroutine calls returning a double scalar type.
3. High-resolution time to perform 1 billion subroutine calls to the high-resolution clock.

The following information was derived from the values collected (See Attachment1 in Appendix B):

1. High-resolution times for: Server Proxy Pre & Post Marshal, Client Proxy Pre & Post Unmarshal, Client Proxy Pre & Post Marshal, Server Proxy Pre & Post Unmarshal.
2. Roundtrip average times, maximum times, minimum times, standard deviation, histogram.
3. Times for: Server Proxy Marshal, Client->Server or Client +Transport (When different clocks are used), Client Proxy Unmarshal, Server Execution, Client Proxy Marshal, Server->Client or Server +Transport (when different clocks are used), Server Proxy Unmarshal, and Client Execution.
4. Input Packets, Bytes, ipInDelivers, "msgs" (transmitted): to see the effect of dynamic tagging and alignment of byte strings and structures, and to see how many messages are transmitted, and to verify that the network is relatively inactive.

CHAPTER 3. RESULTS OF MEASUREMENT

A spreadsheet detailing results in capsule form is included in Appendix B as Attachment 1. This chapter will summarize the extra experiments and comment on each of the three main experiments in sections below.

3.1 EXTRA EXPERIMENTS

The extra experiments are required to help us deal with the confounding factors 3 and 4. They also provide a good idea of the capability of the platform.

The execution of an add instruction on a Sparc architecture takes about one instruction cycle and a load instruction takes between 3 and 6 cycles. A cycle on a 75 Mhz Sparc20 is 1/75,000,000 or 13 nanoseconds (ns).

One billion executions of a for-loop with no executable part required an average of 80.6 ns or about 6 cycles.

One billion calls in a for-loop to a time routine with a null body yielded an average of 1237.8 ns per loop iteration or 1157.2 ns. per subroutine call or about 89 cycles.

One billion calls in a loop to the high resolution clock (presumably through the operating system) yielded an average of 1551.2 ns per loop iteration or 1470.6 ns per call. Note that this was about 313 ns more than the null subroutine or an extra 24 cycles.

3.2 DOUBLE PRECISION TIME RETURNED (MAIN EXPERIMENT 1)

The following is the interface specification for the first experiment.

```
// a simple IDL interface: time. Objects of this interface provide an
// operation 'hr_time' which takes an unsigned long value which may signal
// termination of the Server and returns the double real-time in nanoseconds
// (if vin does not signal termination of the Server).

interface time {double hr_time (in unsigned long vin);};
```

Only the results obtained from the five point experiments are discussed below. They are quite similar to the results of the nine point experiments.

3.2.1 Client and Server on Same Platform

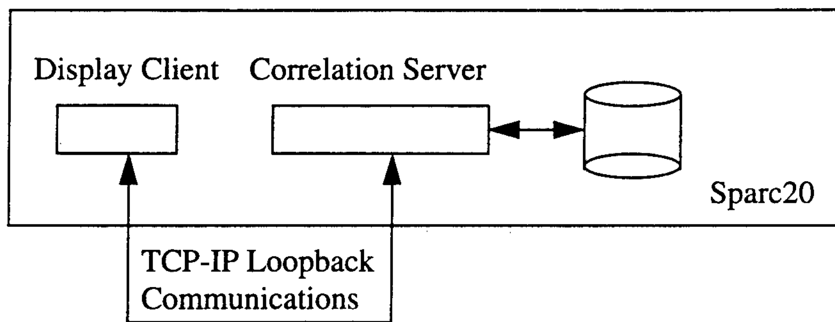


Figure 7. Client and Server on Same Platform

A histogram of the times required for the complete execution of the iteration of a call from the Client to the Server and return was obtained for the 10,000 invocations of the time interface. The histogram portrays the variability of the time for an iteration and gives an idea of the distribution of the values (for this experiment). In all cases the "tail" of the distribution is very long. A sense of how long can be obtained from the ratio of minimum time to maximum time for an iteration or by alternatively noting that the standard deviation is typically on the order of 1/4 the mean.

A complete invocation of the loop is denoted a roundtrip. The shortest[12] roundtrip on same platform for the interface above is ~1.618 ms or 1043 times the per-loop high-resolution-time local procedure call time. The longest roundtrip in the same trial was ~25.3 ms or 16,318 times this basic time. The average time was 1.692 ms and the standard deviation was 0.369 ms about 22% of the mean value. The time from Client to Server was 0.939 ms and from Server to Client was 0.364 ms, yielding a total of 1.303 ms for transmission.

3.2.2 Client and Server on Different SPARC20s

A histogram of the times required for the complete execution of the iteration of a call from the Client to the Server and return was obtained for the 10,000 invocations of the interface. The minimum round-trip time was 1.589 ms, less than the minimum time for the co-hosted Client and Server. The maximum time for a round trip was 134.7 ms, exhibiting much larger dispersion. The average time for round trip was 1.649 ms and the standard deviation was 1.437

[12] For this variety of this experiment; minimum times are subject to the variable time in send. A repetition of the experiment might discover a smaller or a larger minimum time.

ms. Since the long event biased the standard deviation so much, this standard deviation is not unusual.

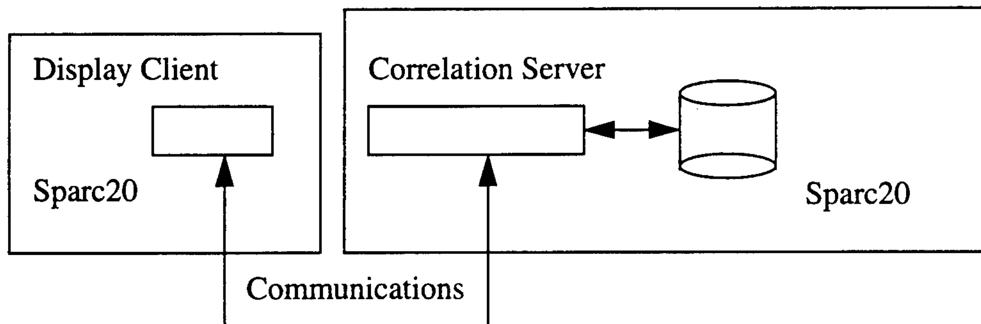


Figure 8. Client and Server on Different Sparc20s

In the case where the real time clock was noted at all nine points, the average time required for the Server + Transport was 1.44 ms which could be determined by the timings on the Client; the average time required for the Client + Transport was 1.60 ms which could be determined by the timings on the Server. In this case the total roundtrip time was an average 1.721 ms. Thus the Client can be inferred to have taken 0.281 ms and the Server 0.121 ms, so Transport must have taken approximately 1.319 ms for the sum of the transfer from Client to Server and that from the Server to the Client. Unix *time* revealed that the Server was busy 1.6 ms per invocation, but this time included amortization of the 50,000 line printout at the end of the experiment. A more detailed experiment might have measured this timing, with the printout function nulled out.

3.2.3 Server on SPARC20 and Client on Pentium 90

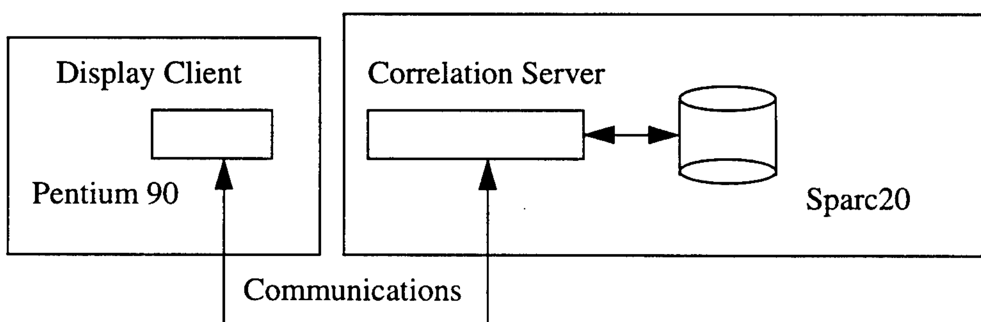


Figure 9. Client and Server on Different Platforms

A histogram of the times required for the complete execution of the iteration of a call from the Client to the Server and return was obtained for the 10,000 invocations of the interface. The minimum roundtrip was 3.63 ms, about 2.2 times that of a Sparc20. (This is likely due to the lower performance of the Pentium 90, but without a high resolution clock, we cannot categorize the reduction in performance precisely.) The maximum roundtrip, 43.56 ms, was substantially better than the pair of Sparc20s. The average roundtrip was 4.06 ms, about 2.46 times that of a Sparc20. The standard deviation was 0.554 ms. Client + Transport took an average time 3.93 ms, leaving the Server at 0.13 ms, very close to what we might expect. The Server was busy about 1.557 ms including writing the output file, which is close to the time required when two SPARC20s interworked.

3.3 938 BYTE STRING RETURNED (MAIN EXPERIMENT 2)

The second experiment involved the return of a string of 938 bytes from the Server, a token track. Conveyance of an actual track as a stream of bytes could be expected to work correctly only if the Client and the Server were on the same type of machine since an actual track is a combination of integers, floating point numbers, structures, and strings. Use of a string to represent a structure of elements might not convey a track record correctly, if one machine was a Sparc with a 64 bit Operating System (OS) and the other had a 32 bit OS, because of alignment and representation differences. It would certainly not work correctly if the actual data represented a VtPlatform track[13] and the Client was a Pentium machine since Sun and Pentium use different alignments and byte orders. If IDL is used for the actual track, the code generated by the IDL compiler correctly deals with alignment and data representation issues.

```
typedef string<938> VtTrack;
interface Trax {
void print (in long signal);
void get (in long index, out VtTrack track, out double time);
};
```

3.3.1 Client and Server on Same Platform

The minimum roundtrip time was 1.77 ms and the maximum was 19.4 ms. This is about the same amount of dispersion as encountered in the previous experiment. The average roundtrip time was 1.86 ms and the standard deviation was 0.321 ms. From Client to Server (in Attachment 1) took 0.979 ms and from Server to Client was 0.384 ms, resulting in a total communication time of 1.363 ms or 0.03 ms more than the transmission time for a transmission of 8 bytes.

[13] See Appendix A.3.5, typedef for A27 — a VtPlatform track.

The time for the Client Proxy to unmarshal its arguments was 0.017 ms and for the Client Proxy to marshal the results was 0.05365 ms. This does not represent a serious difficulty when compared to the communication time above.

3.3.2 Client and Server on Different SPARC20s

When two Sparc20s are used, the minimum roundtrip time increased to 2.629 ms and the maximum to 23.83 ms. The average also increased to 2.684 ms and the standard deviation increased to 0.405 ms. This is likely do to the difficulty of coordinating the two operating systems.

The time for Client + Transport was 2.5 ms and for Server + Transport: 2.391 ms. The latter was measured with a nine-point experiment.

The time for the Client Proxy to unmarshal arguments was 0.015 ms and for the Client Proxy to marshal was 0.053 ms (3.6 • basic), which was as expected.

3.3.3 Server on SPARC20 and Client on Pentium 90

When the Pentium 90 was used as the Client, the minimum roundtrip time increased again to 4.7 ms as did the maximum to 48.59 ms. The average was 5.11 ms and the standard deviation was 0.618 ms.

The Client + Transport time was 4.917 ms average. This is almost twice the time for the two Sparc20s.

Again the Client Proxy unmarshal time was about the same: 0.018 ms as was the Client Proxy marshal time of 0.060 ms.

3.4 VTPLATFORM TRACK RETURNED (MAIN EXPERIMENT 3)

The third experiment was closest to reality: the contents of a VtPlatform track was returned from the Server in addition to the high resolution time. The expected variation in this instance is the marshalling, unmarshalling, and transmission times *because the data is highly structured.*

```
interface Trax {
void print (in long signal);
void get (in long index, out VtPlatformTrack track,
out double time);
};
```

3.4.1 Client and Server on Same Platform

When the Client and Server processes are hosted on the same platform, the minimum roundtrip time increased about 50% over the string results to 2.50 ms, whereas the maximum time remains about the same at 19.6 ms. The average was 2.57 ms; the standard Deviation was 0.322 ms, about the same as before.

The time for transmission from Client to Server was 0.994 ms; from Server to Client it was 0.415 ms, an increase of 14%.

The Client Proxy unmarshal time increased sharply to 0.339 ms (20 times the case of the string) and the Client Proxy marshal time increased to 0.248 ms (almost 5 times the time for the string). It was not clear why the unmarshalling time increased greatly. A hypothesis is that the unmarshalling stage allocates storage for the returned result. In the case of the VtPlatform track, this is a very expensive operation because so many parts of the result must be allocated.

The total time used by the Server process is 2.22 ms including writing all Server timings to the file.

3.4.2 Client and Server on Different SPARC20s

In the case when two different Sparc20s are used, the minimum roundtrip time is 3.6 ms and the maximum time is 19.03 ms. The average time was 3.67 ms and the standard deviation was 0.405 ms.

The Client + Transport time was 2.979 ms; the Server + Transport time was 3.187 ms (measured with 9 measuring points).

Client Proxy unmarshal time required 0.334 ms (about 22 times the time required for the string); Client Proxy marshal time required 0.273 ms (about 5 times the time for the string).

3.4.3 Server on SPARC20 and Client on Pentium 90

When the Client was transported to the Pentium, the minimum roundtrip time was 6.20 ms, about 1/3 higher than with the string. The maximum roundtrip time was 45.21 ms. The average was 6.65 ms and the standard deviation was 0.585 ms. The distribution of values is very high in the general region of the mean and drops off quite sharply; however, a number of the 10,000 values are quite disperse from the mean, yielding the large standard deviation. The distribution of results looks more like a Raleigh distribution than a Gaussian distribution. See the graph of distribution, Appendix B, Attachment 2.

The time for Client + Transport was 5.851 ms.

The time for Client Proxy unmarshal was 0.419 ms and for Client Proxy Marshal was 0.273 ms.

3.5 ADDITIONAL INFORMATION

Additional information can be found in the detailed spreadsheet of Appendix B, Attachment 1.

CHAPTER 4. SUMMARY OF OBSERVATIONS

The set of experiments have led to some summary observations. In the following we will simplify those results in order to aid in "back-of-the-envelope" analyses by developers. (*Caution: your results may vary widely from ours because of equipment, software, applications, or use.*) On a Sparc20, a local procedure call to the timer with a minimal returned result costs about 1.5 microseconds (μs). A minimal Orbix RPC to the timer with a minimal returned result costs about 1.5 milliseconds (after removing timing and looping) or 1000 times longer[14]. Communications costs about 1.3 ms of that RPC time. In order to have 50% efficiency using RPC, the work done by the Server would have to be > 1000 LPCs or 1.5 ms; for a 90% efficiency, more than 9000 LPCs or 13.5 ms would be required. Thus for high efficiency the work performed on the Server should be very much greater than 1 ms.

Substantial time is used by communications: 1.3 ms with very small messages in each direction on the Sparc20 in loopback mode. More time is used if a physical network is used. If a 9600 baud network were used, and the associated 186 bytes of protocol and message were transmitted in the case of the high resolution timing call, assuming 10 bits per character, an additional time of nearly 200 ms would be required for transmission time[15]. Use of an ethernet on the other hand would only require an additional 149 μs for transmission.

Substantial additional time is required for handling heterogeneity, e.g., using Orbix 2.0.1 for marshaling strings (0.13 ms) or structures (0.95 ms). Time required is a strong function of the complexity of the returned quantity, due to allocation and copying. It is likely that different IDL compilers will handle marshalling and unmarshalling differently; hand-optimized code for marshalling and unmarshalling may be required in order to achieve usable results, particularly in the case of very long or very complex arguments.

Even a Sparc20 cannot handle very many object requests per second. Suppose we assume that half of the time of Client-to-Server and Server-to-Client is used by the Server. This

---

[14] Even if a different RPC mechanism were found which offered little overhead in its RPC, it is unlikely to be faster than 250 times a LPC. See Attachment 2 in Appendix B.

[15] There might be an additional latency proportional to the distance travelled.

would yield a total of 0.65 ms in the case of returning high precision time. Client Proxy unmarshal requires 0.02 ms. The Server uses 0.08 ms. Client Proxy marshalling uses another 0.03 ms. This totals 0.78 ms. Thus an *upper bound* on the number of requests that the Sparc20 could handle is approximately 1280 per second without performing any other activity. *The actual maximum will be substantially lower.*

Returning a 938 byte payload, we know that the Server + Transport requires 2.391 ms (from measurements on the Client side of two Sparc20 platforms). Subtracting out the Server times by using measurements on the Server, of 0.689 ms, we get 1.7 ms. If we attribute half to the Client Platform and half to the Server Platform, we find the transmission time attributable to the Server is 0.85 ms. Combined with the Server usage, this results in 1.54 ms. Thus only 649 requests per second could be serviced by a single Sparc20. Further, no additional operations could be handled by the Sparc20.

Use of the information acquired

Suppose we wished to divide the GCCS Track Correlation Application into a display client and a correlation server which accesses a track database using the current track database API. Assume that track updates occur at a rate of 1000 per second, that tracks are stored in a memory cache, and that information is disseminated using RPC based on the API above using a single request to return a single record.

Based on the current experiments are the following tentative conclusions: The simple division of the track correlation application into display client and track correlation database with our simple use of request-reply semantics[16] could not meet the requirement of 1000 queries per second on the measured equipment with the version of software used. It remains to be seen whether a different processor or a different strategy such as sequential transmission of multiple tracks[17] would better utilize the processor so that the server work could be done in the time required. Design strategies must also be reconsidered if a higher performance processor, a different or improved ORB/IDL/Library combination, or a better implementation of TCP/IP is used.

[16] The semantics of local and remote procedure call.

[17] In a push mode where the service sends all relevant data at once instead of one track at a time.

CHAPTER 5. RECOMMENDATIONS FOR DESIGN

Based upon our experiments, we recommend that designers or developers should carefully consider the timing properties of the *delivery platform*(s) and of the software during the design of *any distributed* application:

- Developers should create a *simple prototype* from which they will gain the information required for their design. For the targeted client and server platform, designers should employ the hardware and software which will support the delivered application. Designers should determine the time to marshal[18] and unmarshal the data structures to be used. In addition they should measure the amount of communications time taken by each different kind of remote procedure call and the amount of time required to service each request. This should lead to a mathematical model of the amount of resources required to perform each kind of request. These times can also be used in simulations of the applications to be designed.
- Designers or developers should give special attention to the following factors:
  - Carefully consider the amount of computation to be done on the server for each RPC. Too little computation on the server reduces the overall efficiency of computation because of the communications overhead. Perhaps, move inefficient computations to clients.
  - Carefully consider the type of argument(s) to be passed to the server or returned to the client. Try to make the argument as simple and as aligned as possible. Attempt to find an implementation of IDL that can perform marshalling, unmarshalling, and service in parallel, using threads. This is particularly valuable if the service performs I/O operations.
  - If the task can be parallelized, use multiple servers and distribute the computation (servers) on multiple platforms. Also try to hide communications latencies by using multi-threaded servers to handle multiple requests in a pipelined fash-

[18] See the body of the paper or glossary for definition of technical terms such as marshalling and unmarshalling.

ion. Find the sum of times on the paths which must be performed serially. This is the optimum time which can be achieved [one of Amdahl's laws]. You may think that using many machines executing a task in parallel pieces will reduce the total computation time. But you must account for the overhead introduced by the communication. Verify that the time for using multiple servers is shorter than running the calculation on a single machine using LPC.

- Carefully consider the factors which could render your experimental results invalid such as communications to other servers on the same host. Try to determine if they will contribute to your client-server workload. Especially consider the number of requests per second to database servers and the amount of I/O operations these servers are expected to perform.

- Carefully consider the confounding factors. Try to characterize them in order to determine if they will contribute to your client/server workload. If they will, identify mitigation strategies. Especially consider the number of requests per second to database servers and the amount of I/O these servers are expected to do.

As has been suggested in the illustrative examples, results vary depending on many factors. What is important is to experiment and model during design to determine what factors are most significant and what can be done to improve performance.

5.1 BUILD A PROTOTYPE TESTING CAPACITY

In the same way that this task developed a prototype which could perform a check on the capacity of the system, the developer should build and measure a "simple prototype" which will gain the information required for his/her design. *For the Client and Server platform involved, the designers should employ the hardware and software which will support the delivered application in their prototype.*

As mentioned in bullet 1 of section 5.1 above, for each Client request, determine the following (based on Figure 10):

a. Time for Server Proxy Marshal, $T1=\tau_2-\tau_1$

b. Time for Transmission of Client packets and number of packets and bytes transmitted by Server Proxy, $T2=\tau_3-\tau_2$

c. Time for Client Proxy Unmarshal, $T3=\tau_4-\tau_3$

d. Time for Service Action, $T4=\tau_5-\tau_4$

e. Time for Client Proxy Marshal, $T5=\tau_6-\tau_5$

f. Time for transmission of the Server packets and number of packets and bytes transmitted by the Client Proxy, $T6=\tau_7-\tau_6$

g. Time for Server Proxy Unmarshal, $T7=\tau_8-\tau_7$

h. Time for Client Action, $T8=\tau_1-\tau_8$ (when looping)

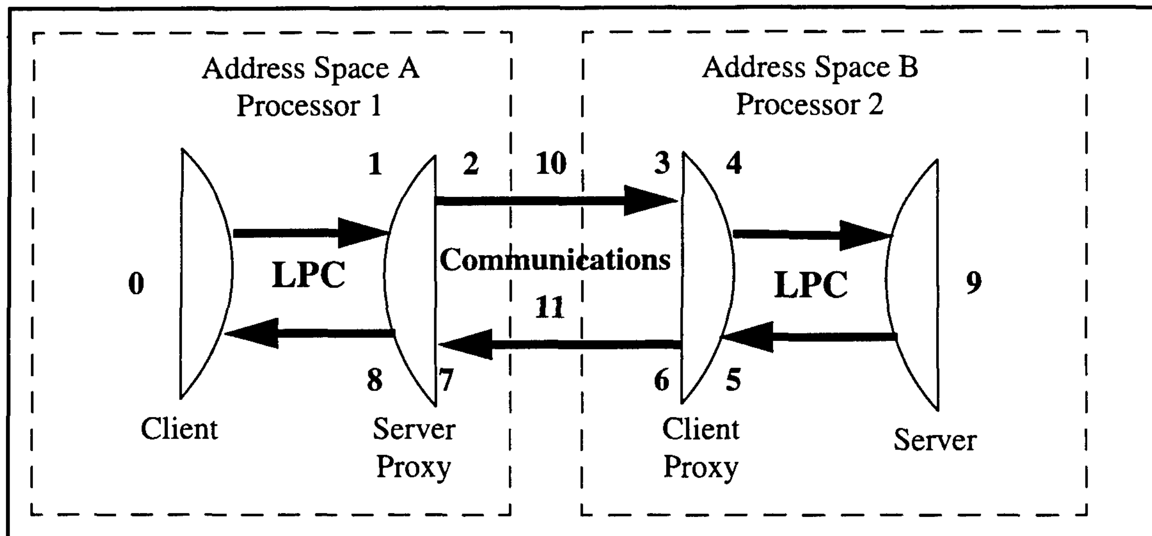


Figure 10. Experimental Monitoring Points

Actual values for the "T"s for our experiments are calculated in Attachment 1. Your values will likely differ from these values, depending on your hardware, network, and software. Inserting the timing probes can be difficult on a PC running Microsoft NT or using a different ORB[19]. If the platform is the problem, use the same ORB on a platform supporting a high resolution clock and attempt to scale the result to account for platform speed and data transport. Most ORBs support interceptors; if yours does not, you may have to try to instrument the operating system in order to get the information you wish or use binary debugging tools to insert a call to the measurement apparatus.

By determining how many bytes B1 and packets, by using *netstat* or equivalent, are transmitted by the Server Proxy at 10, estimate how many packets are required to convey the

[19] For example, PowerBroker by Expersoft only provides a per-process interceptor at points 1,3,5, and 7, making the profiling performed in Orbix extremely difficult to perform.

Client's request to the Server (See attachment 1 for our values). By determining how many bytes B2 are transmitted by the Client Proxy, at 11, estimate how many packets are required to convey the Server's reply to the Client. Determine how much real-time is used by the transmission of the packets assuming optimum transmission (B1+B2)/(BandWidth) (50% of bandwidth on WAN and LAN-ethernet; 80% of bandwidth on FDDI or token ring can be used). Determine how much real-time is used by the transmission of the packets, (T2+T6). Sum the real-time required of the communications device by the Client request and the Server response and determine how many request/response pairs could be sent per second. Sum the number of packets required for each request/response pair and divide this into the maximum number of packets that the Client/Server can handle (whichever is smaller). The number of actual request response pairs determined for communication is the lesser of that determined by the packet method or the bandwidth method.

Attempt to estimate the time which is required by the Server for handling communication to and from the Server: (T2+T6)/2. Add this to T3, T4, and T5 to determine the amount of the time used by the Server's platform in order to perform the request. Estimate the platform's capacity by determining the maximum number of requests that the Server could handle if it only performed this request repeatedly.[20]

Attempt to estimate the time which is required by the Client for handling communication to and from the Client. Add this to times T1, T6, and T7 to determine the amount of the time used by the Client's platform in order to send the request and process its results. Estimate the platform's capacity by determining how many requests the platform can issue per second if it performs no client work on the results it obtains.

The maximum number of requests per second is the least of the number obtained by the communications method, the Server method, and the Client method as described above. This number is likely an optimistic number because it does not take into account the "confounding factors" in Section 2.3.

After having determined these numbers for each kind of request to be issued, estimate the number of each kind of request to be issued per second. Using the Server, Client, and communications times for each kind of request, calculate the total real-time to be used per second for the series of requests. If this number exceeds the maximum computational power/communication bandwidth allocated to this type of request response pair per second, another solution

---

[20] Caution, this is only valid if the server does not idle during the performance of this activity; if it does, another activity might be able to utilize this time.

must be devised. If it does not, determine whether there is a great deal of slack in your numbers, in case your estimates were not accurate. If there is not a great deal of slack, analyze or simulate the application in greater detail. If so, continue implementation remaining aware that the confounding factors may render your analysis invalid.

5.2 FINAL COMMENTS

To determine the best design for a distributed application, you must have or obtain an understanding of the parameters that affect your choices of how to divide your application for execution on distributed platforms. It may be necessary to iterate prototyping and calculation in order to achieve confidence that you understand the characteristics of your suite of applications, your production Servers, and your production Clients. The more types of requests, the more varied your applications, and the more diverse your collection of platforms, the greater the work will be to understand the interactions of all the components.

LIST OF ACRONYMS

API	Application Programming Interface
ATM	Asynchronous Transfer Mode
COE	Common Operating Environment
CORBA	Common Object Request Broker Architecture
C-S	Client Server
DARPA	Defense Advamced Research Projects Agency
DCE	Distributed Computing Environment
DCOM	Distributed Component Object Model
DCWG	Distributed Computing Working Group
DII	Defense Information Infrastructure
DISA	Defense Information Systems Agency
GCCS	Global Command and Control System
IDL	Interface Definition Language
LAN	Local Area Network
LPC	Local Procedure Call
ms	Milliseconds
ns	Nanoseconds
OG	Open Group
OMG	Object Management Group
OO	Object-Oriented
ORB	Object Request Broker
OSF	Open Software Foundation
RPC	Remote Procedure Call
TCA	Track Correlation
TCP/IP	Transmission Control Protocol/Internet Protocol
TCS	Track Correlation Service
WAN	Wide Area Network
μs	Microseconds

APPENDIX A. GLOSSARY

Interface. A specification of the external properties of an object including attributes and methods of the object.Attributes and parameters of methods indicate their type and whether they are input, output, or inout attributes/parameters. Methods provide a "signature" which specifies returned type, object name, method name, and a list of parameter types.

Marshalling. The process of assembling arguments to a remote procedure call into a canonical data structure which can be transmitted to a remote machine for use.

Method. A procedure provided by an object for the manipulation of that object's internal state, which may invoke other methods internal to or external to the object.

Microsecond. One one-millionth (1/1,000,000) of a second.

Millisecond. One one-thousandth (1/1,000) of a second.

Nanosecond. One one-billionth (1/1,000,000,000) of a second.

Track. A description of location as a function of time; may result from data obtained from optical, electrical, acoustic, thermal, or visual sensing as correlated by the track correlation process and as stored in a track record in a Track Data Base (See Section C.3.5 for an interface definition).

Unmarshalling. The process of converting a data structure provided by a remote machine into a list of arguments to be used by a procedure call on the local machine.

APPENDIX B. ATTACHMENTS

Attachment 1
Results of Three Experiments for Four Variations

Experiment	Roundtrip Average	Roundtrip Maximum	Roundtrip Minimum	Roundtrip STDEV	Server Proxy Marshal
5 Point DP Time - S20	1,692,588	25,313,000	1,618,000	368,565	
5 Point DP Time - S20/S20	1,649,965	134,693,500	1,588,500	1,437,322	
5 Point DP Time - S20/p90NT	4,063,088	43,563,500	3,636,500	554,332	
9 Point DP Time - S20	1,746,266	17,351,000	1,690,500	279,260	20,418
9 Point DP Time - S20/S20	1,721,986	18,246,500	1,678,000	337,201	22,335
9 Point DP Time - S20/IPX	3,128,539	58,184,000	2,991,000	970,288	58,378
Byte Track - S20	1,855,274	19,478,500	1,772,500	320,600	16,032
Byte Track - S20/S20	2,684,241	23,830,500	2,629,000	429,759	17,975
Byte Track - S20/IPX	4,452,031	65,779,500	4,271,500	1,041,322	55,878
Byte Track - S20/p90NT	5,108,025	48,589,500	4,699,500	618,196	
Simple Track - S20	2,579,790	19,673,500	2,501,500	322,206	17,544
Simple Track - S20/S20	3,668,581	19,003,500	3,600,000	405,088	16,390
Simple Track - S20/IPX	6,040,525	57,072,500	5,807,000	1,346,110	55,978
Simple Track - S20/p90NT	6,647,939	45,207,000	6,197,000	585,265	

All Units Are ns Unless Noted

Attachment 1
Results of Three Experiments for Four Variations

Experiment	Server+Transport	Client->Server	Client Proxy Unmarshal	Server Phase A	Server Phase B
5 Point DP Time - S20			17,973	4,895	84,229
5 Point DP Time - S20/S20			16,032	4,757	73,864
5 Point DP Time - S20/p90NT			26,075	6,068	78,152
9 Point DP Time - S20		939,141	20,514	5,061	74,760
9 Point DP Time - S20/S20	1,440,298		21,060	6,880	75,041
9 Point DP Time - S20/IPX	2,286,029		21,445	6,859	85,101
Byte Track - S20		978,768	17,997	37,502	79,957
Byte Track - S20/S20	2,391,921		15,285	36,377	79,315
Byte Track - S20/IPX	3,450,697		17,707	46,347	85,874
Byte Track - S20/p90NT			18,114	37,377	75,753
Simple Track - S20		994,449	338,775	5,565	85,977
Simple Track - S20/S20	3,187,199		334,217	8,915	84,725
Simple Track - S20/IPX	4,203,181		326,114	4,989	81,905
Simple Track - S20/p90NT			419,228	5,132	99,286

All Units Are ns Unless Noted

Attachment 1
Results of Three Experiments for Four Variations

Experiment	Client Proxy Marshal	Total Server	Client + Transport	Server->Client	Server Proxy Unmarshal
5 Point DP Time - S20	14,844	121,941	1,570,711		
5 Point DP Time - S20/S20	13,985	108,638	1,541,389		
5 Point DP Time - S20/p90NT	19,475	129,770	3,933,383		
9 Point DP Time - S20	29,809	130,144		363,876	62,453
9 Point DP Time - S20/S20	14,884	117,865	1,604,184		18,643
9 Point DP Time - S20/IPX	15,458	128,863	2,999,741		78,010
Byte Track - S20	53,655	189,111		384,206	59,506
Byte Track - S20/S20	53,441	184,418	2,500,186		57,897
Byte Track - S20/IPX	55,012	204,940	4,247,137		255,931
Byte Track - S20/p90NT	59,785	191,029	4,917,044		
Simple Track - S20	248,011	678,328		415,327	244,263
Simple Track - S20/S20	260,785	688,642	2,979,992		240,776
Simple Track - S20/IPX	272,930	685,938	5,354,640		1,043,796
Simple Track - S20/p90NT	273,155	796,801	5,851,294		

All Units Are ns Unless Noted

Attachment 1
Results of Three Experiments for Four Variations

Experiment	Client Processing	Real Time Server	User Time Server	System Time Server	Total Server	Per Invoc. (ms)	Real Time Client
5 Point DP Time - S20		87.26s	11.26s	4.19s	15.45s	1.545	23.86s
5 Point DP Time - S20/S20		86.17s	11.57s	4.29s	16.26s	1.626	23.45s
5 Point DP Time - S20/p90NT		122.98s	10.64s	4.63s	15.57s	1.557	?
9 Point DP Time - S20	230,292	75.57s	12.24s	3.45s	16.09s	1.609	30.15s
9 Point DP Time - S20/S20	243,208	94.88s	9.94s	4.3 s	14.24s	1.424	30.21s
9 Point DP Time - S20/IPX	707,558	128.4s	11.15s	4.00s	15.15s	1.515	60.74s
Byte Track - S20	232,589	32.87s	12.13s	4.08s	16.21s	1.621	29.26s
Byte Track - S20/S20	218,672	42s	11.35s	4.73s	16.48s	1.648	35. s
Byte Track - S20/IPX	738,784	74.68s	12s	5.12s	17.12s	1.712	70.3s
Byte Track - S20/p90NT		72.59s	10.54s	5.31s	16.25s	1.625	?
Simple Track - S20	234,850	50.48s	17.38s	4.42s	22.20s	2.220	37. s
Simple Track - S20/S20	224,449	51.31s	16.78s	4.39s	21.57s	2.157	49.52s
Simple Track - S20/IPX	738,928	92.71s	16.89s	4.29s	20.58s	2.058	87.57s
Simple Track - S20/p90NT		89.81s	20.63s	5.99s	25.62s	2.562	?

All Units Are ns Unless Noted

Attachment 1
Results of Three Experiments for Four Variations

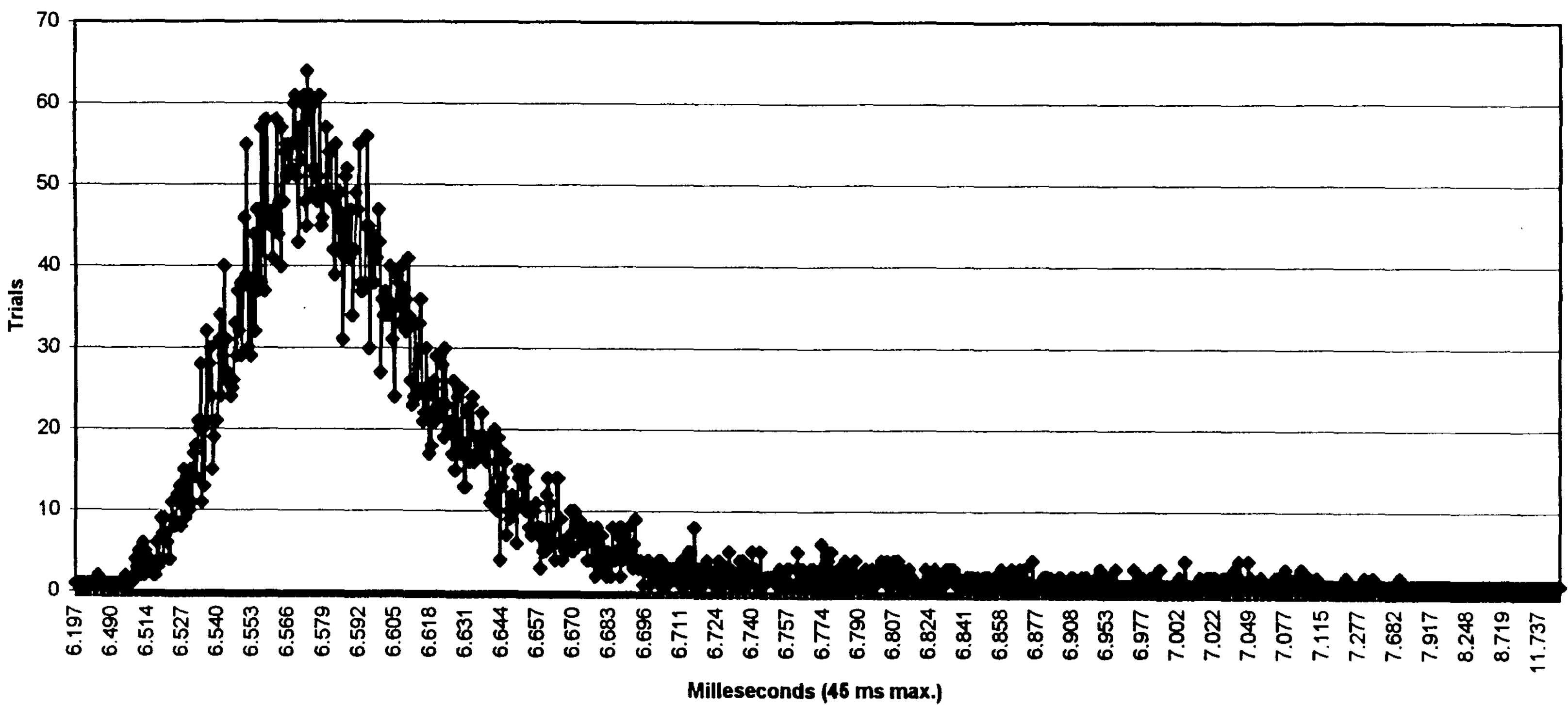
Experiment	User Time Client	System Time Client	Total Client	Per Invoc. (ms)	Total Per Invoc. (ms)	Input Packets	Bytes
5 Point DP Time - S20	5.64s	2.53s	8.17s	0.817	2.362	20,031	1,683,200
5 Point DP Time - S20/S20	3.91s	2.56s	6.47s	0.647	2.273	20,085	644,784
5 Point DP Time - S20/p90NT	?	?				20,031	1,863,784
9 Point DP Time - S20	9.90s	4.38s	14.18s	1.418	3.027	20,045	2,628,166
9 Point DP Time - S20/S20	8.02s	4.7s	12.72s	1.272	2.696	20,084	804,984
9 Point DP Time - S20/IPX	29.21s	14.76s	43.97s	4.397	5.912	21,347	1,754,692
Byte Track - S20	8.71s	4.12s	12.83s	1.283	2.904	20,034	11,795,435
Byte Track - S20/S20	9. s	5.9s	14.9 s	1.490	3.138	21,087	9,975,231
Byte Track - S20/IPX	29.95s	16.89s	46.84s	4.684	6.396	21,142	11,000,875
Byte Track - S20/p90NT	?	?			1.625	20,032	11,225,280
Simple Track - S20	10.88s	4.15s	15.03s	1.503	3.723	20,034	14,818,894
Simple Track - S20/S20	11.53s	6.04	17.57s	1.757	3.914	20,094	13,045,462
Simple Track - S20/IPX	37. s	17.4s	54.4s	5.440	7.498	21,451	14,258,786
Simple Track - S20/p90NT	?	?			2.562	20,033	14,985,764

All Units Are ns Unless Noted

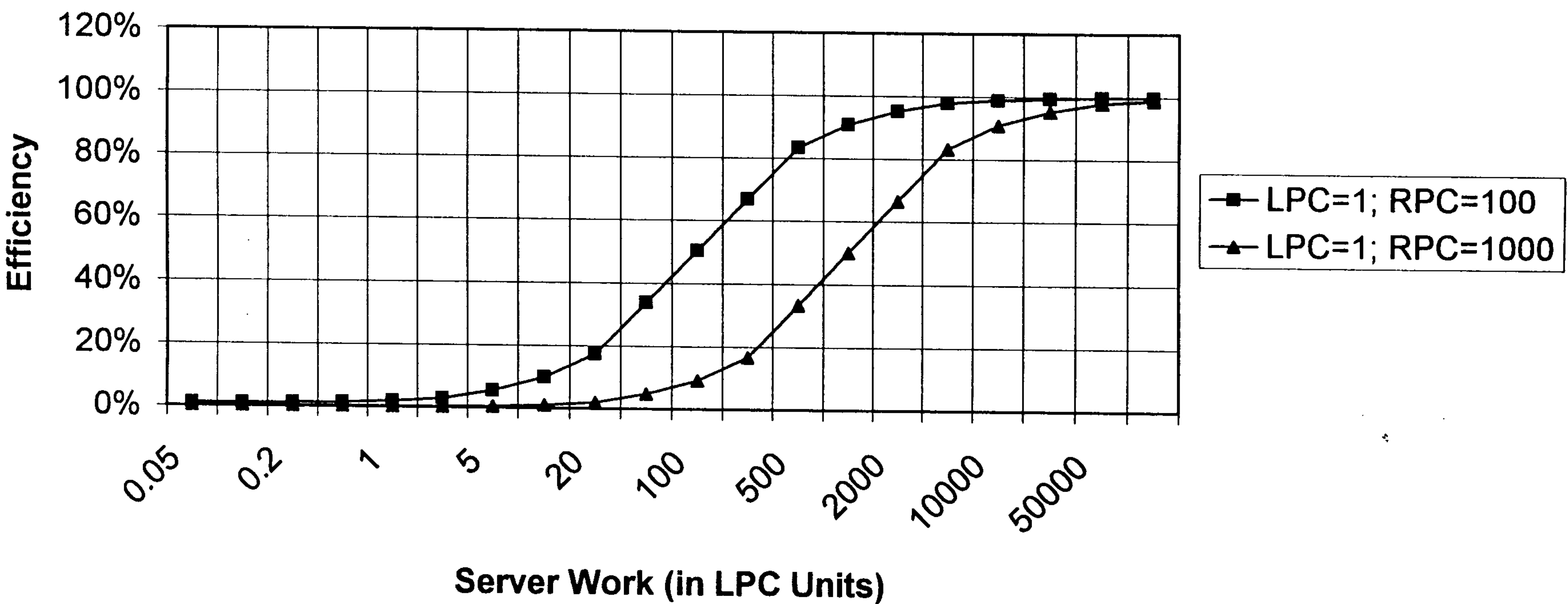
Attachment 1
Results of Three Experiments for Four Variations

Experiment	ipInDelivers	msgs	Notes
5 Point DP Time - S20	20,438	42,289	
5 Point DP Time - S20/S20	20,085	30,462	
5 Point DP Time - S20/p90NT	20,033	33,563	msgs on s20
9 Point DP Time - S20	20,972	45,079	
9 Point DP Time - S20/S20	20,084	30,454	
9 Point DP Time - S20/IPX	21,327	32,491	
Byte Track - S20	22,087	43,783	
Byte Track - S20/S20	20,087	30,516	
Byte Track - S20/IPX	21,132	32,257	
Byte Track - S20/p90NT	20,036	32,792	msgs on s20
Simple Track - S20	20,712	43,785	
Simple Track - S20/S20	20,091	32,029	
Simple Track - S20/IPX	21,431	32,644	
Simple Track - S20/p90NT	20,037	33,353	msgs on s20

All Units Are ns Unless Noted



Attachment 2
Distribution of Round Trip Time in MS for Pentium Client in 10,000 Trials
Trials
70
60
50
40
30
20
10
0
6.197
6.490
6.514
6.527
6.540
6.553
6.566
6.579
6.592
6.605
6.618
6.631
6.644
6.657
6.670
6.683
6.696
6.711
6.724
6.740
6.757
6.774
6.790
6.807
6.824
6.841
6.858
6.877
6.908
6.953
6.977
7.002
7.022
7.049
7.077
7.115
7.277
7.682
7.917
8.248
8.719
11.737
Milleseconds (45 ms max.)
number



Attachment 3
Efficiency: (LPC+Server Work)/(RPC+Server Work)
120%
100%
80%
60%
40%
20%
0%
Efficiency
0.05
0.2
1
5
20
100
500
2000
10000
50000
Server Work (in LPC Units)
LPC=1; RPC=100
LPC=1; RPC=1000

APPENDIX C. SOURCE CODE FOR TESTS

C.1 DOUBLE PRECISION TIMER SOURCE ROUTINES

C.1.1 CLIENT

```
// Client.C

// allow the defintion of the _individual_ system exceptions to be visible.
#define EXCEPTIONS

#include "filter.hh"
#include <stream.h>

// for extern "exit"
#include <stdlib.h>

// for timing
#include <sys/time.h>


#include "ProcFilt.h"

ProcessFilter PFilter;// create one instance of the per-process filter.
               // This will monitor calls to and from this
               // address space.

int main (int argc, char **argv) {
   unsigned long value = 0;
   double temp;
   inc_var bVar;
   hrtime_t start_hires, end_hires, diff_hires, junk, Svr_time[15000];
   int maxS;

   // need a hostname argument:
   if (argc < 2) {
            cerr << "usage:" << argv[0] << " <hostname>" << endl;
            exit(1);
    }

   // now try to bind to the target object.
   //see how long a bind takes
   try {
        cout << "TTTD: Attempting Bind..." << endl;
        start_hires = gethrtime();
        bVar = time::_bind (":xfilter",argv[1]);
        end_hires = gethrtime();
        diff_hires = end_hires - start_hires;
```

```
        cout << "TTTR: Bind time:  " << diff_hires << " ns" << endl;
        cout << "Start value: " << value << "\n";
    }
    catch (CORBA::SystemException &sysEx) {
        cerr << "Unexpected system exception" << endl;
        cerr << &sysEx;
        cerr << "Bind failed" << endl;
        exit(1);
    } catch (...) {
        // the bind failed
        cerr << "Unexpected exception : bind failed" << endl;
        exit(1);
    }


    for (int i = 0; i < 10000; i++) {
        try {
          temp = bVar->hr_time (value);
          Svr_time[i] = temp;
        } catch (...) {
            // should get no exceptions:
            cout << "Exception\n";
            cout << "call to get time failed" << endl;
            exit(1);
        }

    }

    for(i=0; i<orqprmI-1 ; i++)
        { cout << orqprmA[i] << "->C1" << endl; };
        orqprmI = 0;
    for(i=0; i<orqpomI-1 ; i++)
        { cout << orqpomA[i] << "->C2" << endl; };
        orqpomI = 0;
    for(i=0; i<irpprmI-1 ; i++)
        { cout << irpprmA[i] << "->C7" << endl; };
        irpprmI = 0;
    for(i=0; i<irppomI-1 ; i++)
        { cout << irppomA[i] << "->C8" << endl; };
        irppomI = 0;
    for(i=0; i<10000; i++)
        { cout << Svr_time[i] << "->S9" << endl; };
    value = 16000001;
    value = bVar->increment(value); //Print out table
    value = 1;

    return 0;
}
```

C.1.2 SERVER

```
// make the defintions of the _individual_ system exceptions to be visiable.
#define EXCEPTIONS
#include <stream.h>
#include "filter_i.hh"
```

```
#include <sys/time.h>    // for timing

#include "ProcFilt.h"

// for extern "exit"
#include <stdlib.h>

ProcessFilter Filter_instance;    // install per-process filters.
void printtimes(void) {
        int i;
        for(i=0; i<irqprmI; i++)
         cout << irqprmA[i] << "->S3" << endl;
        irqprmI = 0;
        for(i=0; i<irqpomI; i++)
        cout << irqpomA[i] << "->S4" << endl;
        irqpomI =0;
        for(i=0; i<orpprmI; i++)
         cout << orpprmA[i] << "->S5" << endl;
        orpprmI = 0;
        for(i=0; i<orppomI; i++)
                cout << orppomA[i] << "->S6" << endl;
        orppomI =0;
};

int main(int , char *argv[]){
       hrtime_t start_hires, end_hires, diff_hires, junk;  // TTTTT

       // create the target object

        time_i mytime;

        // Export the server to the network
        // tell Orbix that the server has completed its initialisation:
        cout << "TTTD: Attempting impl_is_ready..." << endl;
                start_hires = gethrtime();
        CORBA::Orbix.impl_is_ready("xfilter");
                end_hires = gethrtime();
                diff_hires = end_hires - start_hires;
                cout << "TTTR: impl_is_ready time:  " << diff_hires/1000000 << " ms" << endl;

        } catch (CORBA::SystemException &sysEx) {
        cerr << "Unexpected system exception" << endl;
        cerr << &sysEx;
        cerr << "Unexpected exception : impl_is_ready" << endl;
        exit(1);
        } catch (...) {
        // got an exception from impl_is_ready()
               err << "Unexpected exception : impl_is_ready" << endl;
               exit(0);
        }

        cout << "\n\nFilter Server shutdown." << endl;

        return 0;
}
```

C.1.3 FILTERS

```
#define EXCEPTIONS
#include <CORBA.h>
#include <stream.h>
#include <sys/time.h>
static hrtime_t orqprmA[15000];
static int orqprmI=0;
static hrtime_t orqpomA[15000];
static int orqpomI=0;
static hrtime_t irqprmA[15000];
static int irqprmI=0;
static hrtime_t irqpomA[15000];
static int irqpomI=0;
static hrtime_t orpprmA[15000];
static int orpprmI=0;
static hrtime_t orppomA[15000];
static int orppomI=0;
static hrtime_t irpprmA[15000];
static int irpprmI=0;
static hrtime_t irppomA[15000];
static int irppomI=0;

// class ProcessFilter is a per-process filter which just outputs messages as
// it sees requests.
class ProcessFilter : public CORBA::Filter {

public:
// REQUESTS

// an out going call, before it has been sent out of the addr space:

    CORBA::Boolean outRequestPreMarshal (CORBA::Request& r,
                                          CORBA::Environment&) {
    hrtime_t orqprm;
            orqprm = gethrtime();
            orqprmA[orqprmI] = orqprm;
            orqprmI += 1;
            return 1;
        }

// an out going call, before it has been sent out of the addr space:
    CORBA::Boolean outRequestPostMarshal (CORBA::Request& r,
                                           CORBA::Environment&) {
    hrtime_t orqpom;
            orqpom = gethrtime();
            orqpomA[orqpomI] = orqpom;
            orqpomI += 1;

            return 1;
        }

// an incoming call, before it is sent to the target object
    int inRequestPreMarshal (CORBA::Request& r,
                                           CORBA::Environment&) {
    hrtime_t irqprm;
            irqprm = gethrtime();
```

```
            irqprmA[irqprmI] = irqprm;
            irqprmI += 1;
            return 1;
        }

// an incoming call, before it is sent to the target object
     CORBA::Boolean inRequestPostMarshal (CORBA::Request& r,
                                          CORBA::Environment&) {
     hrtime_t irqpom;
            irqpom = gethrtime();
            irqpomA[irqpomI] = irqpom;
            irqpomI += 1;

            return 1;

        }

// REPLIES

// an out going call, before it has been sent out of the addr space:
    CORBA::Boolean outReplyPreMarshal (CORBA::Request& r,
                                       CORBA::Environment&) {
    hrtime_t orpprm;
            orpprm = gethrtime();
            orpprmA[orpprmI] = orpprm;
            orpprmI += 1;

            return 1;
        }
// an out going call, before it has been sent out of the addr space:
    CORBA::Boolean outReplyPostMarshal (CORBA::Request& r,
                                        CORBA::Environment&) {
    hrtime_t orppom;
            orppom = gethrtime();
            orppomA[orppomI] = orppom;
            orppomI += 1;

            return 1;
            }

// an incoming call, before it is sent to the target object
     CORBA::Boolean inReplyPreMarshal (CORBA::Request& r,
                                       CORBA::Environment&) {
     hrtime_t irpprm;
            irpprm = gethrtime();
            irpprmA[irpprmI] = irpprm;
            irpprmI += 1;

            return 1;
            }


// an incoming call, before it has been sent to the target object
    CORBA::Boolean inReplyPostMarshal (CORBA::Request& r,
                                       CORBA::Environment&) {
    hrtime_t irppom;
            irppom = gethrtime();
```

```
        irppomA[irppomI] = irppom;
        irppomI += 1;

        return 1;
        }


};
```

C.1.4 OBJECT INCLUDES

```
#ifndef filter_ih
#define filter_ih

#include "filter.hh"

class time_i: public virtual incBOAImpl {
public:
        time_i();           // ctor
        virtual ~time_i(); // dtor
            virtual CORBA::Double hr_time (CORBA::ULong vin, CORBA::Environment &IT_env=COR-
        BA::default_environment) ;
};


#endif
```

C.1.5 OBJECT METHODS

```
#include "filter_i.hh"
#include <stream.h>
#include <sys/types.h>
#include <sys/time.h>
void printtimes(void);

time_i::time_i() {};
time_i::~time_i() {};
CORBA::Double time_i:: hr_time (CORBA::ULong vin, CORBA::Environment &IT_env) {
if (vin != 16000001)
return gethrtime();
else printtimes();

}
```

C.1.6 INTERFACE DEFINITION

```
// a simple IDL interface: time. Objects of this interface provide an
// operation `hr_time' which takes an unsigned long value which may signal
// termination of the server and returns the double real-time in nanoseconds
// (if vin does not signal termination of the server).

interface time {double hr_time (in unsigned long vin);};
```

C.2 CHARACTER SEQUENCE SOURCE ROUTINES

C.2.1 CLIENT

```
// Client.C

// This demonstrate the use of both per-process and per-object filtering.
// Classes ProcessFilter and preProces demonstrate per-object filtering.
// Classes fiterPre and filterPost demonstrate per-object filtering.

// allow the defintion of the _individual_ system exceptions to be visible.
#define EXCEPTIONS

#include "Trax2.hh"
#include <stream.h>

// for extern "exit"
#include <stdlib.h>

// for timing
#include <sys/time.h>


#include "ProcFilt.h"

ProcessFilter PFilter;// create one instance of the per-process filter.
                      // This will monitor calls to and from this
                      // address space.

int main (int argc, char **argv) {
   long value = 0;
   hrtime_t start_hires, end_hires, diff_hires, Svr_time[15000];
   double time;
   Trax_var bVar;
   int maxS;

   // need a hostname argument:
   if (argc < 2) {
           cerr << "usage:" << argv[0] << " <hostname>" << endl;
           exit(1);
    }

   // now try to bind to the target object.
   //see how long a bind takes
   try {
        cout << "TTTD: Attempting Bind..." << endl;
        start_hires = gethrtime();
        bVar = Trax::_bind (":sttracks",argv[1]);
        end_hires = gethrtime();
        diff_hires = end_hires - start_hires;
        cout << "TTTR: Bind time:  " << diff_hires << " ns" << endl;
        cout << "Start value: " << value << "\n";
   }
    catch (CORBA::SystemException &sysEx) {
        cerr << "Unexpected system exception" << endl;
        cerr << &sysEx;
```

```
        cerr << "Bind failed" << endl;
        exit(1);
    } catch (...) {
        // the bind failed
        cerr << "Unexpected exception : bind failed" << endl;
        exit(1);
    }

    time = 0;
    char* str;
    for (int i = 0; i < 10000; i++) {
        try {
          str=bVar->get (value, time);
          Svr_time[i] = time;
          CORBA::string_free(str);
        }
        catch (CORBA::SystemException &sysEx) {
        cerr << "Unexpected system exception" << endl;
        cerr << &sysEx;
        cerr << " Why? " << endl;
        exit(1);
        }
        catch (...) {
            // should get no exceptions:
            cout << "Exception\n";
            cout << "call to Trax failed" << endl;
            exit(1);
        }
    }

    for(i=0; i<orqprmI-1 ; i++)
        { cout << orqprmA[i] << "->C1" << endl; };
        orqprmI = 0;
    for(i=0; i<orqpomI-1 ; i++)
        { cout << orqpomA[i] << "->C2" << endl; };
        orqpomI = 0;
    for(i=0; i<irpprmI-1 ; i++)
        { cout << irpprmA[i] << "->C7" << endl; };
        irpprmI = 0;
    for(i=0; i<irppomI-1 ; i++)
        { cout << irppomA[i] << "->C8" << endl; };
        irppomI = 0;
    for(i=0; i<10000; i++)
        { cout << Svr_time[i] << "->S9" << endl; };
    value = 16000001;
    bVar->print(value); //Print out table
    value = 1;

    return 0;
}
```

C.2.2 SERVER

```
// make the defintions of the _individual_ system exceptions to be visiable.
#define EXCEPTIONS
#include <stream.h>
```

```
#include <string.h>
#include "Trax2_i.h"

#include <sys/time.h>   // for timing

#include "ProcFilt.h"

// for extern "exit"
#include <stdlib.h>

ProcessFilter Filter_instance;   // install per-process filters.
void printtimes(void) {
        int i;
        for(i=0; i<irqprmI; i++)
                cout << irqprmA[i] << "->S3" << endl;
        irqprmI = 0;
        for(i=0; i<irqpomI; i++)
                cout << irqpomA[i] << "->S4" << endl;
        irqpomI =0;
        for(i=0; i<orpprmI; i++)
                cout << orpprmA[i] << "->S5" << endl;
        orpprmI = 0;
        for(i=0; i<orppomI; i++)
                cout << orppomA[i] << "->S6" << endl;
        orppomI =0;
};

int main(int , char *argv[]){
       hrtime_t start_hires, end_hires, diff_hires ;  // TTTTT

        // create the string
char *trk = CORBA::string_alloc(938);
        strcpy(trk, "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx"
                "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijk");

       // create the target object

        Trax_i mytrax(trk);

        //
```

```
        // Export the server to the network
        //
        try {

        // tell Orbix that the server has completed its initialisation:
                cout << "TTTD: Attempting impl_is_ready..." << endl;
                start_hires = gethrtime();
                CORBA::Orbix.impl_is_ready("sttracks");
                end_hires = gethrtime();
                diff_hires = end_hires - start_hires;
                cout << "TTTR: impl_is_ready time:  " << diff_hires/1000000 << " ms" << endl;

        } catch (CORBA::SystemException &sysEx) {
                cerr << "Unexpected system exception" << endl;
                cerr << &sysEx;
                cerr << "Unexpected exception : impl_is_ready" << endl;
                exit(1);
        } catch (...) {
                // got an exception from impl_is_ready()
                cerr << "Unexpected exception : impl_is_ready" << endl;
                exit(0);
        }

        cout << "\n\nFilter Server shutdown." << endl;
        CORBA::string_free(trk);
        return 0;
}
```

C.2.3 OBJECT INCLUDES

```
#ifndef Trax2_ih
#define Trax2_ih

#include "Trax2.hh"
#include <string.h>

class Trax_i: public virtual TraxBOAImpl {
      char *m_track;
public:
        Trax_i(char *str);
        ~Trax_i();
                virtual void print (CORBA::Long signal, CORBA::Environment &IT_env=COR-
        BA::default_environment);
            virtual char*  get (CORBA::Long index,  CORBA::Double& time, CORBA::Environment
        &IT_env=CORBA::default_environment) ;
};


#endif
```

C.2.4 OBJECT METHODS

```
#define EXCEPTIONS
#include <CORBA.h>
#include "Trax2_i.h"
#include <sys/times.h>
#include <stdlib.h>
#include <stream.h>
void printtimes(void);

     Trax_i::Trax_i(char *str){
       m_track = CORBA::string_alloc(938);
       strcpy(m_track,str);
        cout << strlen(m_track) << "= track length" << endl;
       };
     Trax_i::~Trax_i(){
       CORBA::string_free(m_track);
       };
void Trax_i:: print (CORBA::Long signal, CORBA::Environment &IT_env) {
printtimes();
exit(1);
};

char* Trax_i:: get (CORBA::Long index, CORBA::Double& time, CORBA::Environment &IT_env) {
int i;
       char* temp = CORBA::string_alloc(938);
       strcpy(temp,m_track);
       time = gethrtime();
        return temp;
};
```

C.2.5 INTERFACE DEFINITIONS

```
typedef string<938> VtTrack;
interface Trax {
void print (in long signal);
void get (in long index, out VtTrack track, out double time);
};
```

C.3 VTPLATFORM SOURCE ROUTINES

C.3.1 CLIENT

```
// Client.C

// This demonstrate the use of both per-process and per-object filtering.
// Classes ProcessFilter and preProces demonstrate per-object filtering.
// Classes fiterPre and filterPost demonstrate per-object filtering.

// allow the defintion of the _individual_ system exceptions to be visible.
#define EXCEPTIONS

#include "Trax2.hh"
#include <stream.h>
```

```
// for extern "exit"
#include <stdlib.h>

// for timing
#include <sys/time.h>


#include "ProcFilt.h"

ProcessFilter PFilter;// create one instance of the per-process filter.
                      // This will monitor calls to and from this
                      // address space.

int main (int argc, char **argv) {
   long value = 0;
   Trax_var bVar;
   VtPlatformTrack VtX;
   hrtime_t start_hires, end_hires, diff_hires, Svr_time[15000];
   double time;
   int maxS;

   // need a hostname argument:
   if (argc < 2) {
           cerr << "usage:" << argv[0] << " <hostname>" << endl;
           exit(1);
    }

   // now try to bind to the target object.
   //see how long a bind takes
   try {
        cout << "TTTD: Attempting Bind..." << endl;
        start_hires = gethrtime();
        bVar = Trax::_bind (":xtracks",argv[1]);
        end_hires = gethrtime();
        diff_hires = end_hires - start_hires;
        cout << "TTTR: Bind time:  " << diff_hires << " ns" << endl;
        cout << "Start value: " << value << "\n";
   }
    catch (CORBA::SystemException &sysEx) {
        cerr << "Unexpected system exception" << endl;
        cerr << &sysEx;
         cerr << "Bind failed" << endl;
        exit(1);
 } catch (...) {
        // the bind failed
         cerr << "Unexpected exception : bind failed" << endl;
         exit(1);
  }


  time = 0;

  for (int i = 0; i < 10000; i++) {
      try {
        bVar->get (value, VtX, time);
        Svr_time[i] = time;
      } catch (...) {
```

```
            // should get no exceptions:
            cout << "Exception\n";
            cout << "call to Trax failed" << endl;
            exit(1);
        }

    }

    for(i=0; i<orqprmI-1 ; i++)
        { cout << orqprmA[i] << "->C1" << endl; };
        orqprmI = 0;
    for(i=0; i<orqpomI-1 ; i++)
        { cout << orqpomA[i] << "->C2" << endl; };
        orqpomI = 0;
    for(i=0; i<irpprmI-1 ; i++)
        { cout << irpprmA[i] << "->C7" << endl; };
        irpprmI = 0;
    for(i=0; i<irppomI-1 ; i++)
        { cout << irppomA[i] << "->C8" << endl; };
        irppomI = 0;
    for(i=0; i<10000; i++)
        { cout << Svr_time[i] << "->S9" << endl; };
    value = 16000001;
    bVar->print(value); //Print out table
    value = 1;

    return 0;
}
```

C.3.2 SERVER

```
// make the defintions of the _individual_ system exceptions to be visiable.
#define EXCEPTIONS
#include <stream.h>
#include "Trax2_i.h"

#include <sys/time.h>   // for timing

#include "ProcFilt.h"

// for extern "exit"
#include <stdlib.h>



ProcessFilter Filter_instance;   // install per-process filters.
void printtimes(void) {
        int i;
        for(i=0; i<irqprmI; i++)
                cout << irqprmA[i] << "->S3" << endl;
        irqprmI = 0;
        for(i=0; i<irqpomI; i++)
                cout << irqpomA[i] << "->S4" << endl;
        irqpomI =0;
```

```
	for(i=0; i<orpprmI; i++)
		cout << orpprmA[i] << "->S5" << endl;
	orpprmI = 0;
	for(i=0; i<orppomI; i++)
		cout << orppomA[i] << "->S6" << endl;
	orppomI =0;
};

int main(int , char *argv[]){
	hrtime_t start_hires, end_hires, diff_hires ;  // TTTTT

	// create the target object

	Trax_i mytrax;

	//
	// Export the server to the network
	//
	try {

	// tell Orbix that the server has completed its initialisation:
		cout << "TTTD: Attempting impl_is_ready..." << endl;
		start_hires = gethrtime();
		CORBA::Orbix.impl_is_ready("xtracks");
		end_hires = gethrtime();
		diff_hires = end_hires - start_hires;
		cout << "TTTR: impl_is_ready time:  " << diff_hires/1000000 << " ms" << endl;
	} catch (CORBA::SystemException &sysEx) {
		cerr << "Unexpected system exception" << endl;
		cerr << &sysEx;
		cerr << "Unexpected exception : impl_is_ready" << endl;
		exit(1);
	} catch (...) {
		// got an exception from impl_is_ready()
		cerr << "Unexpected exception : impl_is_ready" << endl;
		exit(0);
	}

	cout << "\n\nFilter Server shutdown." << endl;

	return 0;
}
```

C.3.3 OBJECT INCLUDES

```
#ifndef Trax2_ih
#define Trax2_ih

#include "Trax2.hh"

class Trax_i: public virtual TraxBOAImpl {
public:
	VtPlatformTrack VtP;
	Trax_i();
	~Trax_i();
		virtual void print (CORBA::Long signal, CORBA::Environment &IT_env=COR-
	BA::default_environment) ;
```

```
        virtual void get (CORBA::Long index, VtPlatformTrack& track, CORBA::Double& time, COR-
        BA::Environment &IT_env=CORBA::default_environment) ;
};


#endif
```

C.3.4 OBJECT METHODS

```
#define EXCEPTIONS
#include "Trax2_i.h"
#include <stdlib.h>
#include <sys/times.h>
#include <stream.h>
void printtimes(void);

Trax_i::Trax_i(){};
Trax_i::~Trax_i(){};
void Trax_i:: print (CORBA::Long signal, CORBA::Environment &IT_env) {
printtimes();
exit(1);
}

void Trax_i:: get (CORBA::Long index, VtPlatformTrack& track, CORBA::Double& time, CORBA::En-
        vironment &IT_env) {
VtP.data.quantity =1;
time=gethrtime();
}
```

C.3.5 VTTRACK INTERFACE DEFINITIONS

```
        //============================================================
//          NEW TRACK STRUCTURES
//============================================================

//
// SYSID
// VtSysid
// VtTrackStatus

// VtAou
// VtReport
// VtShortReport
// VtTracker
// VtElintStats
// VtRadReport
// VtIIIData
// VtIIIReport
// VtRadAndReport
// VtSignaReport
// VtTechData
```

```
// VtRemarks

// VtCandidate
// VtCandidateList

// VtTrknum
// VtTrackHeader
// VtTrackSearch

// VtPlatformData
// VtLinkData
// VtSpa25Data
// VtLateralTellData
// VtEmitterData
// VtAcousticData
// VtLandData
// VtUnitData

// VtPlatformTrack
// VtLinkTrack
// VtSpa25Track
// VtLateralTellTrack
// VtEmitterTrack
// VtAcousticTrack
// VtLandTrack
// VtUnitTrack



#define VT_MAX_RTN 12
#define VT_MAX_TRACKER_RPTS 10

#define VT_MAX_TRACK_SIZE 912

#define VT_MAX_LRAW 50

// LINK11 data types (0-15)
#define EMPTY_TYPE 0
#define ACDS_TYPE 1
#define LINK11_TYPE 2
#define LINK14_TYPE 3
#define LINK16_TYPE 4
#define LN2_TYPE 5
#define CEC_TYPE 6
#define LATTL_TYPE 7

// Platform data types (0-15)
#define TBMD_TYPE 1
#define SITE_TYPE 2


// Track amplification - other database numbers (0-15)
#define DB_NUM_TYPE_EMPTY 0
#define DB_NUM_TYPE_PIN1// PIN number (PN:)
#define DB_NUM_TYPE_DEV2// Developmental Site/Equipment number (DV:)
#define DB_NUM_TYPE_BE 3// Basic Encyclopedia number (BE:)
```

```
#define DB_NUM_TYPE_GENERIC4// Generic number (GN:)
#define DB_NUM_TYPE_AID5// AID number (MIIDS/IDB derived) (ID:)
#define DB_NUM_TYPE_VEHICLE6// Vehicle ID

// Raw data types
#define RAW_EMPTYEMPTY_TYPE
#define RAW_ACDSACDS_TYPE
#define RAW_LINK11LINK11_TYPE
#define RAW_LINK14LINK14_TYPE
#define RAW_LINK16LINK16_TYPE
#define RAW_LN2LN2_TYPE
#define RAW_CECCEC_TYPE
#define RAW_LATTLLATTL_TYPE

#define RAW_POS20


#define VT_ALERT_LEN 3
#define VT_ALERT_WORD_LEN 20
#define VT_BE_NUMBER_LEN 12
#define VT_CALLSIGN_LEN 20
#define VT_CN_LEN 15
#define VT_CATEGORY_LEN 3
#define VT_CD_LEN 8
#define VT_CL_LEN 11
#define VT_CHXREF_LEN 3
#define VT_CLASS_LEN 24
#define VT_CG_LEN 6
#define VT_DI_LEN 4
#define VT_FCODE_LEN 2
#define VT_FDI_LEN 4
#define VT_FLAG_LEN 2
#define VT_HOME_BASE_LEN 20
#define VT_HULL_LEN 6
#define VT_IFF_LEN 3
#define VT_MT_LEN 3
#define VT_MSG_TYPE_LEN 6
#define VT_NTDS_LEN 5
#define VT_PDDG_LEN 2
#define VT_PIF_CODE_LEN 4
#define VT_RAID_LEN 5
#define VT_SCONUM_LEN 8
#define VT_SHORTNAME_LEN 10
#define VT_SN_LEN 8
#define VT_SOURCE_LEN 6
#define VT_SUBJ_ID_LEN 24
#define VT_SERIAL_LEN 5
#define VT_SUBJ_TYPE_LEN 6
#define VT_SUBORD_LEN 24
#define VT_SUFFIX_LEN 5
#define VT_THREAT_LEN 3
#define VT_TYPE_LEN 6
#define VT_NTYPE_LEN 6
#define VT_UIC_LEN 6
#define VT_UNIT_IDENT_LEN 26
#define VT_XM_LEN 1
#define VT_XC_LEN 8
```

```
#define VT_XREF_LEN 4
#define VT_PLOT_ID_LEN 8


typedef struct A1 {

        long timestamp;
        long record;
        long machine;
        long database;
        long spare;

} SYSID ;


typedef struct A2 {
        char serial[20];
} VtSysid;


typedef struct A3 {
        long    dtg;                    // DTG from contributing report
        long    ctcno;                  // Contact serial number from contributing report
        char    serial[20];     // UID from track owner
} VtTagInformation;


typedef struct A4 {
        long    group_mask;                     // group membership bit-mask
        long    lastchange;                     // time last updated in any way by Tdbm

        long    rpts;                                   // Current number of reports
        long    maxrpts;                        // Max reports stored on this tracks

        short plot_id[VT_PLOT_ID_LEN];// symbol plot id
        unsigned short correlation;// Status from Correlator
        char toi_state;                 // TOI status flag
        char ownship;                   // flag for own ship

        long    mask;                                   // Status mask see below
        unsigned short ftn_cs;// Checksum on FTN (less FTN Command)
        unsigned short ctc_cs;// Checksum on CTC data
        unsigned short rmks_cs[4];// Checksum on RMKS data
        unsigned short rig_cs;// Checksum on RIG data
        unsigned short dep_cs;// Checksum on DEP data
        unsigned short des_cs;// Checksum on DES data
        unsigned short arr_cs;// Checksum on ARR data
        unsigned short rtn_cs;// Checksum on PAIR data
        unsigned short signa_cs;// Checksum on SIGNA data

        short force_code ;
        short force_type_id ;// Unique id for each force type

} VtTrackStatus;


// Values for status mask field
```

```
#define VtTrackArchivedMask(1L << 0)
#define VtTrackProtectedMask(1L << 1)
#define VtTrackTargetFileMask(1L << 2)

#define VtTrackResponsibleMask0x00000030
#define VtOTHResponsible0x00000000// OTH track
#define VtOTHMResponsible0x00000010// OTH track held by Organic
#define VtCDSResponsible0x00000020// Organic track held by OTH
#define VtL11Responsible0x00000030// L11 received from OTH
#define VtRTDReceivedMask0x00000040// Protection and responsibility
                                                  // received in a RTD line
#define VtNUTrackMask  0x00000100// NUTrack of non-Platform
#define VtFCSXmitMask           0x00000200  // Transmit updates to FCS
#define VtFCSAncestorMask0x00000400// Track is from FCS database
#define Vt2L11XmitMask          0x00000800  // 2-way Link-11 track
#define Vt2L11TrkMask           0x00001000  // 2-way Link-11 track
#define Vt2L11EmergMask         0x00002000  // 2-way Link-11 Emergency track
#define Vt2L11ForceMask         0x00004000  // 2-way Link-11 Force Tell track

       // Sample form for following is: isArchived(VtTrackStatusMask(trk))

#define isArchived(X)  ((X) & VtTrackArchivedMask)
#define isProtected(X) ((X) & VtTrackProtectedMask)
#define isTargetFile(X)((X) & VtTrackTargetFileMask)

#define isOTHResponsible(X)(((X) & VtTrackResponsibleMask) == VtOTHResponsible)
#define isOTHMResponsible(X)(((X) & VtTrackResponsibleMask) == VtOTHMResponsible)
#define isCDSResponsible(X)(((X) & VtTrackResponsibleMask) == VtCDSResponsible)
#define isL11Responsible(X)(((X) & VtTrackResponsibleMask) == VtL11Responsible)
#define isRTDReceived(X)((X) & VtRTDReceivedMask)

#define isNUTrack(X)((X) & VtNUTrackMask)

#define isFCSXmitTrack(X)       ((X) & VtFCSXmitMask)
#define isFCSAncestor(X)((X) & VtFCSAncestorMask)
#define is2L11XmitTrack(X)      ((X) & Vt2L11XmitMask)
#define is2L11Track(X)          ((X) & Vt2L11TrkMask)
#define is2L11EmergTrack(X)     ((X) & Vt2L11EmergMask)
#define is2L11ForceTrack(X)     ((X) & Vt2L11ForceMask)

       // Sample form for following is: mask = VtTrackStatusMask(trk);
       //                                      VtSetArchived(mask);
       //                                       VtSetTrackArchiveMask(trk,mask);

#define VtSetArchived(X)((X) |= VtTrackArchivedMask)
#define VtSetProtected(X)((X) |= VtTrackProtectedMask)
#define VtSetTargetFile(X)((X) |= VtTrackTargetFileMask)

#define VtSetOTHResponsible(X)((X) = (((X) & ~VtTrackResponsibleMask) | VtOTHResponsible))
#define VtSetOTHMResponsible(X)((X) = (((X) & ~VtTrackResponsibleMask) | VtOTHMResponsible))
#define VtSetCDSResponsible(X)((X) = (((X) & ~VtTrackResponsibleMask) | VtCDSResponsible))
#define VtSetL11Responsible(X)((X) = (((X) & ~VtTrackResponsibleMask) | VtL11Responsible))
#define VtSetRTDReceived(X)((X) |= VtRTDReceivedMask)

#define VtSetNUTrack(X)((X) |= VtNUTrackMask)
```

```
#define VtSetFCSXmit(X)          ((X) |= VtFCSXmitMask)
#define VtSetFCSAncestor(X)((X)  |= VtFCSAncestorMask)
#define VtSet2L11Xmit(X)         ((X) |= Vt2L11XmitMask)
#define VtSet2L11Trk(X)          ((X) |= Vt2L11TrkMask)
#define VtSet2L11Emerg(X)        ((X) |= Vt2L11EmergMask)
#define VtSet2L11Force(X)        ((X) |= Vt2L11ForceMask)

#define VtClearArchived(X)            ((X) &= ~VtTrackArchivedMask)
#define VtClearProtected(X)           ((X) &= ~VtTrackProtectedMask)
#define VtClearTargetFile(X)   ((X) &= ~VtTrackTargetFileMask)
#define VtClearRTDReceived(X)  ((X) &= ~VtRTDReceivedMask)
#define VtClearTrackResponsibleMask(X)((X) &= ~VtTrackResponsibleMask)

#define VtClearNUTrack(X)             ((X) &= ~VtNUTrackMask)

#define VtClearFCSXmit(X)                   ((X) &= ~VtFCSXmitMask)
#define VtClearFCSAncestor(X)  ((X) &= ~VtFCSAncestorMask)
#define VtClear2L11Xmit(X)                  ((X) &= ~Vt2L11XmitMask)
#define VtClear2L11Trk(X)                   ((X) &= ~Vt2L11TrkMask)
#define VtClear2L11Emerg(X)                 ((X) &= ~Vt2L11EmergMask)
#define VtClear2L11Force(X)                 ((X) &= ~Vt2L11ForceMask)


typedef struct A5  {
       long   typ;                       // 1-ell, 2-bbox, 3-lob
       float  brg;                       // bearing
       float  a1;                               // meaning of a1 & a2 depend on typ
       float  a2;                               //
} VtAou  ;


typedef struct A6  {

       double lat;                       // latitude in degrees
       double lng;                       // longitude in degrees
       float  cse;                       // course (DEGT)
       float  spd;                       // speed (KTS)
       float  alt;                       // altitude (+FT/-FT)
       float  anglelv;           // angle of elevation/depression
       long          dtg;                       // Julian seconds since 1 Jan 1970
       long          rawdata;            // raw data file record (ilog)
       long          ctcno;                     // contact serial number
       long          datano;                    // associated data (serial number)
       long          trkrec;                    // trkrec that owns report
       VtAou  aou;                       // area of uncertainty

       char   sensor[7];         // sensor
       char   source[7];         // screen field
       char   rdf_rf[11];        // freq assoc with ew report
       char   callsign[9];   // international radio callsign
       char   xref[5];           // cross-ref flag for origin of report
       char   chxref[4];         // input channel cross-reference
       char   classification;// classification of the contact

       char   releasibility; // releasibilty of the contact
       char   archived;              // Marks report as having been archived
       unsigned short mask;// mask (see below)
```

```
        long    key;                        // unique relational DB key
        unsigned short checksum;// checksum identification
        unsigned short checkkey;// checksum key

        char    error;                      // report received with cs error
        char    ci[3];                      // Correlation Index
                                                           // spare bytes (pad for 4 byte
        alignment)
} VtReport;

//      VtReport.mask defined as character with following bit masks

#define RPT_CHANNEL0x00000001 // 0 - internal, 1 - external
#define RPT_EXISTS0x00000002// 0 - doesn't exist, 1 - exists

#define RPT_PERMANENT0x00000004// for NIPS provided information to prevent

#define RPT_PRECISION0x00000038// 3 bits coordinate precision MTypes.h:
               // {PosLatLong,PosLatLongDMS,PosLatLongDMST,PosMGR,PosUTM,PosGeoRef}

#define RPT_TYPE0x000001c0// 3 bits TBMD identification (see below)

#define VtReportTBMDMask RPT_TYPE
#define VtReportTBMDLaunch (1L << 6)
#define VtReportTBMDObserv(2L << 6)
#define VtReportTBMDObservBO(3L << 6)
#define VtReportTBMDImpact(4L << 6)
#define VtIsReportTBMD(X)(((X->mask) & VtReportTBMDMask))
#define VtIsReportTBMDLaunch(X)(((X->mask) & VtReportTBMDMask) == VtReportTBMDLaunch)
#define VtIsReportTBMDObserv(X)(((X->mask) & VtReportTBMDMask) == VtReportTBMDObserv)
#define VtIsReportTBMDObservBO(X)(((X->mask) & VtReportTBMDMask) == VtReportTBMDObservBO)
#define VtIsReportTBMDImpact(X)(((X->mask) & VtReportTBMDMask) == VtReportTBMDImpact)


// Mask values for datano field

#define VtReportDataTypeMask0xf0000000
#define VtReportRadDataMask(1L << 28)
#define VtReportSignaDataMask(1L << 29)
#define VtReportSIDataMask(1L << 30)
#define VtReportOtherDataMask(1L << 31)

#define RpthasData(X)(((X) & VtReportDataTypeMask))
#define ReporthasData(X)(((X->datano) & VtReportDataTypeMask))
#define RpthasDataNumber(X)(((X) & ~VtReportDataTypeMask))
#define ReporthasDataNumber(X)(((X->datano) & ~VtReportDataTypeMask))
#define RpthasRadData(X)(((X) & VtReportDataTypeMask) == VtReportRadDataMask)
#define ReporthasRadData(X)(((X->datano) & VtReportDataTypeMask) == VtReportRadDataMask)
#define RpthasSignaData(X)(((X) & VtReportDataTypeMask) == VtReportSignaDataMask)
#define ReporthasSignaData(X)(((X->datano) & VtReportDataTypeMask) == VtReportSignaDataMask)
#define RpthasSIData(X)(((X) & VtReportDataTypeMask) == VtReportSIDataMask)
#define ReporthasSIData(X)(((X->datano) & VtReportDataTypeMask) == VtReportSIDataMask)
#define RpthasOtherData(X)(((X) & VtReportDataTypeMask) == VtReportOtherDataMask)
#define ReporthasOtherData(X)(((X->datano) & VtReportDataTypeMask) == VtReportOtherDataMask)

#define VtSetReportRadData(X)((X) |= VtReportRadDataMask)
```

```
#define VtSetReportSignaData(X)((X) |= VtReportSignaDataMask)
#define VtSetReportSIData(X)((X) |= VtReportSIDataMask)
#define VtSetReportOtherData(X)((X) |= VtReportOtherDataMask)

#define VtReportDataNumber(X)((X) & ~VtReportDataTypeMask)
#define VtSetReportDataNumber(X,Y)((X) |= ((Y) & ~VtReportDataTypeMask))

#define VtClearReportDataType(X)((X) &= ~VtReportDataTypeMask)
#define VtClearReportRadData(X)((X) &= ~VtReportRadDataMask)
#define VtClearReportSignaData(X)((X) &= ~VtReportSignaDataMask)
#define VtClearReportSIData(X)((X) &= ~VtReportSIDataMask)
#define VtClearReportOtherData(X)((X) &= ~VtReportOtherDataMask)


typedef struct A7 {

        double  lat;                        // latitude in degrees
        double  lng;                        // longitude in degrees
        float   cse;                        // course (DEGT)
        float   spd;                        // speed (KTS)
        long    dtg;                        // Julian seconds since 1 Jan 1970
        VtAou   aou;                        // area of uncertainty

} VtShortReport;


typedef struct A8 {

        long dtg;                           // Time of tracker solution
        long nrpts;                         // Number of reports for solution

        float lat;                          // Lat of tracker solution
        float lng;                          // Lng of tracker solution
        float lat_spd;              // Speed (lat) of tracker solution
        float lng_spd;              // Speed (lng) of tracker solution
        float cov[10];              // Covariance Matrix of tracker solution
        float alpha;                // Alpha parameter of tracker solution
        float sigma;                // Sigma parameter of tracker solution

        float cse;                          // Computed course
        float spd;                          // Computed speed
        float tol;                          // time on leg in hours
        float ave_spd;              // average speed on leg

} VtTracker ;


typedef struct A9 {

        double  pri_mean;           // Mean value of associated pri's
        double  pri_sigma;          // sigma value of associated pri's
        double  pri_sumsquares;// sum-of-squares value of associated pri's
        long    pri_items;

        double  pri_sumwobs;    // sum weighted observations
        double  pri_sumw;           // sum inverse reported observation deviations
        double  pri_sumwsqobssq;// sum squared observations weighed inversely by
```

```
        //                                              reported observation vari-
    ance
    double pri_sumwsqobs; // sum observations weighed inversely by
        //            reported observation variance
    double pri_sumwsq;          // sum inverse reported observation variance



    double scan_mean;           // mean of scan
    double scan_sigma;          // standard deviation of scan
    double scan_sumsquares;// sum of squares of scan
    long   scan_items;          // number of scan items

    double scan_sumwobs;  // sum weighted observations
    double scan_sumw;           // sum inverse reported observation deviations
    double scan_sumwsqobssq;// sum squared observations weighed inversely by
        //                                              reported observation vari-
    ance
    double scan_sumwsqobs;// sum observations weighed inversely by
        //                                              reported observation vari-
    ance
    double scan_sumwsq;   // sum inverse reported observation variance



    double rf_mean;             // mean of rf
    double rf_sigma;            // standard deviation of rf
    double rf_sumsquares; // sum of squares of rf
    long   rf_items;            // number of rf items

    double rf_sumwobs;          // sum weighted observations
    double rf_sumw;             // sum inverse reported observation deviations
    double rf_sumwsqobssq;// sum squared observations weighed inversely by
        //                                              reported observation vari-
    ance
    double rf_sumwsqobs;  // sum observations weighed inversely by
                        //                              reported observation vari-
    ance
    double rf_sumwsq;           // sum inverse reported observation variance

    double rf_bcmean;           // mean of RFBCs

} VtElintStats ;


typedef struct  A10 {

    long   datano;                  // number of this rad line report
    long   ctcno;                   // Number of the report associated with elint

    double freq;                    // frequency of the emitter(MHZ)
    double pri;                     // pulse repetition interval (micro-sec)
    double prf;                     // pule repetition frequency (pulses per sec)
    double pw;                          // pulse width (micro-sec)
    double scan_rate;           // scan rate for radar (SPC) (1/HZ)

    double bb_pri;                  // Basebanded pri
    double bb_scan_rate;  // Basebanded scan_rate
    char   bb_pri_mode;   // Reported, Crystal, Range, Adaptive
```

```
        char    bb_scan_rate_mode;// Reported or Range
        char    elnot1conf;           // Primary elnot confidence (NN < 10)
        char    elnot2conf;           // Secondary elnot confidence (NN < 10)

        float   freq_stability;// reported sigma for RF
        float   pri_stability; // reported sigma for PRI
        float   scan_stability;// reported sigma for scan rate

        long    dtg;                          // date-time-group of report
        long    scn;                          // sequential contact number

        char    elnot1[6];            // ELINT notation or name
        char    elnot2[6];            // Secondary elnot [ANNNA,ANNNN,NNNAA]

        char    ci[3];                        // Correlation Index
        char    emitter[16];  // emitter name (DON KEY)
        char    scan_type[5]; // scan type (four letters, mapped to 1 ? )

        char    stagger_legs; // stagger legs N
        char    pri_type;             // type of PRI interval
        char    spare[2];             // spare bytes (pad for 4 byte alignment)

    } VtRadReport, VtRadData;


    typedef struct  A11 {

        char    indicator[2]; // warning indication A
        char    msg_type[3];  // Message type
        char    fcode[3];             // Force Code as described in OTG spec

        char    flag[3];      // flag of long BE (CCNNNNANNNNN)
                //                                                  CC = country code

        char    db_type;              // database number type: DB_NUM_TYPE_PIN, ...
        char    db_num[14];
                                                      // BE number NNNNANNNNN
                                                      //         NNNN = world area code
                                                      //          A = BE Type (C)omplex,
        (E)_site
                                                      //         NNNNN = world area code
        PIN

        char    sconum[7];            // tgtid [NNNNNN,NANNNNN,ENNNNN,DANNNN,DNNNNN]
                                                      // PARAGON uses this field to store
        PIN

        char    category[4];  // category
        char    threat[4];            // threat class

        char    freq_agility; // agility of RFs

        char    spare[6];             // spare bytes (pad for 4 byte alignment)

    } VtIIIData;


    typedef struct A12 {
```

```
        VtReport rpt;
        VtRadReport rad;
        VtIIIData iii;
} VtIIIReport ;


typedef struct  A13 {
        VtReport rpt;
        VtRadReport rad;
} VtRadAndReport;


typedef struct  A14 {
        float   freq;
        float   bw;
        char    cn[VT_CN_LEN+1];
        char    sn[VT_SN_LEN+1];
        char    mt[VT_MT_LEN+1];
        char    xm[VT_XM_LEN+1];
        char    cg[VT_CG_LEN+1];
        char    cl[VT_CL_LEN+1];
        char    xc[VT_XC_LEN+1];
        char    cd[VT_CD_LEN+1];
                char    spare[4];                       // byte alignment
} VtSIRptAttribData;


typedef struct  A15 {
        long     datano;                // index to SI contact data
        long     ctcno;                 // index to VtReport.ctcno
        long     quantity;              // number of items
        VtSIRptAttribData       attr;   // report attribute data
        char    source[VT_SOURCE_LEN+1];// reported source
        char    fdi[VT_FDI_LEN+1];      // File Distrib Indicator
        char    pddg[VT_PDDG_LEN+1];    // reported pddg
        char    serial[VT_SERIAL_LEN+1]; // serial number
        char    msg_type[VT_MSG_TYPE_LEN+1];    // Recvd msg type
                char    iff[VT_IFF_LEN+1];// Id Friend or Foe
        char    sanitizable;            // Is contact sanitizable
} VtSIReport;


typedef struct A16  {
        VtReport rpt;
        VtSIReport si;
} VtSiAndReport;


typedef struct A17 {
        long    datano;                 // index to Acoutsic data

        long    ctcno;                  // index to VtReport.ctcno
        long     dtg;            // date-time-group of report
        double  freq;                   // (minimum) fundamental frequency in HZ
        double  freq_max;       // (maximum) fundamental frequency in HZ
        float   rpm;                    // revolutions per minute
```

```
        float   tpk;                    // turns per knot
        char    harm[24];       // Harmonics
        char    source[16];     // source of information
} VtSignaReport;


typedef struct A18 {
        VtReport rpt;
        VtSignaReport signa;
} VtSignaAndReport;


#define VT_TLEVEL_CASE 1
#define VT_TLEVEL_DSUB 1
#define VT_TLEVEL_DESIG 2
#define VT_TLEVEL_TGT 3
#define VT_TLEVEL_CONT 4
#define VT_TLEVEL_POD 4
#define VT_TLEVEL_UNKNOWN 4


typedef struct A19 {

        long  etd;
        long  eta;
        long  arrdtg;
        char appgrp[7];
        char hullprof[13];
        char stern[9];
        char uprights[13];
        char prop[6];
        char tonnage[7];
        char length[5];
        char beam[5];
        char draft[3];
        char blades[3];
        char shafts[3];
        char depport[19];
        char depflag[3];
        char desport[19];
        char desflag[3];
        char arrport[19];
        char arrflag[3];
        char dep_cargo[4][4];
        char des_cargo[4][4];
        char arr_cargo[4][4];

} VtTechData;


typedef struct  A20 {
        char cmd[15];           // command from which remarks came
        char line[4][65];// four remarks lines
        char spare[5];          // byte alignment
} VtRemarks;

typedef struct A21 {
```

```
        long num;                    // current number in rmks[] array
        long size;                   // current size of rmks array
        VtRemarks rmks;// array of pointers to remarks WAS **rmks
} VtRcvdRemarks;


typedef struct A22 {
        long trkrec;                 // candidate track number
        long tm_dif;                 // calculated time difference
        short dist_dif;       // calculated dist difference
        short spd_reqd;       // speed required to position
        short cse_reqd;       // cse required to position
        short ascr;                  // attribute score -100 to 100
        short pscr;                  // position score  -100 to 100
        short pri_scr;        // PRI score  -100 to 100
        short scan_scr;       // Scan rate score  -100 to 100
        short rf_scr;         // RF score  -100 to 100
} VtCandidate;

typedef struct  A23 {
        VtCandidate track[7];
} VtCandidateList;


typedef struct  A24 {
        long dtg;                           // dtg of associated POS
        short state;                 // RTN state (spare)
        char trknum[7];              // Track Number string
        char cmd[15];                // Source command of trknum
} VtTrknum;

// VtTrknum state values for RTN priority order: Fotc, Copy then Local

#define VtTrknumFotcRTN2
#define VtTrknumCopyRTN1
#define VtTrknumLocalRTN0


typedef struct A25 {
        long type;                          // Type of track
        long trkrec;                        // Record of track
        long source;                        // Source bit mask
        long assoc;                         // Associated trkrec
        long child;                         // child trkrec (conditionally)
        long machine;                // Machine mask for Local tracks bits 0-31
        char serial[20];      // Unique identifier
        char ltn[8];                 // Local Track Number
                                                          // Software Version (future)
} VtTrackHeader;


typedef struct  A26 {

        VtTrknum ftn;                // FOTC Track Number
        VtTrknum rtn[VT_MAX_RTN];// Array of Rcvd Track Numbers

        char ltn[7];                 // Local Track Number
```

```
        char shipclass[12];    // class of ship
        char name[27];                 // name of the track
        char trademark[21];    // trademark
        char type[7];          // type of ship (i.e. DDG, CVN)
        char hull[7];                  // hull number of ship
        char flag[3];          // flag of fft
        char sconum[7];                // NOSIC ID
        char pif[5];                   // pif as reported by intel
        char ntds[6];                  // ntds track number if associated
        char di[5];                            // discrete identifier
        char callsign[9];      // call sign of last entered call sign
        char uic[7];                   // Unit Identification Code
        char elnot[6];                 // elnot
        char emitter[16];      // Emitter name

        char alert[4];                 // alert class of track (HIT,TGT)
        char category[4];      // Category (NAV,AIR, etc.)
        char threat[4];                // Threat class
        char xref[5];
        char chxref[4];                // input channel cross-reference
        char shortname[11];    // Symbol Annotation

} VtTrackSearch;


typedef struct A27 {

        VtTrknum ftn;                  // FOTC Track Number
        VtTrknum rtn[VT_MAX_RTN];// Array with up to 12 Rcvd Track Number

        char shipclass[12];    // class of ship
        char name[27];                 // name of the track
        char trademark[21];    // trademark
        char type[7];          // type of ship (i.e. DDG, CVN)
        char hull[7];                  // hull number of ship
        char flag[3];          // flag of fft
        char sconum[7];                // NOSIC ID
        char pif[5];                   // pif as reported by intel
        char ntds[6];                  // ntds track number if associated
        char di[5];                            // discrete identifier
        char callsign[9];      // call sign of last entered call sign
        char uic[7];                   // Unit Identification Code

        long  quantity;              // number in track
    char home_base[21]; // home base/port
        char db_type;                  // database number type: DB_NUM_TYPE_PIN, ...
        char db_num[14];
                                                               // BE number NNNNANNNNN
                                                               //         NNNN = world area code
                                                               //          A = BE Type (C)omplex,
        (E)_site
                                                               //         NNNNN = world area code
        PIN


        char alert[4];                 // alert class of track (HIT,TGT)
        char fcode[3];                 // Force Code as described in BGDBM spec
        char category[4];      // Category (NAV,AIR, etc.)
```

```
        char threat[4];             // Threat class
        char shortname[11];    // Symbol Annotation
        char xref[5];
        char orig_xref[5];
        char chxref[4];             // input channel cross-reference

        double latfixed;
        double lngfixed;

} VtPlatformData;


typedef struct A28 {
        double  lat;                        // latitude in degrees
        double  lng;                        // longitude in degrees
        float   cse;                        // course (DEGT)
        float   spd;                        // speed (KTS)
        float   alt;                        // altitude (+FT/-FT)
        float   anglelv;            // angle of elevation/depression
        long    dtg;                        // Julian seconds since 1 Jan 1970
        unsigned short mask;// mask (see VtReport)
        char db_type;               // database number type: DB_NUM_TYPE_PIN, ...
        char db_num[14];
} VtTBMData;


typedef struct A29 {

        long type;                                                      // LINK11 data types (0-15)
        long nbytes;                                                    //   number   of   link_data
bytes
        unsigned long link_data[VT_MAX_LRAW];// Link data msg buffer

} VtRawData, LRAWDATA;


typedef struct  A30 {

        long updates;               // number of times track has been updated
        long tkno;                          // last received track number

        short link;                         // Link block LINK_A, ... LINK_D
        short quality;              // track quality
        short ru;                           // reporting unit
        short di;                           // pif as reported by link

        short ctsx;
        short local_quality;
        short engagement;
        short spare;                // spare element

        short mode1iff;             // Mode 1 as reported by link
        short mode2iff;             // Mode 2 as reported by link
        short mode3iff;             // Mode 3 as reported by link
        short mode4iff;             // Mode 4 as reported by link

        char threat[4];             // threat class F, H, AF, AE, ..
```

```
        char category[4];       // category NAV, SUB, ...
        char alert[4];                  // Alert
        char shortname[11];     // Symbol Annotation
        char xref[5];                   // Source xref
        VtRawData lrawdata;     // ACDS, Link11 or Link16 specific data

} VtLinkData;


typedef struct A31 {

        long fcsno;                             // FCS track number
        char fcspfx;                    // FCS track prefix S, R, V, C, ...
        char status_mask;       // target status mask:
                //                                      bit 0 engage
                //                                      bit 1 avoid
                //                                      bit 2 CST
                //                                      bit 3 spare
                //                                      bit 4-5 weapon selection [0-3]
                //                                      bit 6-7 spare


        char tktype;                    // FCS NTDS-like track type
        char weather;                   // weather in vicinity of target UNKNOWN, CLEAR, RAIN

        char threat[4];                 // threat class F, H, AF, AE, ..
        char category[4];       // category NAV, SUB, ...
        char alert[4];                  // Alert
        char shortname[11];     // Symbol Annotation
        char xref[5];                   // Source xref

} VtFCSData;

#define VtFCSWthrUnk(0)
#define VtFCSWthrClr(1)
#define VtFCSWthrRain(2)

#define VtFCSEngageMask (1)
#define VtFCSAvoidMask(1 << 1)
#define VtFCSCstMask(1 << 2)
#define VtFCSWeaponMask(3 << 4)

#define VtSetFCSEngage(X)((X) |= VtFCSEngageMask)
#define VtSetFCSAvoid(X)((X) |= VtFCSAvoidMask)
#define VtSetFCSCst(X) ((X) |= VtFCSCstMask)
#define VtSetFCSWeapon(X,Y)((X) = ((X) & ~VtFCSWeaponMask) | ((Y) << 4))

#define VtClearFCSEngage(X)((X) &= ~VtFCSEngageMask)
#define VtClearFCSAvoid(X)((X) &= ~VtFCSAvoidMask)
#define VtClearFCSCst(X)((X) &= ~VtFCSCstMask)
#define VtClearFCSWeapon(X)((X) &= ~VtFCSWeaponMask)

#define isFCSEngage(X) ((X) & VtFCSEngageMask)
#define isFCSAvoid(X)  ((X) & VtFCSAvoidMask)
#define isFCSCst(X)             ((X) & VtFCSCstMask)
#define VtFCSWeapon(X) (((X) & VtFCSWeaponMask) >> 4)
```

```
typedef struct A32 {

        char local_tn[6];
        char system_tn[6];

        char name[27];

        char threat[4];              // threat class F, H, AF, AE, ..
        char category[4];      // category NAV, SUB, ...
        char alert[4];               // Alert
        char shortname[11];    // Symbol Annotation
        char xref[5];                // Source xref

} VtSpa25Data;


typedef struct A33 {

        char local_tn[6];

        char name[27];

        char threat[4];              // threat class F, H, AF, AE, ..
        char category[4];      // category NAV, SUB, ...
        char alert[4];               // Alert
        char shortname[11];    // Symbol Annotation
        char xref[5];                // Source xref

        float cpa_dist;
        long cpa_dtg;

} VtRaycasVData;


typedef struct A34 {

        long updates;                // number of times track has been updated
        short link;                        // Link block LINK_A, ... LINK_D
        short tkno;                        // last received track number
        short quality;               // track quality
        short ru;                          // reporting unit
        short di;                          // pif as reported by link
        short mode1iff;              // pif as reported by link
        short mode2iff;              // pif as reported by link
        short mode3iff;              // pif as reported by link
        short mode4iff;              // pif as reported by link

        char threat[4];              // threat class F, H, AF, AE, ..
        char category[4];      // category NAV, SUB, ...
        char alert[4];               // Alert
        char shortname[11];    // Symbol Annotation
        char xref[5];                // Source xref

} VtLateralTellData;
```

```
typedef struct  A35 {

        VtElintStats stats;    // Ellong statistics

        char emitter[16];      // Emitter name (DON KAY)

        char elnot1[6];                // Primary elnot  [ANNNA,ANNNN,NNNAA]
        char elnot1conf;       // Primary elnot confidence (NN < 10)
        char elnot2[6];                // Secondary elnot [ANNNA,ANNNN,NNNAA]
        char elnot2conf;       // Secondary elnot confidence (NN < 10)

        char flag[3];          // flag of long BE (CCNNNNANNNNN)
                                      //                          CC = country code

        char db_type;                  // database number type: DB_NUM_TYPE_PIN, ...
        char db_num[14];
                                                          // BE number NNNNANNNNN
                                                          //         NNNN = world area code
                                                          //          A = BE Type (C)omplex,
        (E)_site
                                                          //         NNNNN = world area code
        PIN

        char sconum[7];                // tgtid [NNNNNN,NANNNNN,ENNNNN,DANNNN,DNNNNN]
                                                          // PARAGON uses this field to store
        PIN

        char alert[4];                 // alert class of track (HIT,TGT)
        char fcode[3];                 // Force Code as described in BGDBM spec
        char category[4];      // Category (NAV,AIR, etc.)
        char threat[4];                // Threat class
        char shortname[11];    // Symbol Annotation
        char xref[5];
        char orig_xref[5];
        char chxref[4];                // input channel cross-reference

} VtEmitterData, VtElintData;


typedef struct A36 {

        char trademark[21];    // Trademark of sub
        char shipclass[12];    // Class of sub

        char prop[6];                  // propulsion NUC, UNK, DES
        char sconum[7];                // tgtid [NNNNNN,NANNNNN,ENNNNN,DANNNN,DNNNNN]
                                                          // PARAGON uses this field to store
        PIN

        char flag[3];          // flag of fft
        char type[7];

        char alert[4];                 // alert class of track (HIT,TGT)
        char fcode[3];                 // Force Code as described in BGDBM spec
        char category[4];      // Category (NAV,AIR, etc.)
        char threat[4];                // Threat class
        char shortname[11];    // Symbol Annotation
        char xref[5];
        char orig_xref[5];
```

```
        char chxref[4];               // input channel cross-reference

} VtAcousticData;


typedef struct A37 {

        char name[39];                // Name from NIPS displayed as 26 char string
                                                  // Include ???: subject[41] - BUN-
        KER, APRON,

        char flag[3];         // flag of long BE (CCNNNNANNNNN)

        char db_type;                 // database number type: DB_NUM_TYPE_PIN, ...
        char db_num[14];
                                                  // BE number NNNNANNNNN
                                                  //         NNNN = world area code
                                                  //          A = BE Type (C)omplex,
        (E)_site
                                                  //         NNNNN = world area code
        PIN

        char funcl[6];              // Funcl code for site NNNNN (aka NIPS CATEGORY)
                              //                               from the AIF (Automated Intel-
        ligence File)
        char status[4];               // Site Status
        char function[3];     // Site function (AA: RadarFuncDefaults.h)
        char radius[5];               // Site Radius NNNN
        char source_data[8];// Source data

        char alert[4];                // alert class of track (HIT,TGT)
        char fcode[3];                // Force Code as described in BGDBM spec
        char category[4];     // Category (NAV,AIR, etc.)
        char threat[4];               // Threat class
        char shortname[11];   // Symbol Annotation
        char xref[5];
        char orig_xref[5];
        char chxref[4];               // input channel cross-reference

} VtLandData;


typedef struct A38 {

        VtTrknum rtn[VT_MAX_RTN];// Array of Rcvd Track Numbers

        char name[31];

        char uic[7];                  // Unit Ident Code
        char echelon[8];      // Echelon ??? chars
        char service[4];      // Service ??? chars
        char platform[7];     // Platform ??? chars
        char strength[4];     // Force strength ??? chars
        char org_type[9];     // Organization Type ??? chars
        char embarked[31];

        char flag[3];         // flag of fft
        char type[7];                 // type as in MAU etc
```

```
        char shortname[11];    // Symbol Annotation
        char category[4];
        char threat[4];
        char alert[4];                // alert class of track (HIT,TGT)
        char fcode[3];                // Force Code as described in BGDBM spec
        char track_type;
        char xref[5];
        char orig_xref[5];
        char chxref[4];               // input channel cross-reference

 } VtUnitData;


 typedef struct  A39 {

        char local_tn[6];           // Optional, but likely to be useful
        char name[27];              // Optional, but likely to be useful
        char alert[4];                // Alert
        char fcode[3];                // Force Code as described in BGDBM spec
        char category[4];     // Category (NAV,AIR, etc.)
        char threat[4];               // Threat class
        char shortname[11];   // Symbol Annotation
        char xref[5];
        char chxref[4];               // input channel cross-reference

 } VtMinExtData;


typedef struct  A40 {
        long      si_id;                           // Unique SI identifier
        long      quantity;                        // number in track
        char      flag[VT_FLAG_LEN+1];            // flag
        char      threat[VT_THREAT_LEN+1];        // threat class
        char      category[VT_CATEGORY_LEN+1];    // category (AIR, NAV)
        char      native_type[VT_NTYPE_LEN+1];    // native type xlation
        char      class[VT_CLASS_LEN+1];          // shipclass or
                                                  // aircraft type
        char      hull[VT_HULL_LEN+1];            // hull number of ship
        char      fcode[VT_FCODE_LEN+1];          // force code
        char      home_base[VT_HOME_BASE_LEN+1];  // home base/port
        char      op_subord[VT_SUBORD_LEN+1];     // operational subord
        char      admin_subord[VT_SUBORD_LEN+1];  // admin subord
        char      callsign[VT_CALLSIGN_LEN+1];    // track's callsign
                                                  // or callword
        char      subj_type[VT_SUBJ_TYPE_LEN+1];  // Subject type
        char      raid[VT_RAID_LEN+1];// raid number
        char      pddg[VT_PDDG_LEN+1];            // pddg
        char      suffix[VT_SUFFIX_LEN+1];        // suffix
        char      alert_word[VT_ALERT_WORD_LEN+1];        // alert word

              char    xref[VT_XREF_LEN+1];
              char    chxref[VT_CHXREF_LEN+1];// channel xref
              char    shortname[VT_SHORTNAME_LEN+1];// Symbol Annotation
              char    spare[3];                                  // byte alignement
} VtSIData;
```

```
typedef struct  A41 {

        VtTrackHeader hdr;     // Track Header
        VtTrackStatus trkstat;// Track status
        VtTracker state;       // Tracker state at time of last report
        VtReport rpt;                  // Last Associated report
        VtPlatformData data;// Data associated with platform tracks

} VtPlatformTrack, VtPTrack;


typedef struct A42 {

        VtTrackHeader hdr;     // Track Header
        VtTrackStatus trkstat;// Track status
        VtTracker state;       // Tracker state at time of last report
        VtReport rpt;                  // Last Associated report
        VtLinkData data;       // Data Associated with Link Track

} VtLinkTrack, VtLTrack;


typedef struct A43 {

        VtTrackHeader hdr;     // Track Header
        VtTrackStatus trkstat;// Track status
        VtTracker state;       // Tracker state at time of last report
        VtReport rpt;                  // Last Associated report
        VtFCSData data;                // Data Associated with FCS Track

} VtFCSTrack, VtFTrack;


typedef struct A44 {

        VtTrackHeader hdr;     // Track Header
        VtTrackStatus trkstat;// Track status
        VtTracker state;       // Tracker state at time of last report
        VtReport rpt;                  // Last Associated report
        VtSpa25Data data;      // Data Associated with Link Track

} VtSpa25Track, VtSTrack;


typedef struct  A45 {

        VtTrackHeader hdr;     // Track Header
        VtTrackStatus trkstat;// Track status
        VtTracker state;       // Tracker state at time of last report
        VtReport rpt;                  // Last Associated report
        VtRaycasVData data;    // Data Associated with Link Track

} VtRaycasVTrack, VtRTrack;


typedef struct A46  {
```

```
        VtTrackHeader hdr;     // Track Header
        VtTrackStatus trkstat;// Track status
        VtTracker state;       // Tracker state at time of last report
        VtReport rpt;                 // Last Associated report
        VtLateralTellData data;// Data Associated with Link Track

} VtLateralTellTrack, VtTTrack;


typedef struct  A47 {

        VtTrackHeader hdr;     // Track Header
        VtTrackStatus trkstat;// Track status
        VtTracker state;       // Tracker state at time of last report
        VtReport rpt;                 // Last Associated report
        VtRadReport rad;              // Last Associated Rad data
        VtEmitterData data;    // Emitter attributes

} VtEmitterTrack, VtETrack;


typedef struct  A48 {

        VtTrackHeader hdr;     // Track Header
        VtTrackStatus trkstat;// Track status
        VtTracker state;       // Tracker state at time of last report
        VtReport rpt;                 // Last Associated report
        VtSignaReportsigna;    // Last reported Signa data
        VtAcousticData data;// Acoustic attributes

} VtAcousticTrack, VtATrack;


typedef struct A49  {

        VtTrackHeader hdr;     // Track Header
        VtTrackStatus trkstat;// Track status
        VtTracker state;       // Tracker state at time of last report
        VtReport rpt;                 // Last Land report
        VtLandData data;       // Land attributes

} VtLandTrack, VtGroundTrack,VtGTrack;


typedef struct A50  {

        VtTrackHeader hdr;     // Track Header
        VtTrackStatus trkstat;// Track status
        VtTracker state;       // Tracker state at time of last report
        VtReport rpt;                 // Last Land report
        VtUnitData data;       // Land attributes

} VtUnitTrack, VtUTrack;

//==============================================================
//==============================================================
```

```
typedef struct  A51 {

        VtTrackHeader hdr;      // Track Header
        VtTrackStatus trkstat;// Track status
        VtTracker state;        // Tracker state at time of last report
        VtReport rpt;                   // Last Associated report
        VtSIReport sirpt;       // SI Data from last report
        VtSIData data;                  // single source track
} VtSITrack, VtCTrack;

//=========================================================
//=========================================================

typedef struct A52 {

        VtTrackHeader hdr;      // Common Track Header
        VtTrackStatus trkstat;// Common Track status
        VtTracker state;        // Common Tracker state at time of last rpt
        VtReport rpt;                   // Common Last report
        VtMinExtData data;      // Mostly Common data
        char pad[554];  // VT_MAX_TRACK_SIZE -
                        // sizeof(VtTrackHeader) -
                        // sizeof(VtTrackStatus) -
                        // sizeof(VtTracker) -
                        // sizeof(VtReport) -
                        // sizeof(VtMinExtData)];// Remaining space to be defined seperately

} VtExtTdbmTrack, VtXTrack;


typedef struct A53 {
        char pad[VT_MAX_TRACK_SIZE];
} VtGenericTrack;


union VtTrack switch (long) {

        case 1: VtGenericTrackntrk;
        case 2: VtTrackHeaderhdr;
        case 3: VtPlatformTrackptrk;
        case 4: VtEmitterTracketrk;
        case 5: VtAcousticTrackatrk;
        case 6: VtSpa25Trackstrk;
        case 7: VtLateralTellTrackttrk;
        case 8: VtRaycasVTrackrtrk;
        case 9: VtUnitTrack    utrk;
        case 10: VtLandTrack   gtrk;
        case 11: VtSITrack     ctrk;
        case 12: VtLinkTrack   ltrk;
        case 13: VtFCSTrack    ftrk;
        case 14: VtExtTdbmTrackxtrk;

} ;


// VtTrack union types
```

```
#define VtNoTrackType0

#define VtPlatformTrackType1
#define VtEmitterTrackType2
#define VtAcousticTrackType3
#define VtSpa25TrackType4
#define VtLateralTellTrackType5
#define VtRaycasVTrackType6
#define VtUnitTrackType7
#define VtLandTrackType8
#define VtFCSTrackType 8
#define VtSITrackType  9
#define VtLinkTrackType10
#define VtExternalTrackType11

#define VtNTrkVtNoTrackType

#define VtPTrkVtPlatformTrackType
#define VtETrkVtEmitterTrackType
#define VtATrkVtAcousticTrackType
#define VtSTrkVtSpa25TrackType
#define VtTTrkVtLateralTellTrackType
#define VtRTrkVtRaycasVTrackType
#define VtUTrkVtUnitTrackType
#define VtGTrkVtLandTrackType
#define VtFTrkVtFCSTrackType
#define VtCTrkVtSITrackType
#define VtLTrkVtLinkTrackType
#define VtXTrkVtExternalTrackType

#define MAX_TYPE_INDICATORS   12

interface Trax2 {

void print ( in long signal );
void get ( in long index, out VtPlatformTrack track, out double time );

};
```

C.4 PROCEDURE CALL TIMING

```
// Client.C

#include <stream.h>

// for extern "exit"
#include <stdlib.h>

// for timing
#include <sys/time.h>
double xtime=1;
double gtime() {
   return xtime;
}

int main (int argc, char **argv) {
```

```
   long value = 0;
   hrtime_t start_hires, end_hires, diff_hires, Svr_time[15000];
   double time;


        start_hires = gethrtime();
        for(value=0;value<1000000000;value++) {
            diff_hires = gtime();
        };
end_hires = gethrtime();
        cout << setprecision(16) << start_hires << " to " << end_hires
<< endl;
}
```

C.5 NULL PROCEDURE TIMING

```
// Client.C

#include <stream.h>

// for extern "exit"
#include <stdlib.h>

// for timing
#include <sys/time.h>
double xtime=1;
double gtime() {
   return xtime;
}
void gx() { }


int main (int argc, char **argv) {
   long value = 0;
   hrtime_t start_hires, end_hires, diff_hires, Svr_time[15000];
   double time;


        start_hires = gethrtime();
        for(value=0;value<1000000000;value++) {
            gx();
        };
end_hires = gethrtime();
        cout << setprecision(16) << start_hires << " to " << end_hires
<< endl;
}
```

C.6 NULL LOOP TIMING

```
// Client.C

#include <stream.h>

// for extern "exit"
#include <stdlib.h>
```

```
// for timing
#include <sys/time.h>
double xtime=1;
double gtime() {
   return xtime;
}

int main (int argc, char **argv) {
   long value = 0;
   hrtime_t start_hires, end_hires, diff_hires, Svr_time[15000];
   double time;


        start_hires = gethrtime();
        for(value=0;value<1000000000;value++) {
        };
end_hires = gethrtime();
        cout << setprecision(16) << start_hires << " to " << end_hires
<< endl;
}
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1997	Final

4. TITLE AND SUBTITLE
Predicting CORBA Performance Through Prototyping

5. FUNDING NUMBERS
DASW01-94-C-0054
Task Order T-S5-1446

6. AUTHOR(S)
Edward A. Feustel, Clyde G. Roby

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Defense Analyses (IDA)
1801 N. Beauregard St.
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER
IDA Paper P-3327

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Information Systems Agency
Center for Computer Systems Engineering
5600 Columbia Pike
Falls Church, VA 22041-2717

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; unlimited distribution: 18 September 1997.

12b. DISTRIBUTION CODE
2A

13. ABSTRACT (Maximum 200 words)

This paper describes experiments that show how the results of simple measurements can be used to design complex distributed applications. The experiments used IONA Orbix, an Object Request Broker (ORB) that is Common Object Request Broker Architecture (CORBA) compliant. The experiments were conducted on Sun Sparc 20s and Intel Pentium 90s using the Microsoft NT 4.0 operating system. The purpose of the experiments was to obtain information about resource expenditures needed to support distributed computing and to use that information to support development methodologies for distributed applications. The paper shows why a simple division of a replacement for the Global Command and Control System's Track Correlation application into a specific Client and Server has little chance of success. A worked-out example experiment, using C++, and outlines of similar experiments which should be performed prior to the development of any distributed applications are also provided.

14. SUBJECT TERMS
Distributed Applications; Design; Timing Models; CORBA; Object Request Broker; DCE, ORBIX, GCCS, Track Correlation.

15. NUMBER OF PAGES
114

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102