The Lightweight Software Bus : A Message Broker for Prototyping Problem Solving Environments [*]

Rajini I. Balay, Harry Perros and Mladen A. Vouk

TR-97-06

April 7, 1997

Abstract

This report describes the Lightweight Software Bus (LSB) developed at NC State to support prototyping of Problem Solving Environments (PSEs). LSB is a portable IP-based wide-area message broker that currently operates on a variety of Unix based platforms and will soon be available on Windows 95. LSB architecture and design are presented alongside its performance profile. Two performance metrics are used: message throughput and message latency. Both the latency and the throughput of LSB were measured in a high performance ATM-based testbed. LSB performance was compared with that of plain Berkeley sockets and with the IBM SOMobjects. The results show that LSB overhead is low and that, for the functionality examined, both its throughput and its latency are significantly better than that of the SOMobjects. Simple interface, together with a low latency and high throughput, make LSB a good choice for building PSE prototypes.

Department of Computer Science
North Carolina State University
Raleigh, NC 27695

[*]This work was supported in part by MCNC/EPA grant C94-7050-027/029, NSF award ACS-9696131, IBM SUR grants and the CACC award 5-3007

1 Introduction

In this paper, we describe the architecture and performance of the Lightweight Software Bus (LSB). LSB is a message broker developed to support prototyping of Problem Solving Environments (PSE). A modern PSE is envisioned as a collection of programs that cooperate to accomplish overall system goals [4]. An example is a PSE that helps an environmental scientist or regulator pose environmental engineering questions (problems), develop, execute and validate solutions, analyze results, and arrive at a decision (e.g., cost-effective emission control strategy) [12]. Such a PSE would consist of a management, analysis and computational framework which utilizes a variety of models and data that describe the science behind the phenomena, the solutions of interest, and decision rules. It is usually assumed that a modern PSE is distributed across a number of central processing units that may or may not reside in one physical computer. An important part of the PSE framework is its ability to facilitate effective and efficient communication among the PSE components (or objects). In addition to the request-reply paradigm, a PSE framework should provide an event notification mechanism for collaborative applications.

Various communication systems such PVM [5], Isis [2] etc., have been developed that can support PSEs. Also, distributed object frameworks like the Common Object Request Broker Architecture (CORBA) [9] have been proposed for developing reliable and flexible applications. Although comprehensive, these systems are often cumbersome to use, they require a considerable learning curve, and they may be expensive to acquire. When building prototypes of PSEs, it may be more cost-effective to use a lightweight rather than a general-purpose (heavyweight) communication framework. Our solution, LSB, was designed specifically for flexible, user-friendly and low-overhead support of PSE prototyping efforts.

LSB is *lightweight* in that (a) it provides minimal but adequate functionality for PSE-level interprocess messaging, (b) its performance overhead is low, and (c) the client "interface" overhead (amount of access code) is low. So far, we have used the system to build two operational PSE framework prototypes. One framework supports an Environmental Decision Support System (EDSS) [1, 15] and the other one supports an educational and training system [16]. We are in the process of adding Quality of Service and Web/HTTP interfacing functionalities to the LSB library.

The paper is organized as follows. Section 2 discusses the LSB features. Section 3 provides an overview of IBM's SOMobjects. In section 4, we report on an experimental study that compares LSB with plain Berkeley sockets and SOMobjects. We show that for data transfer functions, LSB performance overhead is very low and that it appears to perform far better than SOMobjects. Conclusions are in section 5.

2 Lightweight Software Bus

LSB is a set of client/server communication and message management shims and routines written in C. It currently runs on a variety of Unix platforms including DEC Ultrix, DEC OSF1, SUN Solaris 5.3, SGI IRIX5.2, HP UXA9, AIX 4.2 and Cray Y-MP. The library provides an easy way of gluing PSE components (clients) to the LSB message-brokering server (BusMaster).

2.1 General LSB Architecture

LSB consists of a central server called the *BusMaster* that brokers messages between processes in the system. Cooperating PSE subsystems (clients) communicate with each other by sending messages via the LSB. The LSB client library is a layer of communication functions that resides between the client application and TCP sockets. These functions allow clients to communicate with each other and with the BusMaster without dealing with the socket layer communication details. The PSE clients can be located on the machine that hosts the BusMaster, or they can be on other machine as long as they are interconnected by TCP/IP. LSB allows easy integration of both X-based and non X-based clients into the system. As a result, unlike applications built using systems like PVM [5], Isis [2] etc., a client process can receive messages as well as user input at the same time.

The BusMaster intercepts messages sent by a client and forwards them to the appropriate destination. It also answers queries from the clients regarding system status, existence of a certain client, or about the message types registered in the system. For example, a client can issue a "BusFunction" call, to determine the *client identifier* of the destination client. *BusFunction* calls made by a client are *synchronous.* That is, a client is blocked while waiting for a reply from the BusMaster In contrast, message transfer requests between clients are asynchronous, i.e, the sending client is not blocked after it sends a message.

Communication systems whose architecture is similar to that of LSB are the *Broadcast Message Server(BMS)* in the HP Softbench environment [3], the *Multicast Messaging System (MCMS)* in the DEC FUSE environment [6], which have been greatly influenced by the *Msg* system used for messaging in the FIELD environment [10]. They all have a central server which distributes messages and they all provide support for X and Motif applications and they are all well-suited for applications that use 'request/ notification' type of services. In addition to the above capabilities, LSB allows clients to communicate directly with other clients, and also with other processes which are not part of the application.

2.2 LSB Message Interface

The LSB client message interface is specified by means of unique *message types* and possibly non-unique *callback functions*. At the message interface, a client registers with the BusMaster the types of messages that it is willing to accept and process. For each message type that the client accepts, it registers a callback function, which is invoked automatically when a message of that type arrives at the client message interface. For example, let a system consist of four clients C1, C2, C3 and C4 as

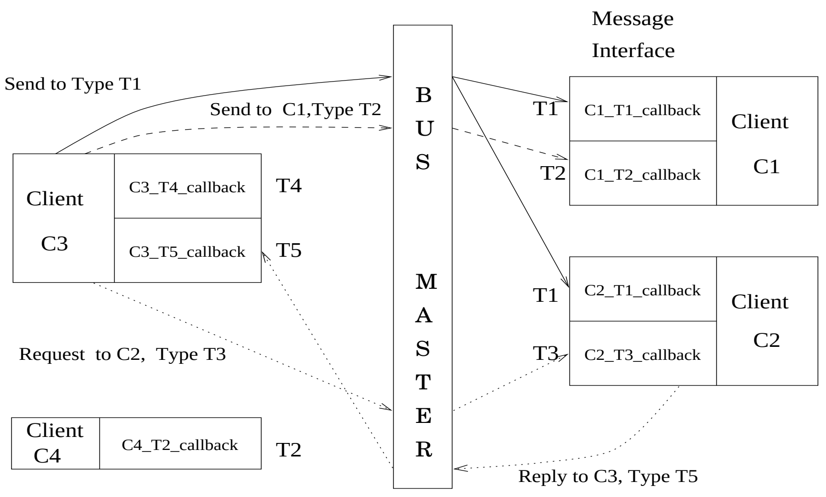


Figure 1: Message Path through the LSB

shown in figure 1. The message interface for client C1 has entry points for message types T1 and T2, while the entry points for clients C2, C3 and C4 are (T1, T3), (T4, T5) and (T2) respectively. Each of the message types has a callback routine. The callback routines C1_T1_callback and C1_T2_callback are registered for the message types T1 and T2 respectively at client C1. Similar callback routines are defined for the other clients as illustrated in figure 1

Communication between LSB clients is primarily *asynchronous*. That is, the sending client is not blocked after it sends a message to another client. Messages trigger functions advertised by a client. Various flavors of communication, *point-to-point*, *multicast* and *request-reply*, are possible in LSB.

A client can send a *point-to-point* message specifically to another client. For example, Client C3 in figure 1 can send a message of a specific type, T2, to client C1. The BusMaster delivers the message to client C1 and the callback function C1_T2_callback is invoked to process the message.

A client can send a *selective multicast* message by requesting the Bus Master to deliver a message based on its type. The Bus Master forwards the message to only clients that understand the type. For example, the message sent by client C3 to type T1 (denoted by the solid line in figure 1) is delivered to the two clients that understand that message type, but not to C4 that does not. However, even though the same message is received by both clients, the callback function for each client may be different, so that the message may be interpreted by each client in a different fashion.

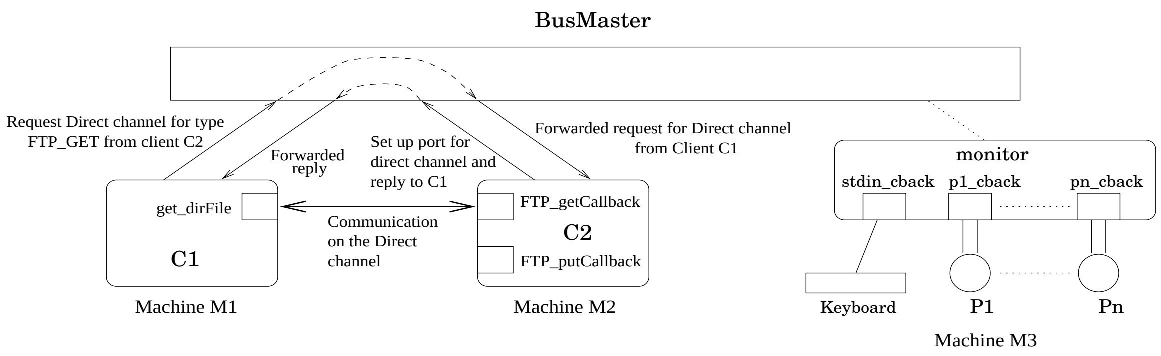
Request-reply type of communication may be achieved in two different ways. In the first method, a *sender* that wants a reply for a request message, registers a callback for the reply message type. The *receiver*, on obtaining the request, sends the reply in a message of the reply type, which is then processed by the *sender*'s callback function. This process is illustrated in figure 1 by the dotted line where, client C3 is the sender and client C2 is the receiver. Another method for achieving request-reply communication is to use a direct communication channel, which is explained in the following subsection.

2.3 Other LSB Client Interfaces

In addition to the *message interface*, LSB library provides the interfaces described below. QoS and Web/HTTP interfaces will be added to the LSB library.

2.3.1 Direct channel interface

Clients can request the BusMaster to establish a *direct channel* with another client. This type of communication can be used for large file transfers or for synchronous communication between clients. Input is again classified into *types*, with appropriate callback functions registered for each type. Figure



BusMaster
Request Direct channel for type FTP_GET from client C2
Forwarded reply
Set up port for direct channel and reply to C1
Forwarded request for Direct channel from Client C1
get_dirFile
C1
Communication on the Direct channel
FTP_getCallback
C2
FTP_putCallback
Machine M1
Machine M2
monitor
stdin_cback
p1_cback
pn_cback
Keyboard
P1
Pn
Machine M3

Figure 2: Communication through the Direct channel and the Socket interfaces in LSB

2 illustrates the operation of this interface. The two clients, C1 and C2 run on machines M1 and M2, respectively. Client C2 has registered two types FTP_GET and FTP_PUT for its direct communication interface. Their callback functions FTP_getCallback and FTP_putCallback transfer files from and to the host, M2. For example, when C1 wants to obtain a file from M2, it sends a request to the BusMaster to set up a direct communication channel with client C2 of type FTP_GET. The BusMaster forwards

the request to client C2, which then checks if it has registered a type FTP_GET at its direct channel interface. If it has, it replies to C1 with the port number to which C1 can connect to. This reply is forwarded by the BusMaster to C1. On receiving the reply, the LSB library connects to the port that C2 is listening to, thereby establishing the direct channel. Once the channel is established, FTP_getCallback is invoked at C2 to transfer the requested file to C1. A corresponding function at C1, *get_dirFile* (which is specified when a request for the direct channel is made), receives the file arriving on the direct channel and processes it.

2.3.2 Socket interface

The socket interface allows clients in an LSB system to communicate with other processes that are not connected to the BusMaster. Using the socket interface, a client can register a callback function for a socket, which is invoked to process the data that arrives at that socket. For example, consider a client, a *monitor*, which starts processes and oversees processes on a particular machine, M3, as shown in figure 2. The monitor, starts process *Pi* as its child process and monitors it by establishing a UNIX *pipe* between itself and the process. Through the pipe, the monitor can receive the output generated by the process, its error messages and its completion status. For each child process *i*, the monitor registers a callback function, *pi_cback*. Output from the child process is handled by the callback function. The socket interface is also useful for applications that read from the standard input (*stdin*). A callback registered for *stdin*, *stdin_cback*, as shown in figure 2, will process user input from the keyboard. This allows the application to receive commands from *stdin* as well as from other clients.

The message interface, together with the direct channel and socket interfaces, provides a comprehensive functionality for the clients in the system that wish to communicate with each other, as well as processes outside the system in either a synchronous or an asynchronous manner.

2.4 Application Development

Developing a PSE application using LSB involves identification of the PSE subsystems, identification of the services that each subsystem performs, design of message types and callback functions for the advertised services, and use of these services by other components by issuing appropriate messages. To incorporate a subsystem into LSB, an initialization function call has to be made. It connects the subsystem to the BusMaster and initializes its data structures. Then, for each message type that the module wants to process, a LSB library call is made to register the message type. A callback function for the message type should be defined. Now, the module is integrated into the system and can receive messages from other modules. An existing subsystem can also be added very easily into an application through the LSB library with minor modifications to its code. Specifically, an extra function call to connect to the BusMaster, and callback functions to process messages would have to be added.

3 SOMobjects

The SOMobjects toolkit [8, 7] is the IBM implementation of the CORBA standard. SOM stands for the System Object Model. It provides the ability to create frameworks composed of cooperating distributed objects. Clients can invoke methods on objects residing in any process, on any machine. The distribution mechanism is kept hidden. In this section, we present an overview of the CORBA architecture and the features of CORBA implemented in SOMobjects.

The Common Object Request Broker Architecture (CORBA) was proposed by the Object Management Group (OMG) as a standard specification for distributed object computing [9, 14]. CORBA makes possible the reuse of software through distributed object computing, which combines the concepts of distributed computing with object-oriented computing.

The primary component in CORBA is the *ORB core*, which transparently transfers a request from a client to an object and then transfers the results of the request from the object to the client. It hides the object location, object implementation, object execution state and object communication mechanisms from the client program that invokes an object's method. An object defines its interface, which specifies the operations and types it supports, through the *Interface Definition Language (IDL).* An object can define its method to be a *oneway* operation in which case the process invoking the method does not have to wait for the completion of the method before proceeding with its processing. The *Language Mappings* provide mappings between the OMG data types defined in the IDL to a programming language. An IDL compiler is used to compile the object interface into *client stubs* and *server skeletons*. These mechanisms perform the parameter marshalling and unmarshalling at the client and server respectively. The client stub is used by the client application for transferring the request and the server skeleton is used by the object implementation. An *Interface Repository* (IR) maintains a databank of the available IDL types in the system. It allows the IDL type system to be accessed and written programmatically at runtime. The other component of CORBA is the *object adaptor*, which serves as a glue between the object implementations and the ORB itself.

In addition to the Static Invocation Interface (SII) where the client stubs and server skeletons are used, CORBA also supports a Dynamic Invocation Interface (DII) and a Dynamic Skeleton Interface (DSI). They can be viewed as a generic stub and generic skeleton respectively. Using DII, a client application can invoke requests on any object without having compile time knowledge of the object's interface. DII may cause the ORB to transparently access the IR to obtain information about the types, arguments and return values of a request. Since the IR is itself an object, each invocation of an IR method could potentially be a remote invocation, making the DII an expensive operation. CORBA supports the operation *send* and *invoke* to issue asynchronous and synchronous DII requests. DSI was introduced in CORBA 2.0. It allows servers to be written without having the object skeleton compiled statically into the program. CORBA 2.0 introduced a general interoperability architecture that provides for direct ORB-to-ORB and bridge based interoperability.

SOMobjects Version 2.1 implements CORBA 1.1 while SOMobjects Version 3.0 implements CORBA 2.0. They provide language mappings for C and C++. Objects in SOMobjects can reside in the same process as the client, or in a different process. Objects residing in a different process are managed by a server process, which receives method invocations directed to the object(s) that it manages and returns the results back to the client. SOMobjects uses a daemon process called *somdd*, to manage server processes on a particular machine. When a client invokes a method on a remote object, the client ORB communicates with the *somdd* process (listening on a "well known" port) at the remote host to determine the port that the server for the particular object is listening to. The client directs its communication required for various method invocations to this port. SOMobjects uses UDP for communication between clients, object servers and the *somdd* processes.

4 Performance Measurements

This section compares performance of LSB with a) plain BSD sockets and b) SOMobjects 2.1. Two measures of performance were used : *throughput* and *latency*. Throughput is the amount of data that can be transmitted through the system per unit time. We measured the user-level throughput which is the number of bytes of data that a user application transmits or receives divided by the time it takes to send/receive the data. Latency is the delay incurred in transferring a message through the system. The measurements were made using RS6000 3CT and 3BT platforms equipped with IBM Turboways ATM 155 adaptors and interconnected by IBM 8260 ATM switches. The operating system was AIX 4.2 and transmissions over the ATM links were made using "Classical IP" over ATM.

4.1 Throughput measurements

Since both LSB and SOMobjects reside on top of the TCP/IP and UDP/IP layers, we first measured the throughput profile of these layers using TTCP [13, 11]. We did that in order to clearly distinguish performance characteristics of LSB and SOMobjects from the underlying TCP/IP and UDP/IP implementations and of the "Classical IP" layer.

4.1.1 LSB versus TTCP

Throughput of LSB was compared with that of BSD sockets by performing experiments of type described in figure 3. TTCP [13] measures the transmitter-side and receiver-side throughput, by measuring the delay for transmitting/receiving a given block of data. The parameters that can be controlled include the payload size (b), and the number of times the buffer is transmitted (n). The total number of bytes to be transferred (M) was set to be 64 MBytes and the payload size, b, was varied. The number

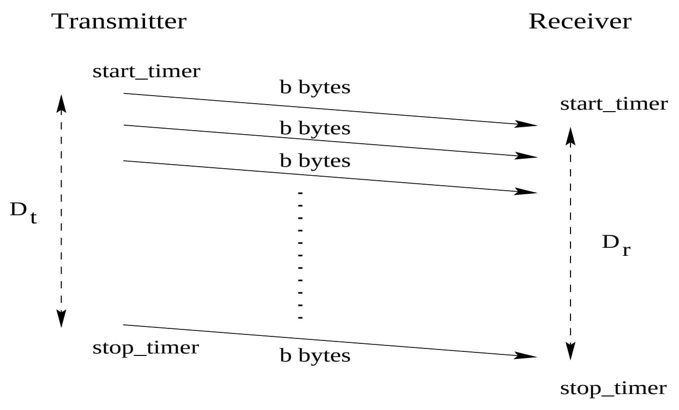


Figure 3: TTCP operation

of times the buffer was transmitted was therefore, $n = M/b$. As shown in figure 3, the transmitter started its timer, transmitted the data, and then stopped its timer. The delay incurred in the data transmission, D_t, was calculated, from which the throughput in $Mbps$ was obtained.

Two LSB clients, a *Transmitter* and a *Receiver*, were used to measure the LSB throughput. The *Transmitter* client, started it timer, sent n messages whose length was b bytes each. After n messages were sent, the delay in transferring the data, was calculated. An experiment was also conducted to measure the throughput over the direct LSB communication channel. Similar to TTCP, after a direct communication channel between the *Transmitter* client and the *Receiver* client was established, the data was transferred from the transmitter to the receiver, as a series of payload buffers.

4.1.2 LSB versus SOMobjects and TTCP versus SOMobjects

Since messages in LSB are asynchronous, an experiment with SOMobjects that would transmit messages in a similar fashion as LSB, required using the SOM *oneway* method call. The *oneway* method calls work similar to an asynchronous message, i.e., the process making the invocation does not wait for a reply. The flow control mechanism in SOMobjects [1] did not allow use of *oneway* method calls to measure SOM throughput for bulk data transfers. We therefore devised another experiment that measured the throughput of the system. It is illustrated in figure 4. The server process implemented an object that supported the methods `start_timer` and `recv_data`. `start_timer` is a oneway method and `recv_data` is a twoway method. To measure the oneway throughput at the receiver, the client process made method invocations on the server object in the following fashion (part (a) of figure 4).

[1]SOM uses UDP to send its messages. So when the transmitter process flooded the receiver with UDP packets, the packet losses became unacceptable.

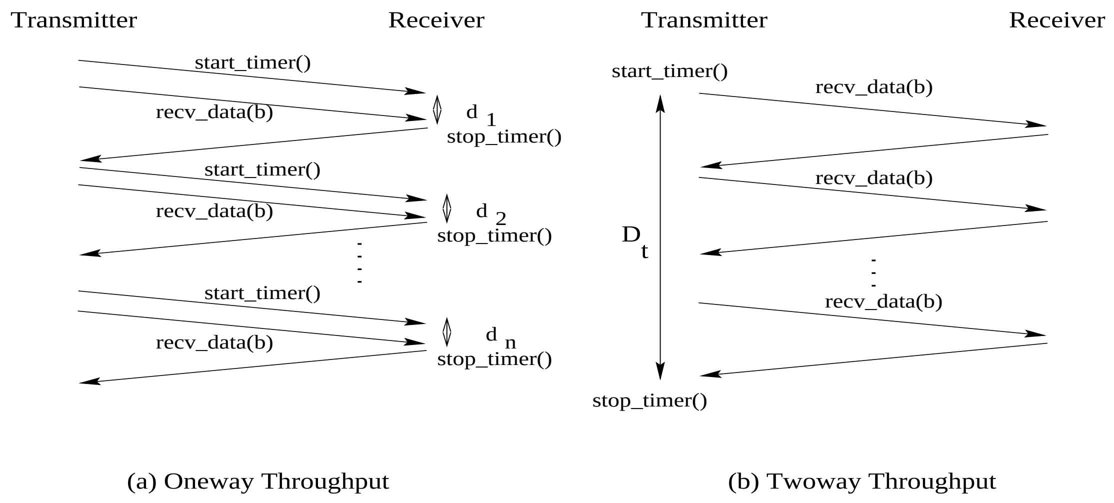


Figure 4: SOMobjects experiment operation

It first invoked the `start_timer` method and then immediately `recv_data`. A buffer of size b bytes was sent as an argument to `recv_data`. The method `recv_data` terminated the timer and measured the delay in receiving b bytes of data, d_i. It then sent a response back to the client process. This operation was repeated n times to effectively transfer M bytes of data. The delay for the oneway experiment, D_o was $\sum_{i=1}^{n} d_i$. The experiment to measure the twoway throughput is described in part (b) of figure 4. Here the transmitter process started its timer, and consecutively invoked the `recv_data` method n times, each time passing as its argument a buffer of size b bytes. The total delay for the twoway operation, D_2 was obtained and used to calculate the twoway throughput. This experiment was conducted with both the static invocation interface (SII) and the dynamic invocation interface (DII) of SOM.

Equivalent experiments were designed for both LSB and TTCP to measure the oneway and twoway throughput using the stop-and-go method of operation described above. These experiments are referred to as the *paced* oneway and twoway experiments.

4.2 Latency measurements

Messages in LSB are untyped and therefore a message transfer requires only marshalling the message header and not the message body. The software overhead in LSB is therefore due to the message header manipulation and the extra hop at the BusMaster. Messages in SOMobjects are typed and therefore a message transfer incurs parameter marshalling/unmarshalling overhead. In view of this, we only considered the latency of null messages. This delay reflects the overhead involved in processing the message header during message transmission.

The latency measurements were conducted for null *oneway* asynchronous communication and *twoway* communication using bi-directional asynchronous messages in LSB and synchronous communication in SOMobjects. Since messages in LSB are asynchronous, the oneway latency was obtained by sending a number of consecutive messages and measuring the average message transmission delay. To obtain the twoway latency, the *request/reply* scenario described in Section 2.2 was used. The number of messages that were sent consecutively was 10,000.

The oneway and twoway latencies for SOMobjects SII and DII were measured. For SII, the oneway latency for SOMobjects was measured by invoking a method that was defined with a *oneway* method interface. A normal method invocation was used for the twoway latency. For the dynamic invocation interface, the *send* and *invoke* operations were used for the oneway and twoway measurements respectively. Due to the previously mentioned flow control problem encountered with SOMobjects, we were unable to make more than 75 successive oneway invocations. The measurements for the twoway latency were performed for 10,000 successive method invocations.

4.3 Parameters

We focused on the following parameters in the experiments:

1. Socket queue size: The socket queue size significantly affects the TCP-level performance over a high-speed network. We set the sender and receiver socket queue size to 64 KBytes [11].

2. Data buffer size: For the throughput measurements, the data buffer size was varied from 1 Kbytes to 256 Kbytes in powers of two.

3. TCP_NODELAY : The TCP_NODELAY option was set for the TCP sockets in the experiments for TTCP and LSB used for comparison with SOMobjects. This was done to avoid the TCP buffering delay which comes into play due in the stop-and-go mode of operation. This option was also used for measuring the twoway latency of LSB.

4. BusMaster Location : The BusMaster process ran on the same machine as the Transmitter process for the LSB experiments.

Each experiment was run ten times, and the average delay values are reported here.

4.4 Results

4.4.1 Throughput

Figure 5 presents the results of the throughput measurements for TTCP and LSB. The actual through-

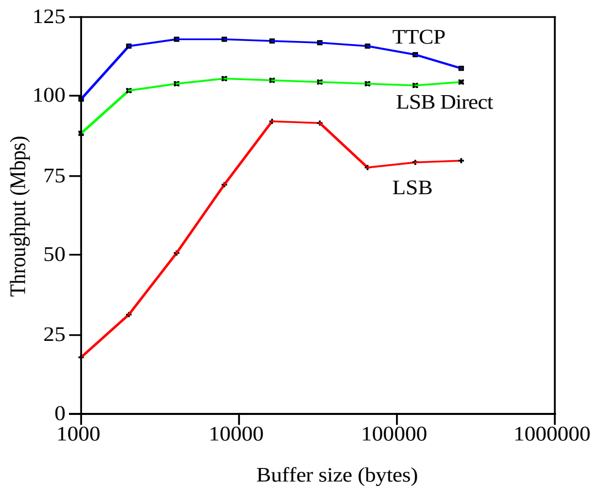
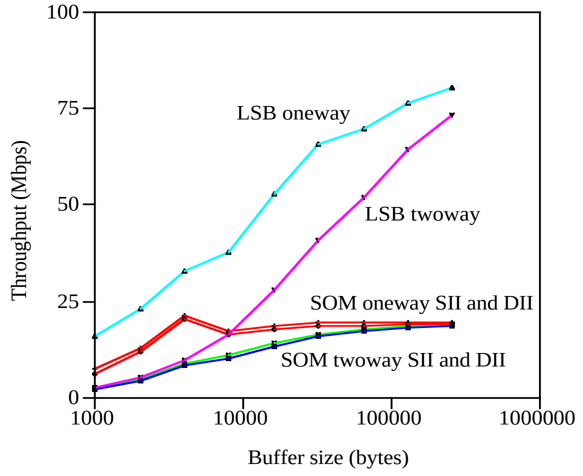


Figure 5: LSB versus TTCP

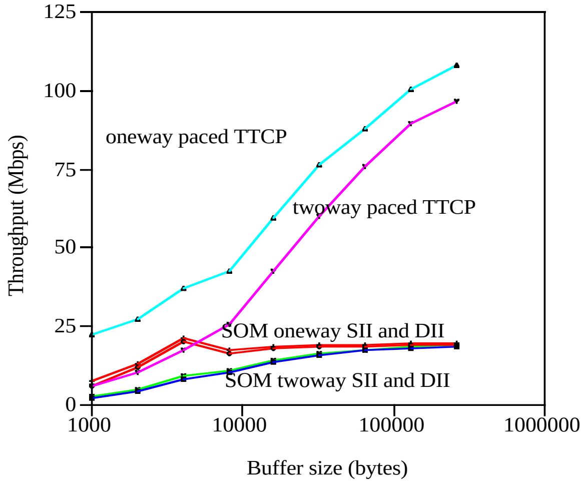
put measurements are in *Mbps* for varying buffers sizes *b*. We observe that, on a 155 Mbps ATM network, TTCP could obtain a peak throughput of 118 Mbps for buffer sizes of 4 Kbytes and 8 Kbytes. For larger buffer sizes, TCP's throughput tends to decrease. LSB achieves a throughput of about 92 Mbps for buffer sizes of 16 Kbytes and 32 Kbytes, after which it reduces to 77.6 Mbps at 64 KBytes, which is the socket queue size. The LSB direct communication channel gives throughput very close to that of TTCP. The differences in the throughput between TTCP and the LSB direct communication is due to the connection establishment delay and the difference in the implementation of data transfer in TTCP and the LSB direct channel. LSB obtains about 20% of the throughput of TTCP for a buffer size of 1 KByte, but obtains 78% of the throughput of TTCP for buffer sizes of 16 Kbytes and 32 Kbytes. In contrast, the direct communication channel consistently obtains about 90% of the throughput of TTCP.

A comparison of the SOMobjects throughput with the throughput obtained using LSB and TTCP is shown in figure 6. As can be seen in both figures 6(a) and 6(b), the SII and DII throughput for SOM oneway and twoway experiments are very similar. For the oneway experiment, SOMobjects obtains a throughput of 8 Mbps for a buffer size of 1 Kbytes. It reaches a maximum of 21.3 Mbps for a buffer size of 4 Kbytes, and drops to 17.65 Mbps for a buffer size of 8 Kbytes. This is probably mainly due to the internal fragmentation by SOMobjects for messages larger than 8 Kbytes. The throughput then increases to 20 Mbps for larger buffer sizes. In the twoway experiment, the throughput for a buffer size of 1 Kbytes was 2.85 Mbps. It increased to 19.7 Mbps for a buffer size of 512 Kbytes. Initial experiments with SOMobjects 3.0 indicate that the twoway throughput for larger buffer sizes is as much as twice the throughput obtained with SOMobjects 2.1.

Part (a) of the figure shows the throughput of SOMobjects and LSB. The throughput of the oneway experiment for LSB is 16.4 Mbps for a buffer size of 1 Kbyte. The throughput steadily increases to 80.5



(a) SOMObjects versus LSB



(b) SOMObjects versus paced TTCP

Figure 6: Comparison of SOMobjects throughput with LSB and TTCP

Mbps for a buffer size of 256 Kbytes. For the twoway experiment, the throughput is 3.1 Mbps for 1 Kbyte buffer size, and it increases to 78 Mbps for a buffer size of 256 Kbytes. The oneway SOMobjects experiment achieves upto 65% of LSB throughput for a buffer size of 4 Kbytes, which reduces to 25% for larger buffer sizes. The twoway SOMobjects experiment, in contrast obtains about 91% of the throughput of the LSB twoway experiment for buffer sizes of 1 Kbytes to 4 Kbytes. But for larger buffer sizes, LSB performs better, with SOMobjects obtaining only about 25% of the throughput.

The paced version of oneway TTCP achieves a throughput of 23 Mbps for a buffer size of 1 Kbytes and increases upto 113 Mbps. In the twoway case, the throughput was 6.5 Mbps for a buffer size of 1Kbytes, and it increased to 95 Mbps for 256 Kbytes. The oneway SOMobjects experiment achieved 35% of the throughput for a 1 Kbyte buffer size, and 57% for 4 Kbytes. As the buffer size increased, the throughput reduced to 17.7% as that of TTCP. The twoway experiment obtains 44% of the throughput of TTCP for 1 Kbyte buffers, and peaks to 52% for a buffer size of 8 Kbytes. Due to the internal fragmentation of SOMobjects for message sizes greater than 8Kbytes, the performance reduces to about 20% for larger buffer sizes.

Although the SII and DII throughput for the oneway and twoway cases for SOM are very similar, the SII throughput is slightly greater than that of the DII throughput. This is due to the additional overhead involved in locating the remote object in DII. In general, the static invocation is expected to perform much better than the dynamic invocation. However, this is not the case in SOMobjects. This is because, SOMobjects accesses the interface repository during parameter marshalling for static invocation, similar to the processing involved in the dynamic invocation. This extra overhead could be reduced by creating intelligent stubs that marshal the parameters instead of using the interface repository.

4.4.2 Latency

The results of the latency measurements are summarized in the following table.

	LSB	SOM SII	SOM DII
Oneway Latency	0.203 ms	0.326 ms	2.314 ms
Twoway Latency	1.827 ms	2.59 ms	2.668 ms

The latency measurements show that LSB has a lower message manipulation overhead than SOMobjects, for both oneway and twoway message transmission. The latency for a oneway message transmission for LSB was $0.203ms$ whereas the oneway SII latency for SOMobjects was $0.326ms$, which is 1.6 times the LSB latency. The twoway latency for LSB was $1.827ms$, as compared to $2.59ms$ for SOMobjects, which is about 1.4 times the LSB latency. LSB has lower latency even though the twoway latency involves extra hops at the BusMaster, whereas a SOMobjects client communicates directly with the object server. The twoway latency for SII and DII in SOMobjects are very similar. This indicates that the same amount of processing occurs for both types of invocations at the client process. The oneway DII latency is slightly lower than the twoway DII latency as the client does not wait to receive the result of the method invocation.

5 Conclusions

We have described a message broker that provides a simple and low-overhead communication mechanism for aiding in prototyping problem solving environments. We presented a comparison of the performance of LSB with TTCP and SOMobjects. The results show that LSB has significantly lower latency than SOMobjects for null message transmission. LSB could obtain up to 77% of the throughput of TTCP and the throughput offered by LSB was far better than that of SOMobjects. LSB's latency was approximately 0.6 times that of SOMobjects and its throughput was greater than 1.5 times that of SOMobjects, for oneway experiments. For twoway messages, LSB latency was 0.7 times that of SOMobjects and its throughput was similar to that of SOMobjects for small buffer sizes, but increased to 4 times the throughput of SOMobjects for larger buffer sizes.

References

[1] J. Ambrosiano, R. Balay, C. Coats, A. Eyth, S. Fine, D. Hils, T. Smith, S. Thorpe, T. Turner, and M. Vouk. The Environmental Decision Support System: Air Quality Modeling and Beyond. In *Proceedings of the U.S. EPA Next Generation Environmental Modeling Computational Methods (NGEMCOM) Workshop , Bay City, Michigan*, 1995.

[2] K. P. Birman, A. E. Abbadi, W. Dietrich, T. Joseph, and T. Raeuchle. An Overview of the Isis Project. Technical Report TR 84-642, Department of Computer Science, Cornell University, 1984.

[3] M. R. Cagan. The HP SoftBench Environment: An Architecture for a New Generation of Software Tools. *Hewlett-Packard Journal*, pages pp. 36–47, June 1990.

[4] E. Gallopoulos, E. Houstis, and J. R. Rice. Computer as Thinker/Doer: Problem-Solving Environments for Computational Science. In *Problem-Solving Environments*, pages pp. 11–23. IEEE Computational Science & Engineering, Summer 1994.

[5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing.* The MIT Press, 1994.

[6] R. O. Hart and G. Lupton. DEC FUSE: Building a Graphical Software Development Environment from UNIX Tools. *Digital Technical Journal*, 7(2):pp. 5–19, 1995.

[7] IBM. *SOMobjects Developer Toolkit Users Guide*, version 2.1 edition, October 1994.

[8] C. Lau. *Object-Oriented Programming Using SOM and DSOM.* VNR Computer Library, 1994.

[9] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, revision 2.0 edition, July 1995.

[10] S. P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, pages pp 57–66, July 1990.

[11] A. Rindos, S. Woolet, D. Cosby, L. Hango, and M. Vouk. Factors Influencing ATM Adapter Throughput. *Multimedia Tools and Applications*, 2(3):253–271, May 1996.

[12] R.L.Dennis, D. Byun, J. Novak, K. Galluppi, C. Coats, and M. Vouk. The Next Generation of Integrated Air Quality Modeling: EPA's Models-3. *Atmospheric Environment*, 3(12):1925–1938, 1996.

[13] USNA. TTCP: a test of TCP and UDP Performance, December 1984.

[14] S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 35(2):46–55, February 1997.

[15] M. Vouk, R. Balay, and J. Ambrosiano. EDSS - An Environment for Large-Scale Numerical Computing and Decision Making. In *International Workshop on Current Directions in Numerical Software and High Performance Computing, Kyoto, Japan*, 1995.

[16] M. A. Vouk and D. L. Bitzer. Regional Training Center for Parallel Processing. http://renoir.csc.ncsu.edu/RTCPP, 1995.