

DTIC FILE COPY

2

RADC-TR-90-185
In-House Report
July 1990



AD-A226 555

DISTRIBUTED SYSTEM EVALUATION

Vaughn T. Combs, Patrick M. Hurley, Charles B. Schultz, Anthony M. Newton

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.



Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

90 09 13 090

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC TR-90-185 has been reviewed and is approved for publication.

APPROVED:



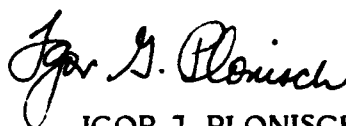
RONALD S. RAPOSO
Chief, C² Systems Technology Division
Directorate of Command and Control

APPROVED:



RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command and Control

FOR THE COMMANDER:



IGOR J. PLONISCH
Directorate of Plans and Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441-5700, This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

REPORT DOCUMENTATION PAGE			Form Approved OPM 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 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 Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503.				
1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE July 1990		3. REPORT TYPE AND DATES COVERED In-House Jul 88 - Feb 90
4. TITLE AND SUBTITLE DISTRIBUTED SYSTEM EVALUATION			5. FUNDING NUMBERS PE - 62702F PR - 5581 TA - 28 WU - 17	
6. AUTHOR(S) Vaughn T. Combs, Patrick M. Hurley, Charles B. Schultz, Anthony M. Newton				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Rome Air Development Center (COTD) Griffiss AFB NY 13441-5700			8. PERFORMING ORGANIZATION REPORT NUMBER RADC-TR-90-185	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Air Development Center (COTD) Griffiss AFB NY 13441-5700			10. SPONSORING/MONITORING AGENCY REPORT NUMBER N/A	
11. SUPPLEMENTARY NOTES RADC Project Engineer: Vaughn T. Combs/COTD/(315) 330-3623				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum: 200 words) This paper describes an RADC in-house distributed systems evaluation project. As a part of the project, a list of attributes were first identified that are necessary and/or desirable in a distributed system. A set of metrics were then designed that would suitably measure distributed system performance for a subset of the attributes identified. The metrics were then implemented using the Cronus Distributed Computing Environment. The results obtained for this implementation are presented.				
14. SUBJECT TERMS Distributed System Evaluation Distributed Systems Distributed Operating Systems			15. NUMBER OF PAGES 68	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT SAR	

Table of Contents

1.0 INTRODUCTION	1
2.0 Distributed System Attributes	3
2.0.1 Concurrent Processing	3
2.0.2 Greater Availability and Survivability	3
2.0.3 IPC Efficiency and Network Transparency	3
2.0.4 Global Resource Management	4
3.0 Distributed Operating System Evaluation	5
3.1 The Hardware and Software Environments	5
3.1.1 The Distributed Environments	5
3.2 Benchmarking Computational Throughput	9
3.2.1 The Proposed Model	9
3.2.2 Implementation Details	11
3.2.3 Discussion of Results	12
3.3 Benchmarking Availability and Survivability	30
3.3.1 The Proposed Model	30
3.3.2 Choice of Data	31
3.3.3 Replication in Cronus	31
3.3.4 Benchmarking Cronus Replication	34
3.3.5 Discussion of Results	37
3.4 Benchmarking Interprocess Communication	40
3.4.1 Results from Benchmarking Interprocess Communication	42
4.0 Overall Remarks and Conclusions	51
Appendix A. Example Type and Manager Definition Files	55
Appendix B. Pseudocode of Benchmarking Procedures (Concurrent Processing)	56



Project Name: ☒	
Date: ☐	
By: ☐	
Title: ☐	
Description:	
For:	
Distribution/	
Priority Codes	
and/or	
Serial	Special
A-1	

Section 1

1.0 Introduction: The growing interest in distributed computing systems can be attributed to the number of benefits offered by these systems. Among these are increased performance, higher availability of resources, and improved fault tolerance and survivability characteristics. These attributes are associated with the distributed environment's ability to use and manage the increasing number of resources provided by its cooperating nodes.

A number of systems have been proposed to support functionality required in distributed operating systems [1,4,5,6,21,23], distributed database and file systems [7,8,9,10,11], distributed management and control [12,13], and distributed artificial intelligence [14,15]. However, with the emergence of diverse heterogeneous computer systems and due to the significant changes which are taking place in communication technology and protocols, it is becoming rather difficult to analyze a distributed environment and to assess its capability for efficiently supporting various applications.

In response to this apparent inadequacy an RADC/COTD in-house working group has been working on a project to identify and characterize the attributes necessary and desirable, in a distributed operating system (DOS). Next a set of metrics were designed and implemented that would suitably measure distributed system performance for a subset of the attributes identified. The metrics were implemented using the Cronus distributed computing environment which was developed for the Department of Defense by Bolt Beranek and Newman (BBN) Laboratories Incorporated. Cronus is currently being used for several in-house research projects as well as in many governmental agencies.

A distributed computing system can be broadly defined as a collection of computers connected by a network that cooperate to complete some computation. This definition, however, describes every class of distributed system imaginable from a distributed database system to a more complex heterogeneous distributed operating system that might necessarily handle complex scheduling and other global resource management tasks. A distributed operating system (DOS), in general, takes on the more classical role of an operating system but does so utilizing a collection of, possibly geographically dispersed, set of resources. In other words the DOS creates and manages logical resources (e.g. application/user processes and files) and physical resources (e.g. processors, memories). There are, however, many classes of distributed operating systems. For example a distributed system may be homogeneous (designed to run on a common hardware base) or heterogeneous (designed to utilize a collection of machines of differing architectures). In a heterogeneous environment the DOS must deal with the potentially burdensome job of translating data to and from some canonical representation when it is passed between machines. A DOS may be a real time or a non-real time DOS. A real time DOS in some way accepts temporal constraints and relationships from a user (application) and makes resource management decisions (e.g. process scheduling) in an attempt to meet these real time constraints. The distributed system may also be classified by the level at which the system is implemented. For example, there is a class of distributed systems that are built on top of an existing local or constituent operating system (e.g. Unix, VMS, etc.). This class of distributed systems are usually referred to as distributed computing environments. While the distributed computing environment provides or at least facilitates distribution of computation, global naming, location transparency, etc., as in a

distributed operating system, resource management, however, is usually handled locally by the underlying constituent operating system (i.e. no global resource management strategies are usually considered). The distributed computing environment considered in this report (Cronus) is such a distributed system. In contrast, there is a class of distributed systems that are built assuming that all operating system functions (subsystems) are implemented by the DOS code itself (i.e. it is a local operating system kernel replacement).

The following report will be organized as follows. In section 2 a brief outline is given of distributed operating system attributes deemed most important by our working group based on performance and function. A complete description of the metrics used to characterize performance with respect to some of the important attributes described is included in section 3. Finally, section 4 contains our concluding remarks.

Section 2

2.0 Distributed System Attributes: In order to effectively quantify the performance of a distributed operating system it is important to identify and understand the system components or attributes that provide desired functionality or utility. A distributed operating system provides a higher degree of functionality not found in most centralized systems. A brief description of this increased functions is provided in the following sub-sections.

2.0.1 Concurrent Processing: In general we would expect that there would be a definite benefit in applying a greater number of computers to a problem. Simply stated, we expect the computational power of one computer to be somewhat less than a collection of computers. If a distributed operating system is designed with aschrony in mind efficient usage of overall system resources can be employed through the use of increased parallelism. However it is not inconceivable that through poor system design (i.e. centralized control approach, inefficient implementation of inter-process communication, etc.) that system performance can actually be degraded in comparison to a single centralized system. In general it will be advantageous to operate in a distributed environment if the added computational power of a node is not overcome by the overhead necessary for the computers to cooperate in the distributed environment. This assessment assumes that our only consideration is for added computational capability and dismisses other attributes for which an application developer may be willing to pay a great deal (i.e. data and process survivability, data availability, etc.). So it is necessary to develop a set of metrics that will give us some insight into how well a particular distributed environment provides us with a means of making use of a collection of resources. One such metric will be discussed in detail later in this report.

2.0.2 Greater Availability and Survivability: Another quality or attribute of a distributed operating system is its ability to make processes and data available in the presence of system (and software) faults and failures. While centralized systems continue to evolve so do their reliability. There is still, however, a non-zero probability of a failure occurring. In a distributed system it is possible to protect data and processes through the judicious use of replication. Many distributed systems provide varying levels of support mechanisms for implementing a number of different replication schemes. It is important to design a set of metrics that will measure how available processes and data are in the presence of faults and failures. It is also very important to quantify the cost of mechanisms provided by the system to support replication (i.e. mechanisms designed to maintain consistency among replicated copies of data, mechanisms to support replicated processes, etc.). One such performance metric is read and write access latency times for replicated copies of data. This metric will be described in detail later in this report.

2.0.3 IPC Efficiency and Network Transparency: Another attribute that distinguishes distributed operating systems from the earlier network operating systems is the primitives designed to hide the implementation of the lower level network from the application designer. This may allow the designer to deal with relatively simple commands (*Send, Receive, Invoke*, etc.) for communication between entities and manipulation of data without requiring

knowledge of where processes and data reside. An efficient distributed operating system relies heavily on efficient *Inter-Process Communication (IPC)*. The complexity of the communication sub-system can vary greatly from system to system. For example in a homogeneous distributed environment there is no need for complex data translations to accommodate for differences in machine architectures while in heterogeneous systems canonical data can be a costly, albeit necessary, overhead of communication. It then becomes necessary to quantify the performance of the system's IPC facility at all levels of communication (i.e. cost of message formation, data translation, routing algorithms, etc.). In short, we must be concerned with the amount of overhead is incurred in passing a message from one entity in the system to another. Metrics used to characterize this aspect of performance are thoroughly described in a subsequent section.

2.0.4 Global Resource Management: Another desirable attribute shared by most distributed systems is the ability to do (or support for doing) global resource management. For some systems this capability may be as robust as an algorithm that does load balancing or global scheduling based on application level specifications of real-time attributes. While the efficiency of such algorithms can be the most important factor in overall distributed system performance it has been deemed outside the scope of our work as Cronus, the distributed computing environment studied, does not perform global scheduling and management of resources automatically at the kernel level.

Cronus relies on the underlying local or constituent operating system for scheduling of its processes. The choice of where to run distributed system services and distributed application processes is left to the application designer and implementor. As a consequence, the complex problem of efficiently balancing CPU, disk, and communications resource usage in the distributed environment must be solved by the application implementor. A more detailed description of the Cronus distributed environment is contained in a subsequent section.

Section 3

3.0 Distributed Operating System Evaluation: In this section we outline the metrics used to quantitatively and qualitatively assess the performance of the Cronus distributed environment with respect to some of the attributes described in section 2. The hardware and software environments used during the evaluation are described in section 3.1. This includes a description of the two distributed environments considered, namely, the Cronus Distributed Computing Environment and Sun Microsystem's implementation of the *Remote Procedure Call (RPC)*. In section 3.2 an application model is described that was used to characterize the distributed system's ability to concurrently process. The results obtained after implementing and running this application in both the Cronus environment and using Sun Microsystem's implementation of the RPC are presented and discussed. In section 3.3 the metrics used to assess the overhead incurred in using the replication mechanisms provided in the Cronus environment is discussed. Again, the results obtained in the Cronus environment are presented and discussed. Finally, in section 3.4 we describe a detailed analysis of overhead incurred throughout all phases of the invocation and response cycle for a standard invocation within Cronus. This data can be used to characterize possibly burdensome subsystems within Cronus with respect to inter-process communication.

3.1 The Hardware and Software Environments: The hardware configuration of the experimental system used for benchmarking consisted of three Sun workstations (two 3/260's and one 3/280). The constituent operating system used was Sun OS version 3.5 (generic kernel with 16 MB main memory). These workstations were connected on a local area network with a bandwidth of 10 Mb/s. and using IEEE 802.3 media access protocol. The whole system was dedicated to this experiment with no other load on it.

3.1.1 The Distributed Environments: This section briefly describes some of the attributes and features of the two distributed environments studied in this portion of the evaluation. The section also elaborates on some of the environment specific implementation issues addressed for each.

3.1.1.1 Cronus Distributed Computing Environment: The distributed computing environment evaluated during this effort is called Cronus and was developed by BBN Laboratories Incorporated. This section contains a brief description of Cronus.

The Cronus distributed environment is based on the object oriented model for distributed computing [21]. Cronus basically consists of services, clients, and the Cronus kernel. A service consists of one or more manager processes that define and manage the objects of one or more types. An object within Cronus is a resource, such as a file, a directory, a mailbox, an inventory, or sensors. Objects are generally considered as passive entities stored on a disk. Object type definitions are organized in a type heirarchy that allows new types to be created as subtypes of existing ones. Services (often referred to as managers) implement both system functions and application functions. Current *system services* provided by Cronus include an authentication service, a symbolic naming service (global), a network configuration service, a distributed file service, and an object type definition service. Clients within Cronus are treated simply as processes that use services.

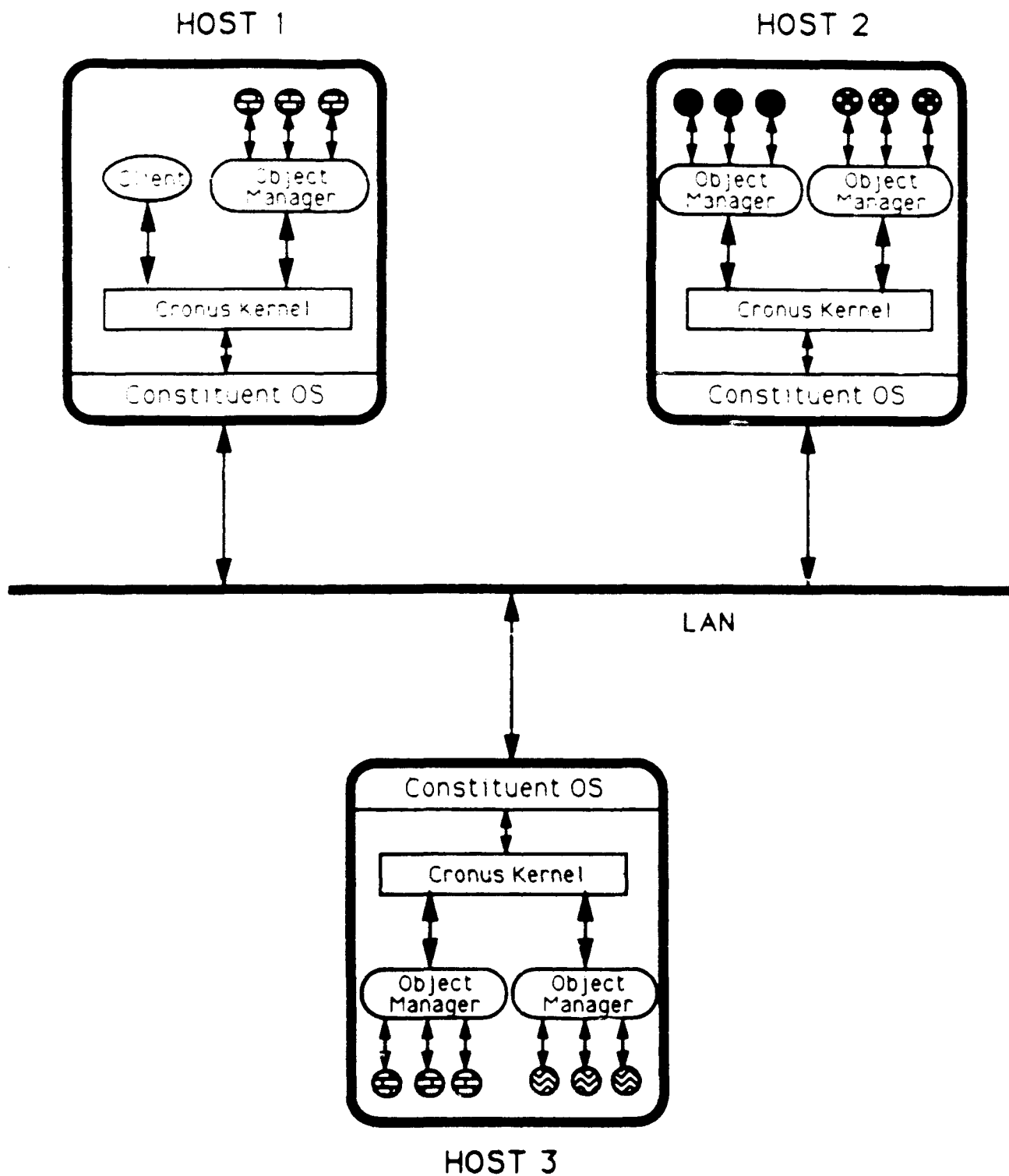


Figure 3.1.1 The Cronus Client-Server Implementation

The main function of the Cronus kernel is to route invocations from the invoking clients to their appropriate servicing Object Manager. The Cronus kernel is implemented as a constituent operating system process (in our case a Unix process) and executes in user space. In other words the Cronus kernel is essentially a locator and an operation switch which helps in identifying the appropriate entity responsible to carry out the computation. The kernel is run on each node in the cluster. Figure 3.1.1 shows the architecture of the Cronus distributed environment[21]. Cronus interprocess communication (IPC) is designed to support operation invocations from clients to services, where the invocations can be synchronous or asynchronous. An invocation can be broadcasted or can be targeted to a single object manager. The IPC is implemented as a series of protocol layers. Cronus relies on a standard communication protocols such as TCP, UDP, and IP [16] which are implemented by the native operating system. Cronus adds three layers on top of the Transport layer of ISO reference model [22].

The lowest layer defined by Cronus is the IPC layer. This layer implements three communication functions: Send, Invoke, and Receive. Invoke is used by a client process to invoke an operation on an object. The invoke is implemented as a message addressed to the object. The message is routed by the Cronus kernel to the process (local or remote) serving as the object's manager. The object manager fetches the message from its message queue using the Receive function in order to perform the requested operation. The operation is performed by a lightweight process created by the object manager to execute the code implemented by the application designer. After completing the operation execution the manager uses the Send primitive to return the reply to the client or application process. The reply message is returned to the client by the Cronus kernel who receives the reply using the Receive function. The separation of the client's Invoke from its later Receive allows asynchrony and concurrency.

Above the IPC layer is the message encoding layer. This layer is responsible for encoding and decoding messages using canonical data representations which are system independent and allow transmission of messages between machines with differing internal representations. Cronus defines canonical data representations for a number of common data types and structures and allows the user to define new canonical types from existing ones [2].

The highest layer implemented by Cronus is the operation protocol (OP) layer. This layer presents the remote procedure call (RPC) interface to the application designer. This layer allows only synchronous or blocking invocation.

It should be noted that Cronus helps with the burden of coding applications in a distributed environment through the use of a nonprocedural program development specification language [3]. The user can provide nonprocedural specifications of a new object type, and operations to be implemented by the manager (an example can be found in Appendix A). Subsequently the code for a *skeleton* object manager can be automatically generated which includes client RPC stubs, data conversion between canonical and system specific data representations, message parsing and validation, operation dispatching, and stable storage management (for persistent objects). The user then completes the object manager by designing and coding the routines that implement the operations defined for the new object type.

3.1.1.2 The Sun RPC Environment: Since RPC is becoming an international standard to support communication among heterogeneous distributed systems, we decided to benchmark the performance of the Sun Microsystems implementation of RPC and compare it with the Cronus Distributed computing environment based on computational throughput (*reference section 3.2*). For this reason we briefly discuss the Sun RPC implementation in this section. Sun RPC provides a communication paradigm for distributed applications also using a client-server model[20]. For this purpose the client first calls a procedure to send a data packet to the server. When the packet arrives, the server calls a dispatch routine, performs whatever service is requested, sends back a reply, and the procedure call returns to the client. When specifying the server the user needs to register it with a server daemon (portmapper). Also the external data representation (XDR) routines in both the client's and server's code need to be specified so that they can be used to translate arguments and results to and from the local machine's internal data representation. The aforementioned functions are included in supplied libraries as well as other stub routines necessary for remote invocation. In a typical interaction scenario the client first marshalls the arguments into an invocation structure (i.e. translation from local machine representation to canonical representation and complete message formation). It then broadcasts a message to all the node's portmappers to identify the desired server (i.e. determine what socket the server is listening to). After a response is received a direct connection is established between the client and the server (server's dispatcher). Next the message is sent to the server where the arguments are taken from the message structure and translated into the local machine's internal representation. The server executes the requested operation in its own address space making use of any of the aforementioned translated input parameters. The result of the operation, if any, is then translated canonically and the response message structure is formed. The message is finally sent to the client where it is translated into its internal representation before it is delivered to the client. Upon the reception of this message the client process continues. This represents the standard RPC call.

3.2 Benchmarking Computational Throughput: As we have mentioned in section 2.0.1 a distributed system's ability to concurrently process can be used to improve application performance since an increasing number of computational resources can be applied to the problem at hand. Thus, a distributed environment should be benchmarked with respect to this attribute in an attempt to characterize its performance. In this section we describe a distributed application that was designed and used to benchmark this attribute of the distributed environment. The results obtained using this benchmark, as well as the results obtained from any benchmark routine, should not be interpreted as a gauge of overall performance but as a measure of performance subject to a very specific set of conditions.

3.2.1 The Proposed Model: To accomplish the aforementioned benchmarks we designed and implemented a computational model within the Cronus and Sun RPC distributed environments that was used to characterize the underlying distributed environment's ability to concurrently process. A model for benchmarking the aggregate performance (in Dhrystones/sec) of a distributed computing system, while varying the number of concurrently active processes in the whole system has been developed. Specifically, the following three effects were studied for the distributed environment in question:

1. The effect of increasing the number of processing nodes on the aggregated performance measure, that is Dhrystones/sec.
2. The effect of increasing the number of processes per node, which directly carry out the benchmark calculations.
3. The effect of varying the processing load assigned to each of above processes.

Within this framework we developed some important benchmarks for the Cronus Distributed Computing Environment [21] and, by way of comparison, the Sun Microsystems implementation of the RPC [17]. The developed model is general enough to characterize the concurrent processing capability of any distributed computing environment.

It was decided to use a benchmark figure that is familiar within the research community. The Dhrystone benchmark [18] was chosen because it is a well known set of procedures used to benchmark centralized systems [19] (based on processor/processor clock speed and compiler used). In order to obtain meaningful benchmark results the main consideration was to use more computational intensive processes with less interaction with the underlying constituent operating system. The Dhrystone does not interact with the underlying constituent operating system (in our case UNIX) and can be fully encapsulated within a server (or manager). In other words, the routines make no system calls and do not interact with the file system. For the types of distributed systems that are implemented on a native operating system it is necessary to carefully choose the encapsulated benchmark procedures so as not to interact with the underlying systems such as UNIX, VMS etc.

Based on the characteristics of Cronus and Sun RPC, we assume a client-server model [20]. In this model clients make calls to a population of servers which can be resident on several nodes in the system. These nodes are generally processors interconnected through a

network which provides a transportation mechanism for carrying messages among the entities. The servers are inactive entities waiting for an invocation from a remote client. The processing model describes the types of processing and interaction needed among a client and servers to carry out the desired benchmarks. The server encapsulates a routine capable of calculating a desired number of Dhrystones requested by the client process. The proposed processing model is essentially a distributed benchmark application that is designed to invoke a number of servers that are evenly distributed among a number of nodes (see Fig. 3.1.1).

As mentioned above, in order to obtain the overall performance the number of Dhrystone computations performed by each server is varied by the client application process. In general, we first decide on the total number of nodes which constitute the distributed operating environment, then, through a series of remote calls, a fixed number of server processes are created on each of the nodes. The aforementioned initialization of the distributed environment (as well as all test runs) was fully automated using a set of simple shell scripts written using the underlying constituent operating system (Unix). This demonstrates one of the unique qualities of a distributed computing environment like Cronus. An application designer may make full use of the many features of the local operating system he is accustomed to while also using the many features gained through the use of distribution and multiple resources in Cronus. Next each of the server processes are then invoked by the client to carry out a certain number of Dhrystone calculations. When a server completes its calculations successfully, it then sends an appropriate message to the client application process. The client process, upon receiving all such messages, calculates an aggregate performance measures (in Dhrystones/sec.) for the distributed environment.

The general flow of processing for the client application process and the server processes are described below. Implementation pseudocode specification for the Cronus and Sun RPC environments used in our experiment are described in Appendix B.

Client Application Process

- 1.) Obtain the system time locally to determine the start time of the experiment. It is assumed that all of the server processes on the appropriate nodes have been created.
- 2.) For each of "n" servers running on every node invoke a Dhrystone operation with "k" number of benchmarks to be performed. (NOTE: The application process invokes servers in a round robin fashion on the nodes integrated in the distributed system. That is, the invocation of server 1 is first sent to node 1, then to node 2, and then to node 3 (assuming 3 nodes are integrated in the system). Then invocations for server 2 are then sent using the same order of nodes and so on. The process of invoking servers continues until all the servers integrated are invoked.
- 3.) Wait for a success response from each and every server.
- 4.) Take another time hack locally to record the finish time of the experiment.

- 5.) Since the total number of invocations is known to the client application process and the elapsed time to do those benchmarks is also known, the aggregate rate of calculating Dhrystones benchmarks for the distributed environment is then computed.

It is important to note that all invocations must obviously be non-blocking (if not, the computation will not be concurrent) and that the invocations are done in round robin manner in an attempt to balance the processing load on each node. Also, the invocations are performed sequentially rather than by broadcast in an attempt to keep the application design sufficiently generic (that is, not all distributed environments support lower level broadcast).

Server Process

- 1.) The number of Dhrystones to be calculated (k), is extracted from the message received from the client process (this also assumes that the data has been translated into the machine's internal representation as well).
- 2.) Call the local Dhrystone calculation procedure k times.
- 3.) Prepare and send the response to the client application process.

Since every distributed environment must support remote computations (albeit with varying facility), the above model is sufficiently generic for characterizing distributed systems and studying their performance while varying the various parameters mentioned above, assuming full load balancing capability.

3.2.2 Implementation Details: This section briefly describes the benchmark implementation details that were specific to each of the two distributed environments used.

3.2.2.1 Cronus Benchmark Implementation Details: Specifically the Cronus test application uses direct addressing of operation invocations, that is the application *knows* where all of the individual object managers (servers) reside. This is not necessary since the Cronus kernel provides a locate mechanism that enables the application designer to only specify the *type* of the object manager and the operation to be invoked. The direct addressing mode was used in order to avoid possible overhead of doing a kernel locate (i.e. wanted the maximum capability of the environment). Also, the test servers (managers) do not maintain persistent object state. That is they do not fetch the variables manipulated by the Dhrystone procedures from the object database stored on disk. The reason for using this approach is to attempt to achieve a maximum capability. It would be interesting, however, to declare some fixed percentage of the manipulated variables as object state and force the operation (Dhrystone calculations) to interact with the object instance database every time they are manipulated.

3.2.2.2 Sun RPC Benchmark Implementation Details: The benchmark application requires

that all the invocations must be non-blocking. It would appear at first glance that the RPC environment would be inappropriate for such an application. This is not true, however, since it is possible to design such an application using Sun RPC. Sun RPC allows non-blocking calls provided a result is not expected from the server (which is also referred to as remote batch processing). To implement the benchmark application and to receive a result at the client site, it is essential in the Sun RPC implementation that the client register itself as a server, after sending all invocations. We call the client operating in this mode a *pseudo server*. In this mode the application then essentially waits for the Dhrystone servers to invoke operations on it to provide the result (Success flag) as an argument. An analogous method is used in the implementation of the *Dhrystone servers*. They first start as a server to the application client routine then, after calculating their k Dhrystones, they become *pseudo clients* of the original client process (which would be acting as a *pseudo server*). This technique can work satisfactorily but has one limitation. Sun RPC requires that the transport layer mechanism be TCP when doing *remote batch invocation*. Normally RPC requires a connection for each invocation of a server but, since we require an invocation on the the application for each result, we double the number of connections being established and broken down. This is an overhead if we constrain ourselves to using Sun RPC and not altering lower layers of its protocol hierarchy. A pseudocode specification for the Sun RPC implementation has been included in Appendix B.

3.2.3 Discussion of Results: For both the Cronus and Sun RPC domains the test application described was run in the configuration described in section 3.1.

In order to study the effects of increasing the number of processing hosts and the number of servers per host, the system was initially configured as a single host system and was populated gradually with an increasing number of servers. Subsequently, the configuration was expanded by incorporating two more hosts. The servers were instantiated on all the three hosts by gradually increasing their number but keeping their population balanced among the hosts.

More specifically the environment and workload for benchmarking is described as follows:

{1 *host case*:} In this case experiments were performed by first invoking a single server and gradually increasing their number to 10. For each of these configurations the benchmark application described in section 3.2.1 was implemented both for Cronus and Sun RPC as described by the pseudocode in Appendix B which essentially consisted of a series of Dhrystone calls. The number of Dhrystones to be computed by each server per call was varied as 500, 700, 1000, 2000, 3000, 4000, 5000, 7000, 8000, 9000, 10000, 20000, 30000, 40000, and 50000.

{2 *host case*:} This case is essentially the same as case 1 in terms of implementation. However we attempted to achieved balanced loading among the two hosts by invoking the servers in a round-robin scheme. The round-robin scheme provided a good balance of load, except when the number of Dhrystones to be calculated was small. More discussion of this experience is given later in the section.

{3 *host case*:} This case is also identical to the 2 Host case in terms of its implementation and invocation of servers for load balancing.

The raw data from each run for both the Cronus and Sun RPC domains are included in Appendix C. The analyses of the trends and behavior for each of the environments are given in Figures 3.2.1 through 3.2.5. A discussion of these graphs as well as some possible reasons for the observed behavior are discussed in the following subsections.

3.2.3.1 Results for the Cronus DOS Implementation: Figure 3.2.1(a-d) provides the aggregate response of the distributed environment (Dhrystones/sec.) versus the load placed on each OM (Object Manager - server) in terms of number of Dhrystones to be performed per call. This figure represents the cases where 1, 2, and 3 hosts integrated into the environment. In these figures we have included the cases where 2,4,6, and 10 OM's were running per host. It can be noted from these figures that the the throughput (in terms of aggregate Dhrystones/sec) increases as the workload increases. Since, with the increase in the load, a greater percentage of the overall time spent by the distributed environment is in actual calculation of Dhrystones as opposed to the percentage of time spent in communication involving server invocation and response, the canonical translation on both ends, and actual message transmission, the environment starts acting more like a multiprocessor system rather than a distributed system.

It can also be noticed that there is always a payoff when we add more hosts to the configuration (see Figure 3.2.1(d)). However at the same time communication overhead increases as we increase the number of hosts. Therefore there exists a tradeoff between the amount of computation performed per host (Dhrystones/call), the number of servers, and the number of hosts integrated into the system. This is discussed later in this section.

Another important observation which can also be made from Figure 3.2.1(d) is that for the test cases of 1, 2 and 3 hosts, the maximum throughput approached very closely to the benchmarked capability of 1, 2, and 3 Sun 3's respectively, regardless of the number of the servers invoked on each host. During this experimentation the Sun 3 was also benchmarked using the same Dhrystone procedures which were encapsulated within each server. The resulting performance was 6216 Dhrystones/sec. This result was also obtained for the single host case for which the maximum aggregate performance was 6090 Dhrystones/sec. For the two host case the maximum aggregate performance was 12030 Dhrystones/sec, which approaches the maximum capability of 2 Sun 3's. Also, for the three host case the result was 17952 Dhrystones/sec which also approaches the performance of three Sun 3's. The results presented here are valid benchmarks due to two reasons. First, the system was dedicated to the experiments with no other user. Second, since the throughput in all the cases approaches the maximum capability of the Sun 3 hosts, we assert that all the hosts were kept busy during our experimentation with negligible interference from the constituent operating system.

Figures 3.2.2 (a) and (b) give a slightly different view of the data. For each of the plots the aggregate response is plotted versus the number of OM's (servers) that are running in the test configuration per host. Each curve corresponds to a specific load being placed on the individual OM's (Dhrystones/invocation). It can be noticed from these figures that by adding more OM's running on each host an incremental increase in the aggregate performance (Dhry./sec.) can be obtained. After reaching a certain level the performance remains approximately constant before it starts decreasing. The range over which performance remains at a certain maximum level depends on the number of hosts and the number of

CRONUS 1 HOST

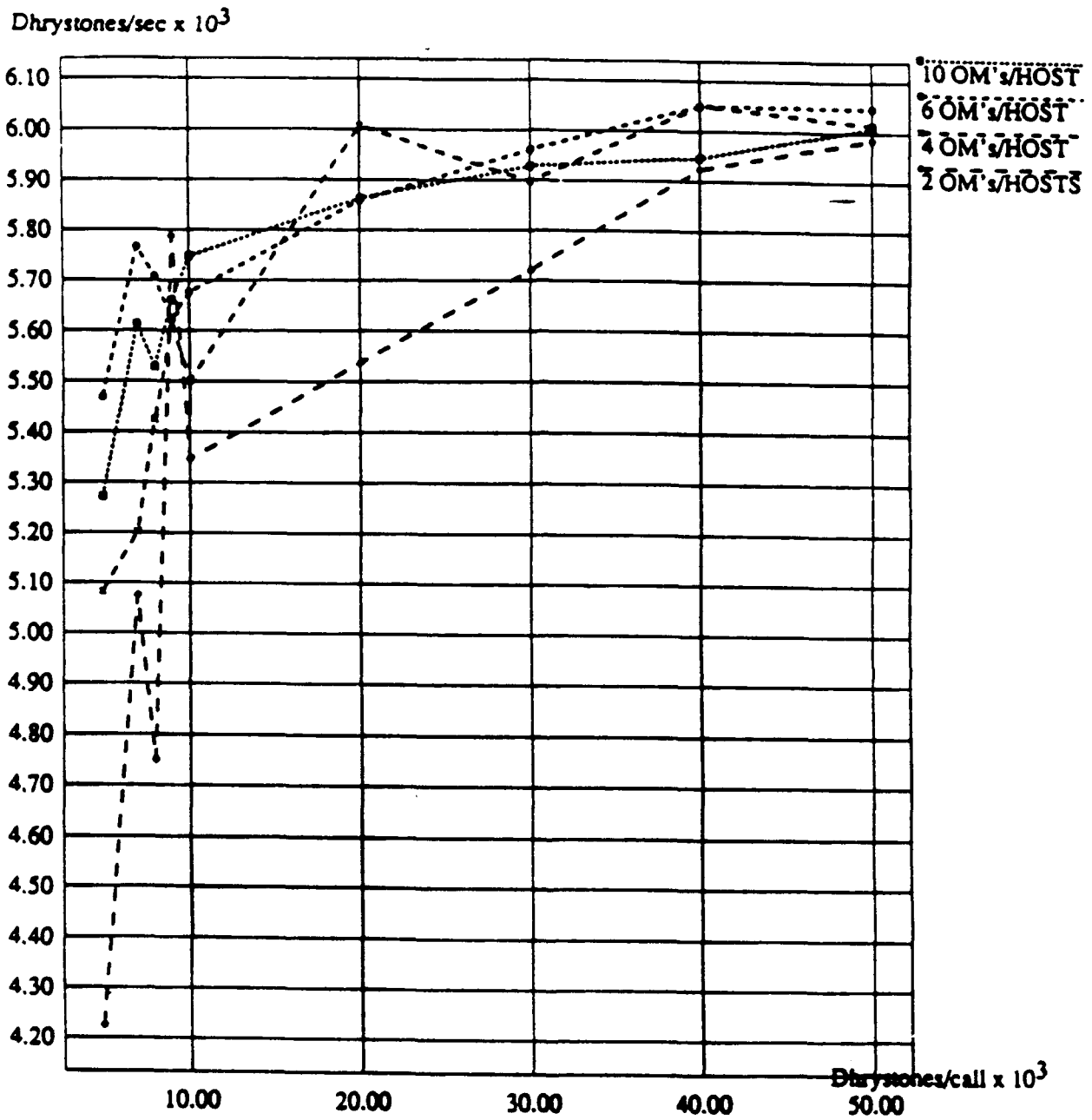


Figure 3.2.1(a) Throughput of Cronus for 1 host.

CRONUS 2 HOSTS

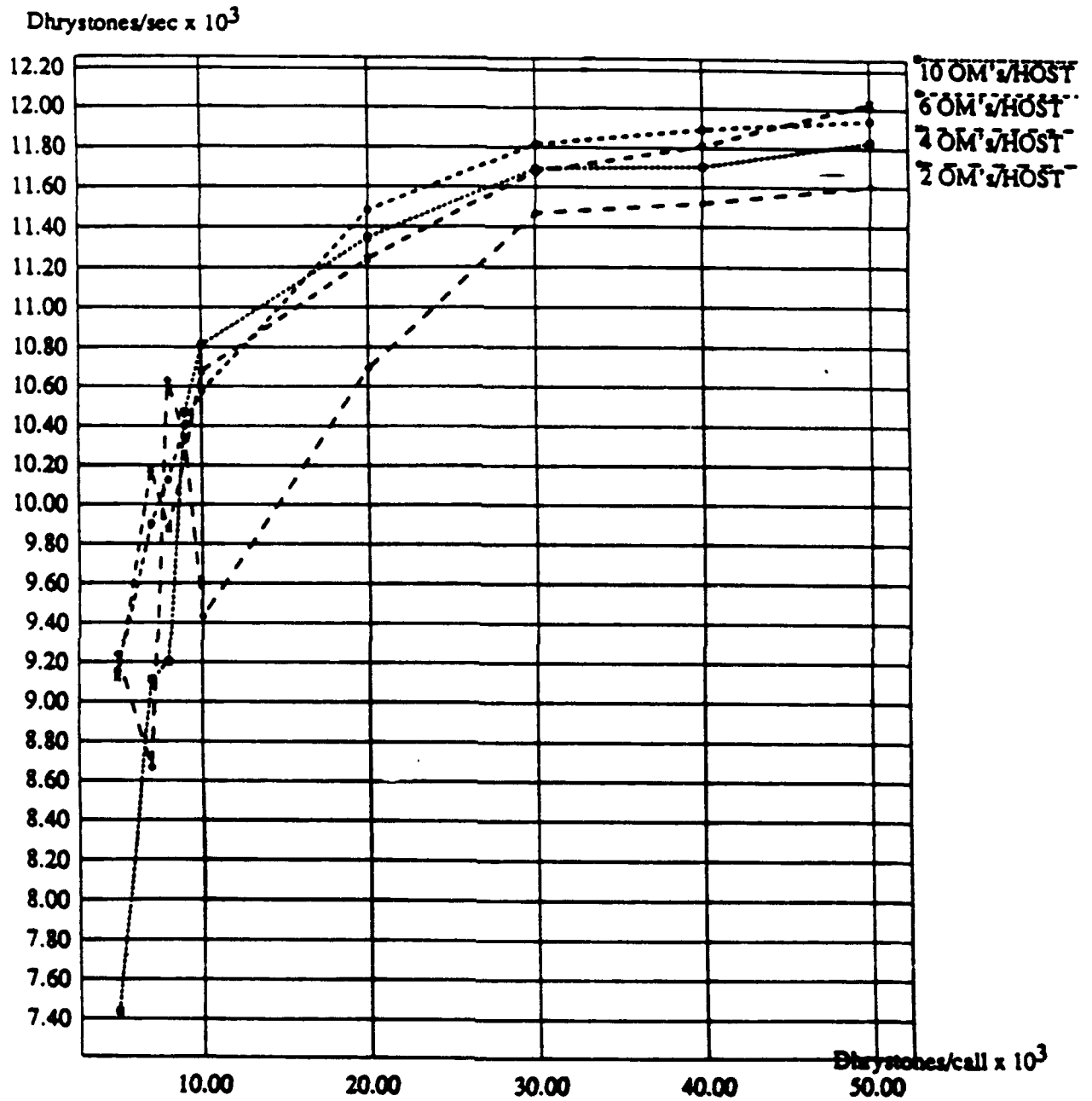


Figure 3.2.1(b) Throughput of Cronus for 2 host.

CRONUS 3 HOSTS

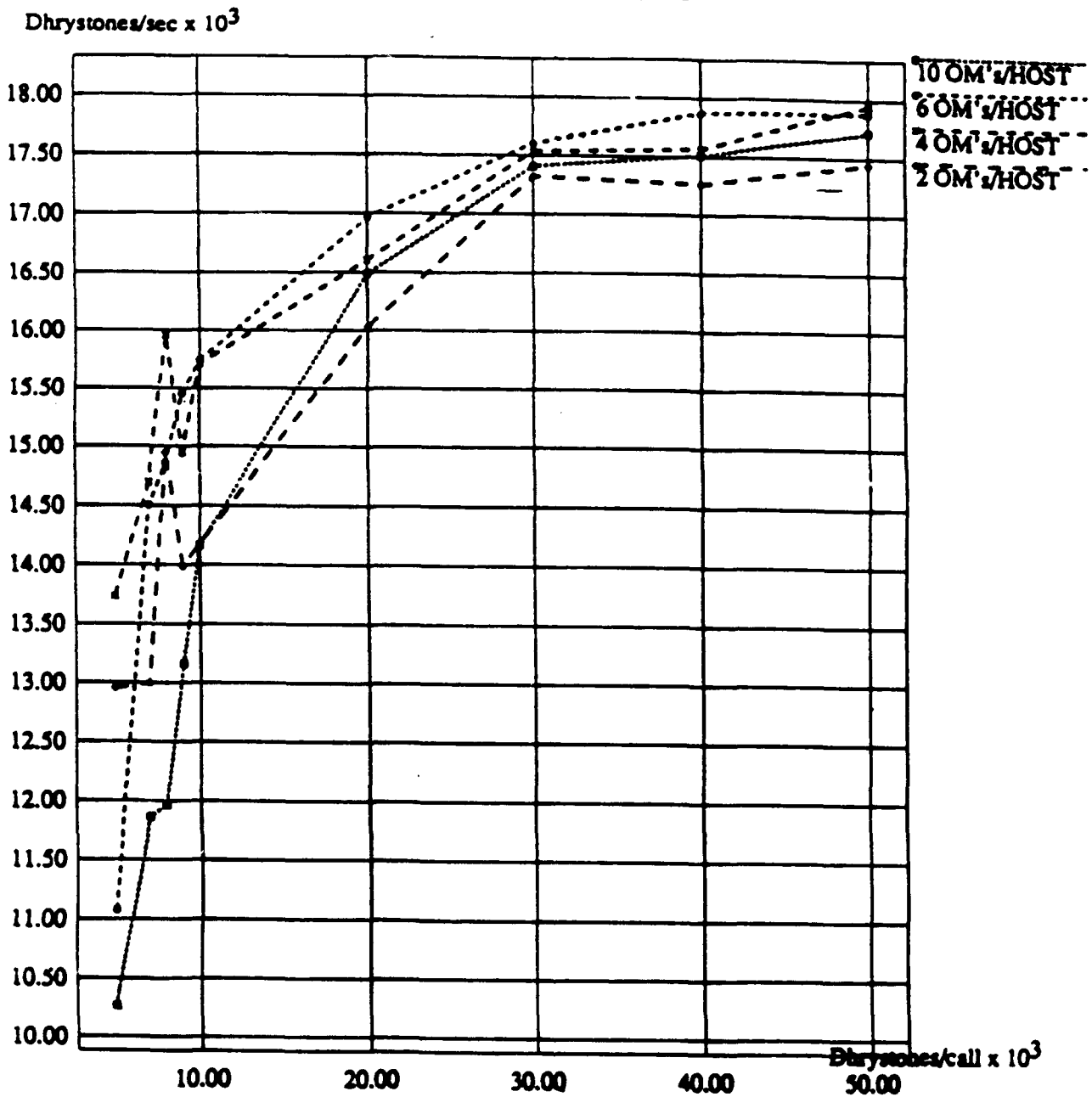


Figure 3.2. 1(c) Throughput of Cronus for 3 host.

CRONUS 1,2 AND 3 HOSTS

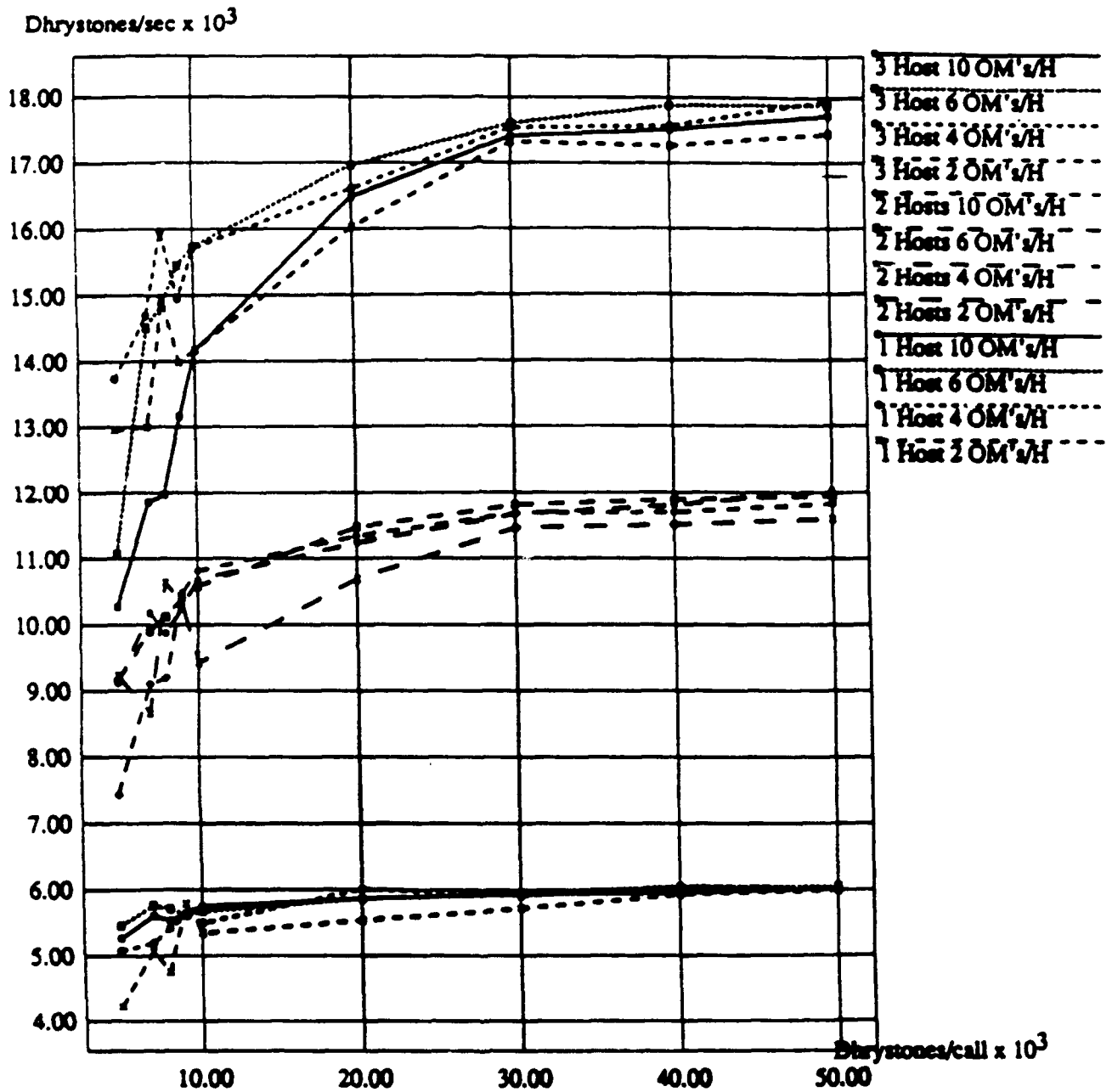


Figure 3.2.1(d) Throughput of Cronus for various hosts.

CRONUS 2 HOSTS

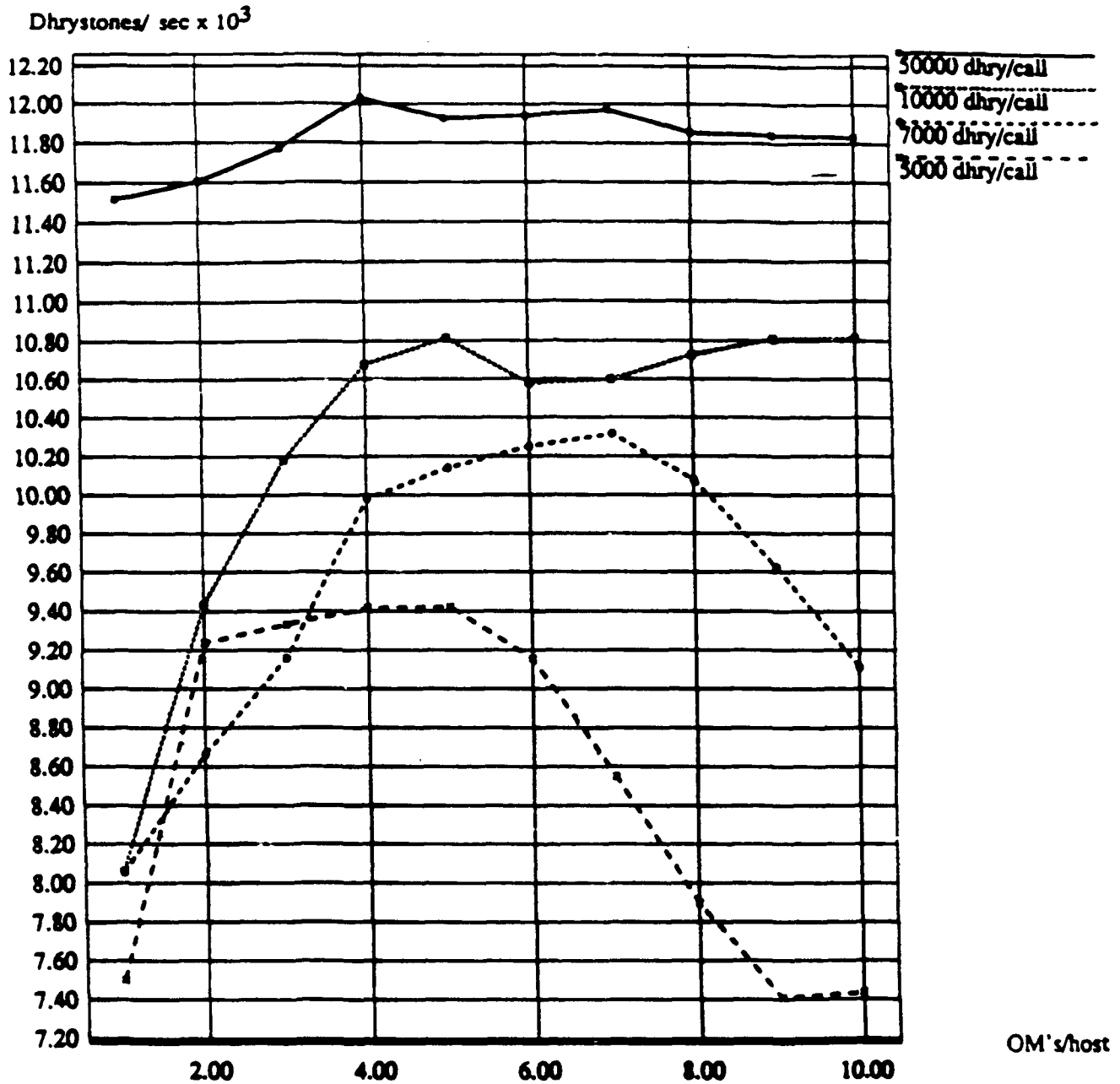


Figure 3 2.2(a) Throughput of Cronus for various OM/Host (2 host).

CRONUS 3 HOSTS

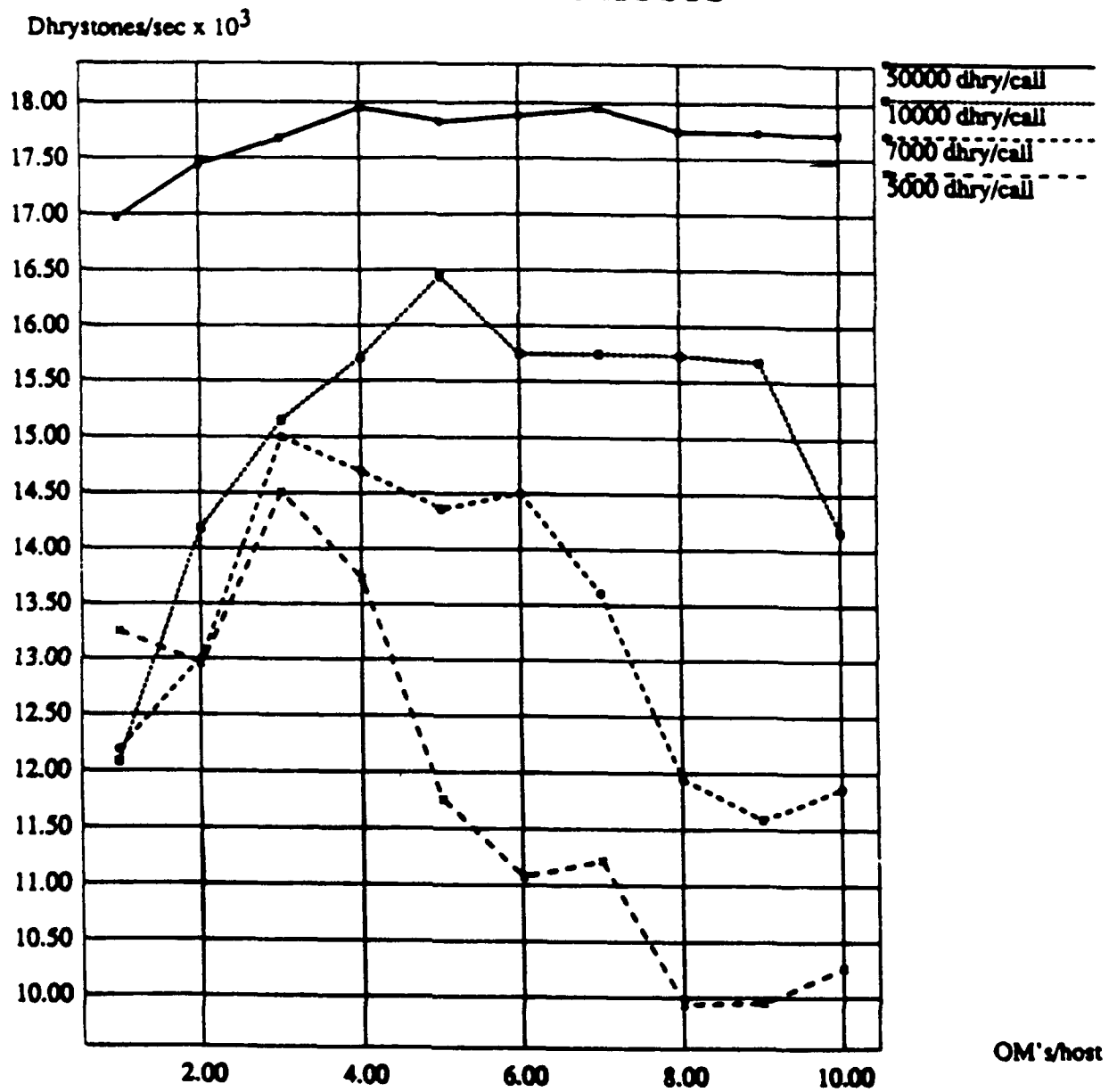


Figure 3.2.2(b) Throughput of Cronus for various OM/Host (3 host).

Dhrystones/call. This effect can be explained in general by realizing that there is a trade-off between the cost of interaction in the distributed environment (i.e. message formation, canonical translation, etc.) and the amount of useful work being done by each host (i.e. Dhrystone calculations).

For example it can be noticed that the range at which the performance remains at the maximum level increases with the number of Dhrystones/call. This indicates that the amount of useful work requested for each host remains substantially higher as compared to the overhead associated with the number of invocations. In other words, this overhead has little impact on the overall performance of the distributed system even if a large number of servers are invoked. However for a smaller number of requested Dhrystone calculations (Dhrystones/call) the overhead associated with invocations has a pronounced impact on performance. The reason for this decrease in performance is due to the fact that as the number of OMs running per host are increased (or increasing number of hosts as well), the overhead involving invocations in terms of message formations, and canonical translations also increases.

Another possible explanation of this effect may be found when we consider that Cronus runs on a constituent operating system. It is possible that we are witnessing the effects of increased process scheduling, paging, etc. on the underlying operating system (Unix).

We want to indicate that this overhead does not appear to be a linear function of the total number of invocations. For example, for the case of 2 hosts, with 5000 Dhrystones executed by each of the 5 servers (per host) produce an aggregate performance of 9421 Dhrystones/sec. On the other hand, for the case of 10 servers per host, the aggregate performance is 7436 Dhrystones/sec. The total number of Dhrystones to be calculated at the remote site in the latter case is exactly twice the number in the former case. Since, the host is always active for the experiment, we can note that the average time spent to process (generation, unpacking, etc.) an invocation is not exactly the same in both the cases. A simple calculation can reveal that for the case of 10 servers/host the time incurred for total overhead is not twice that for the case of 5 servers/host. Such an observation can also be made from the non-linear trends of the graphs for a small and a large number of OMs in Figures 3.2.2(a) and (b).

For low computational requirements this effect is prominent. This may be due to the fact that for low computational requirements, a process suffers less of a penalty for swapping out of a host CPU than for the case where a process needs to perform extensive computation (for example, more than 10,000 Dhrystones). The effective overhead associated with process swapping is the probable cause of the sharp increase in throughput for a small load with a small number of OMs (ref. Figures 3.2.2(a) and (b)). On the other hand, the overhead associated with a large number of invocations (for a large number of OMs) causes performance degradation at an increasing pace for smaller load values than for larger load values.

The non-linear phenomena disappears as we increase the number of Dhrystone calculations per server. This is clear for the case of 50,000 Dhrystones/call in Figs. 3.2.2(a) and (b). However, as mentioned above, we also expect this curve would drop off as we increase the number of servers beyond a certain value.

We want to comment that although there exist certain spurious data points in the range of maximum performance of these graphs, the maximum deviation from the average trend is rather negligible. A number of factors may be causing these spurious data points, such as retransmission of messages on the communication channel. However, these graphs are meant to display more of a trend rather than specific observation points.

3.2.3.2 Comparison of Results for Cronus and Sun RPC Environments: An analogous set of graphs provided for the RPC environment have been included in Figures 3.2.3(a)-(f). It can be observed from these figures that the performance trends for RPC are similar to those exhibited by the Cronus environment. The comparison between the two systems is shown in Figures 3.2.4 and 3.2.5. From Figures 3.2.1(b)&(c), 3.2.3(b)&(c), 3.2.4, 3.2.5, we notice that both environments are comparable in performance for higher loads (Dhrystones/call), irrespective of the number of servers invoked at each host. The reason being that the percentage of time spent in actual calculation of Dhrystones is much higher than the percentage of time spent in interaction among servers and client (i.e. canonical translations, communication). However, for smaller loads (see Figures 3.2.2(a)&(b), 3.2.3(e)&(f)) (less than 10,000 Dhrystones/call) the Cronus throughput reaches its maximum value for a lower number of servers per host as compared to Sun RPC, while in the Cronus environment this maximum level is for the most part greater than or equal to that of the Sun RPC environment. Also, we notice that performance decreases more abruptly as we increase the number of servers running per host in the Cronus environment.

There may be a number of reasons for this observed result. The first one is the added overhead present in the Cronus implementation of the IPC layer (discussed in Section 3.1.1.1). Such an overhead is not present in Sun's implementation of the RPC communication hierarchy. Also, Cronus routes all messages through a Cronus kernel process running on each host while Sun's implementation of RPC establishes a connection between the communicating client and server. The effect of added message processing done by the Cronus kernel and handling lower level communication through its constituent operating system (Unix) manifests itself in the observed performance as we increase the number of servers in this environment and hence the amount of message traffic being generated. On the other hand, in Sun RPC interprocess communication is directly handled by the lower levels of the IP hierarchy. Another reason for this observed result may be a possible optimization of Sun RPC implementation by using implementation specific knowledge of the constituent operating system (Sun OS version 3.5).

SUN RPC 1 HOST

Dhrystones/sec $\times 10^3$

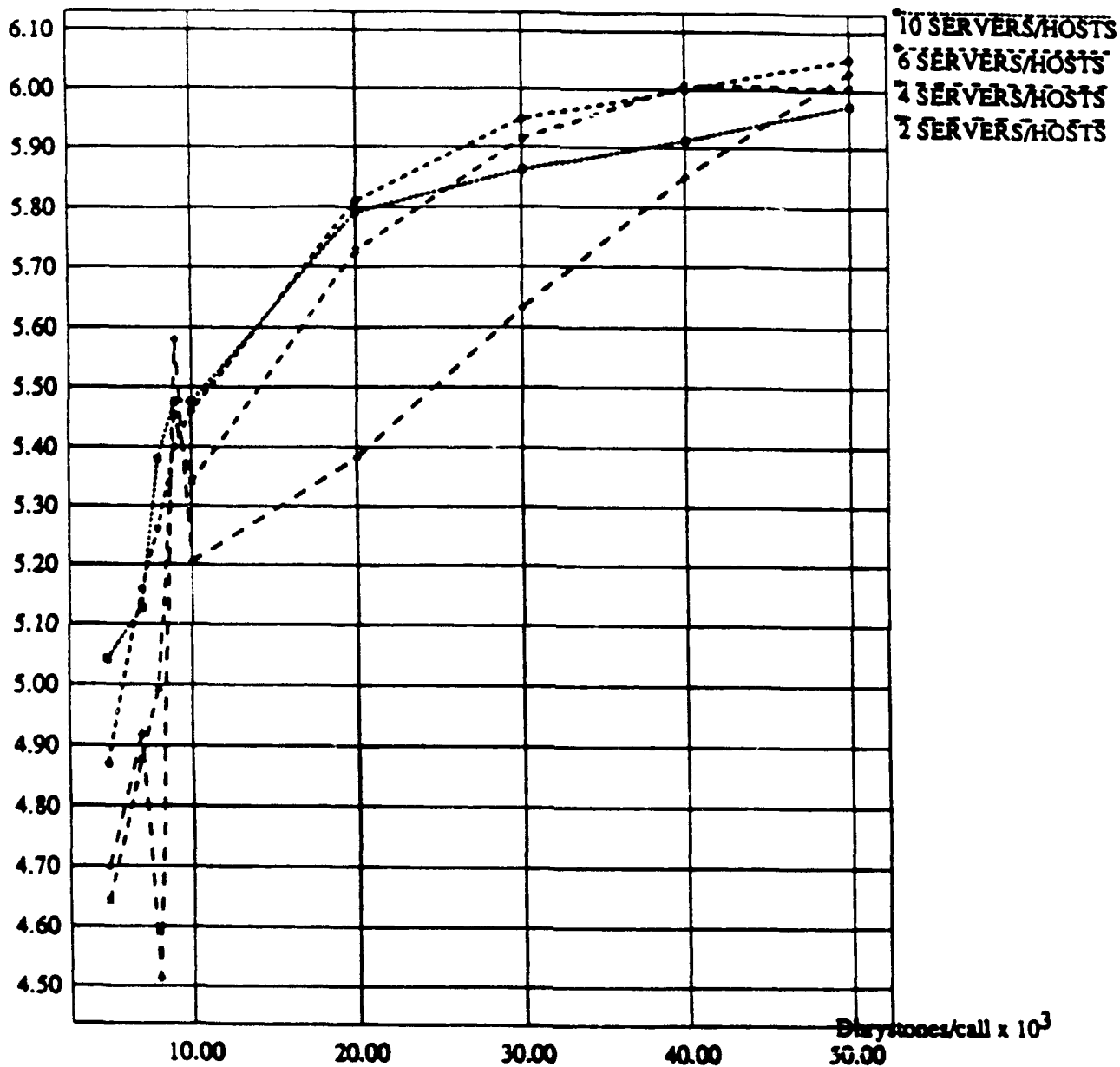


Figure 3.2.3(a) Throughput of SUN RPC for 1 host.

SUN RPC 2 HOSTS

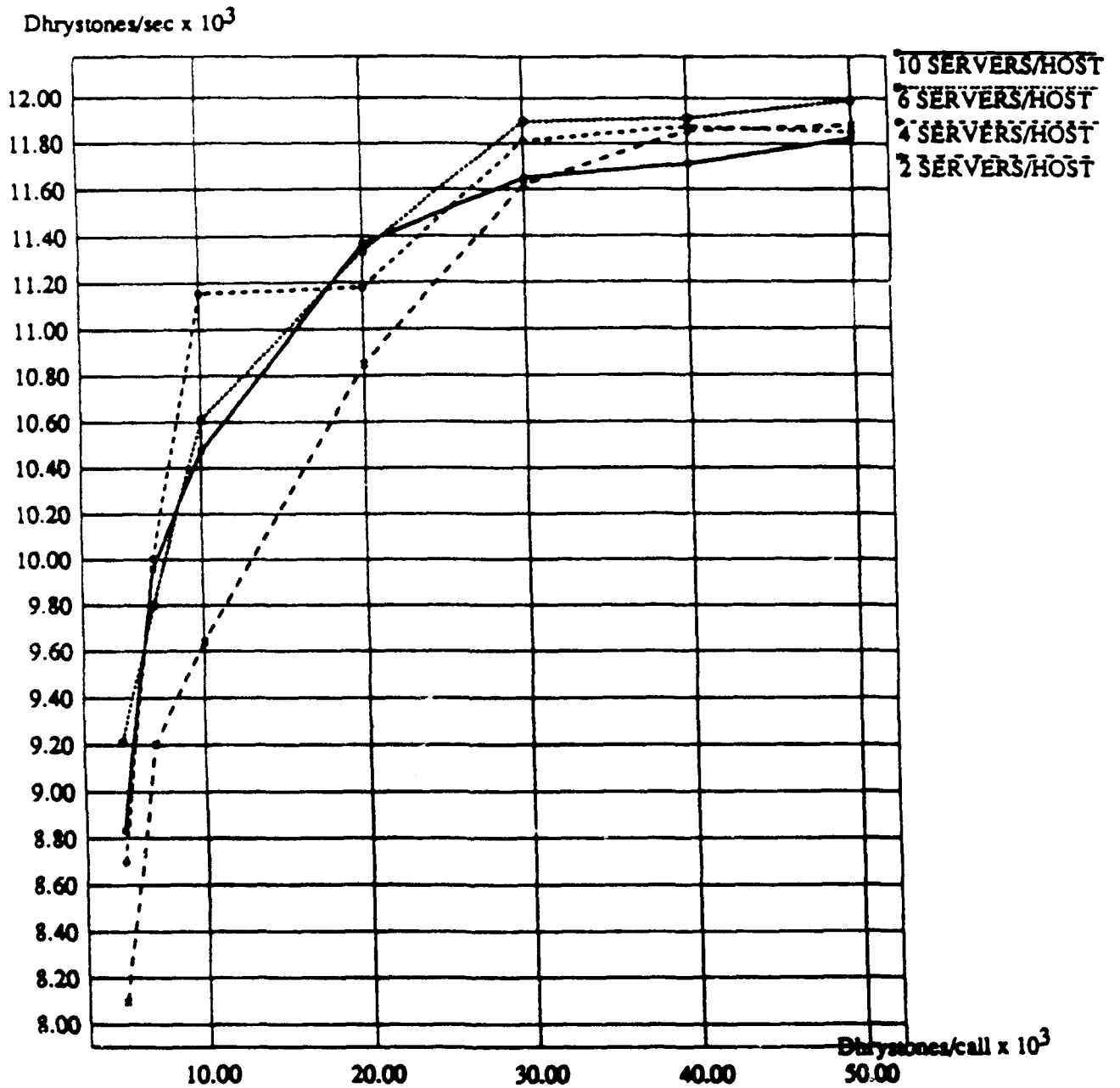


Figure 3.2.3(b) Throughput of SUN RPC for 2 host.

SUN RPC 3 HOSTS

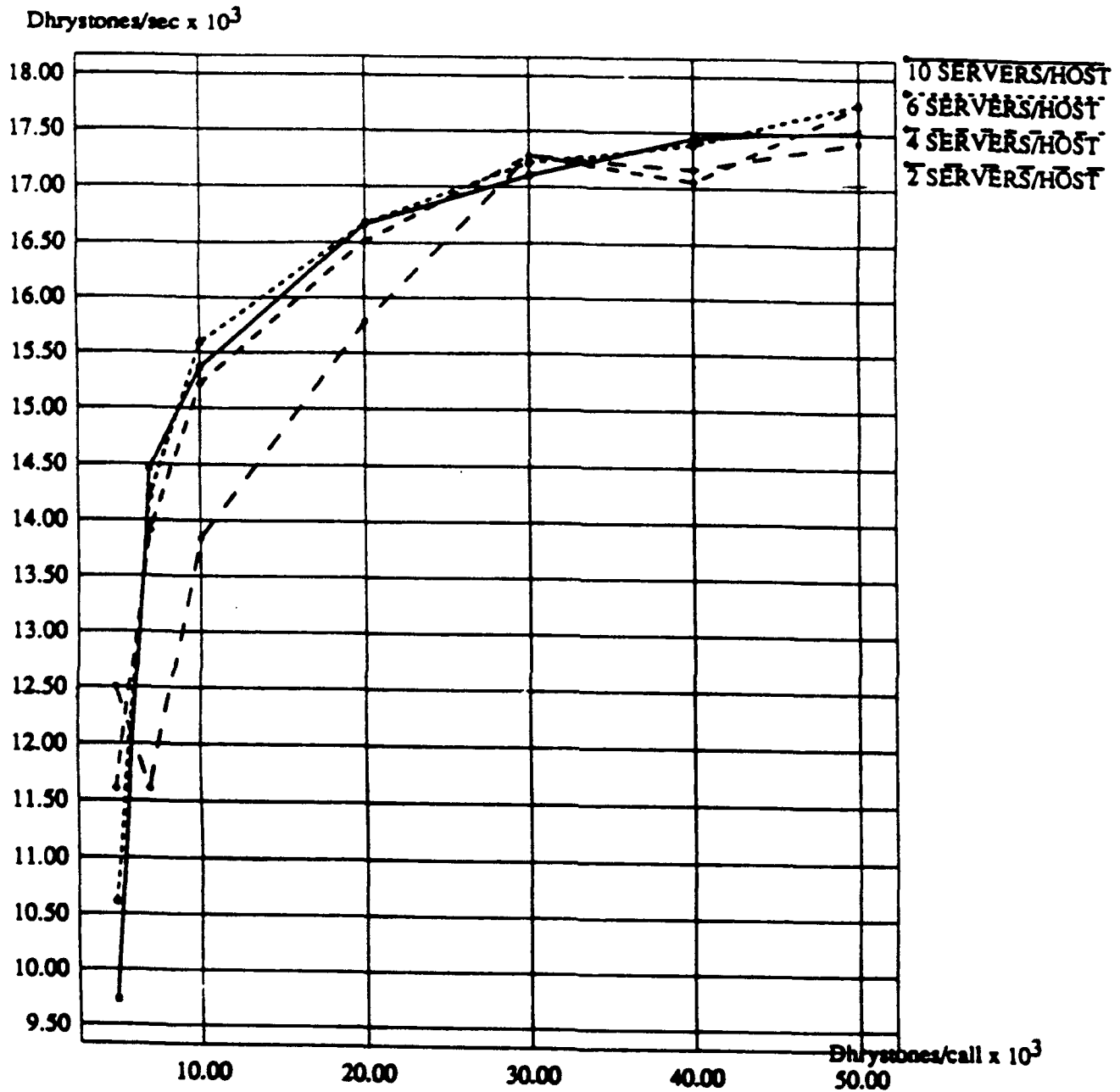


Figure 3.2.3(c) Throughput of SUN RPC for 3 host.

SUN RPC 1,2 AND 3 HOSTS

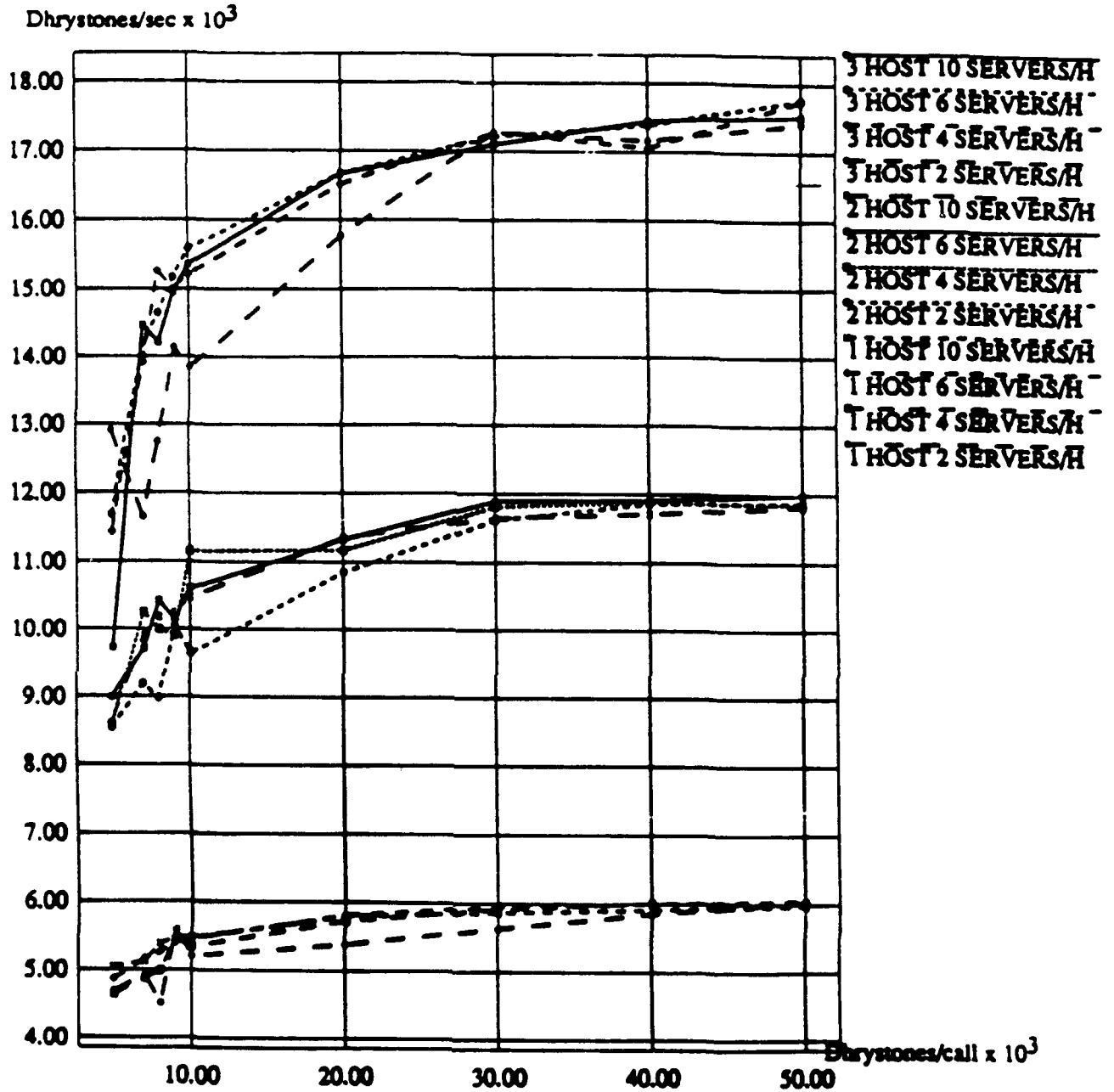


Figure 3.2.3(d) Throughput of SUN RPC for various hosts.

SUN RPC 2 HOSTS

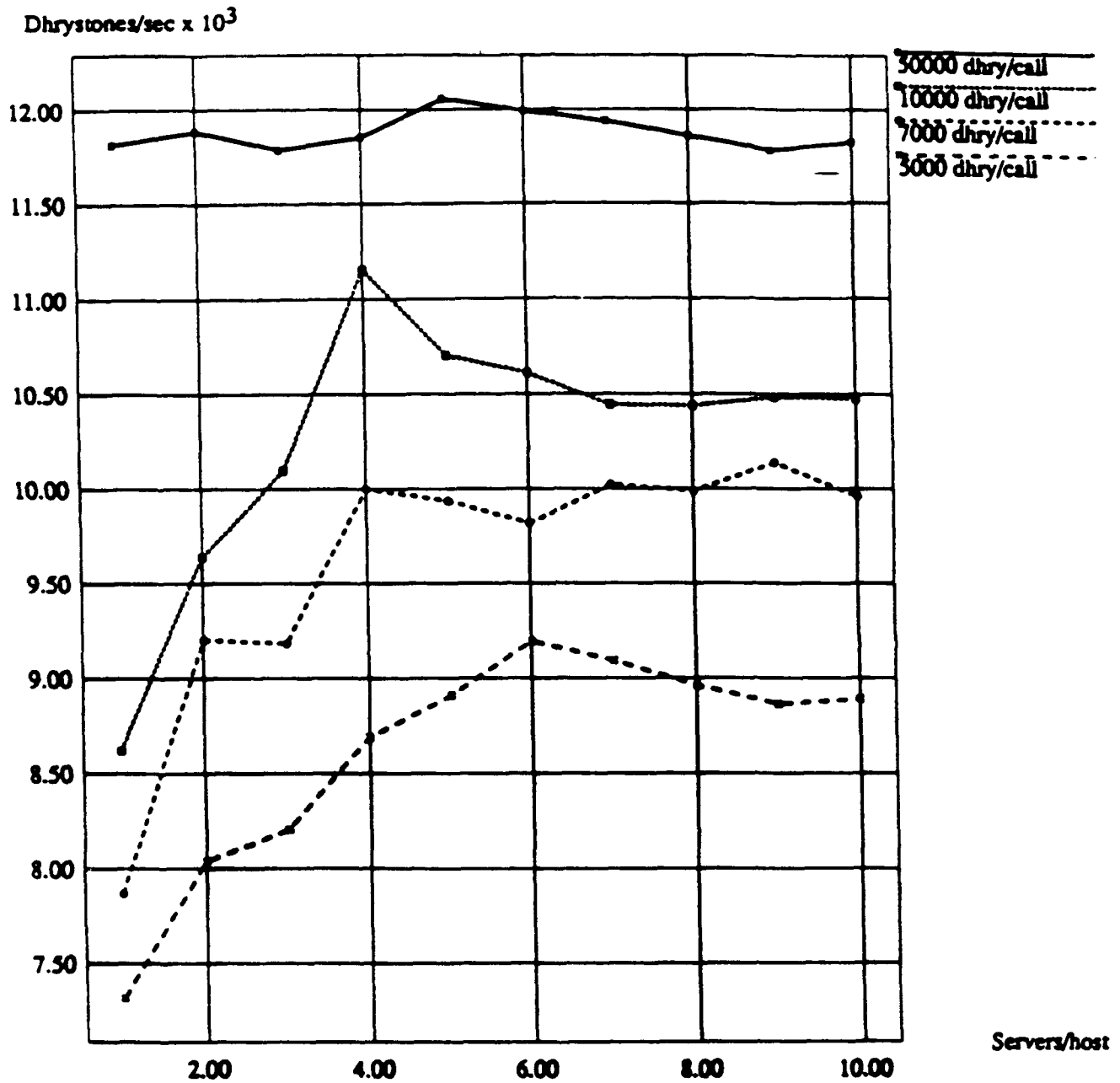


Fig. 3.2.3(e) Throughput of SUN RPC for various Servers/Hosts (2 hosts)

SUN RPC 3 HOSTS

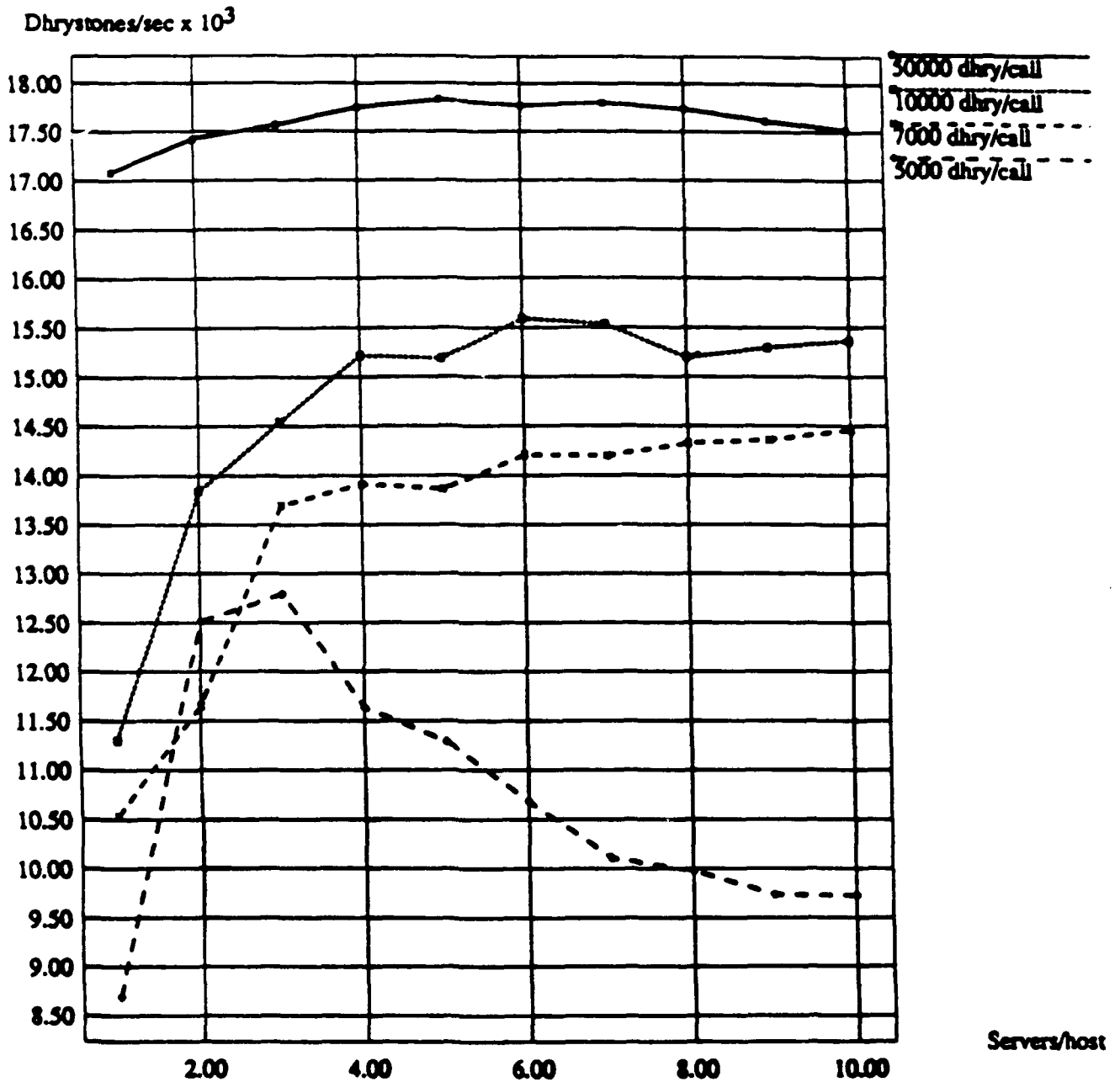
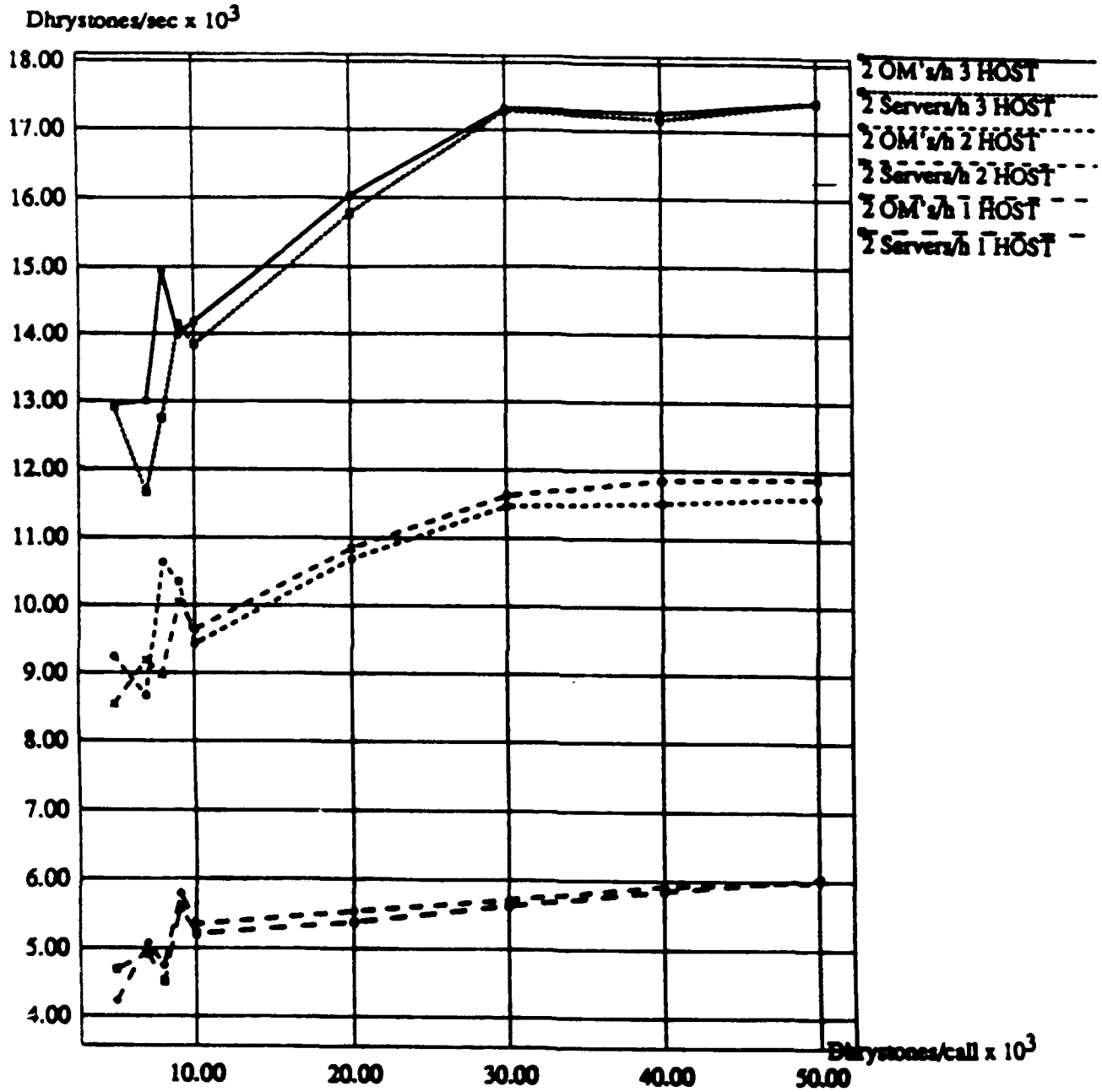


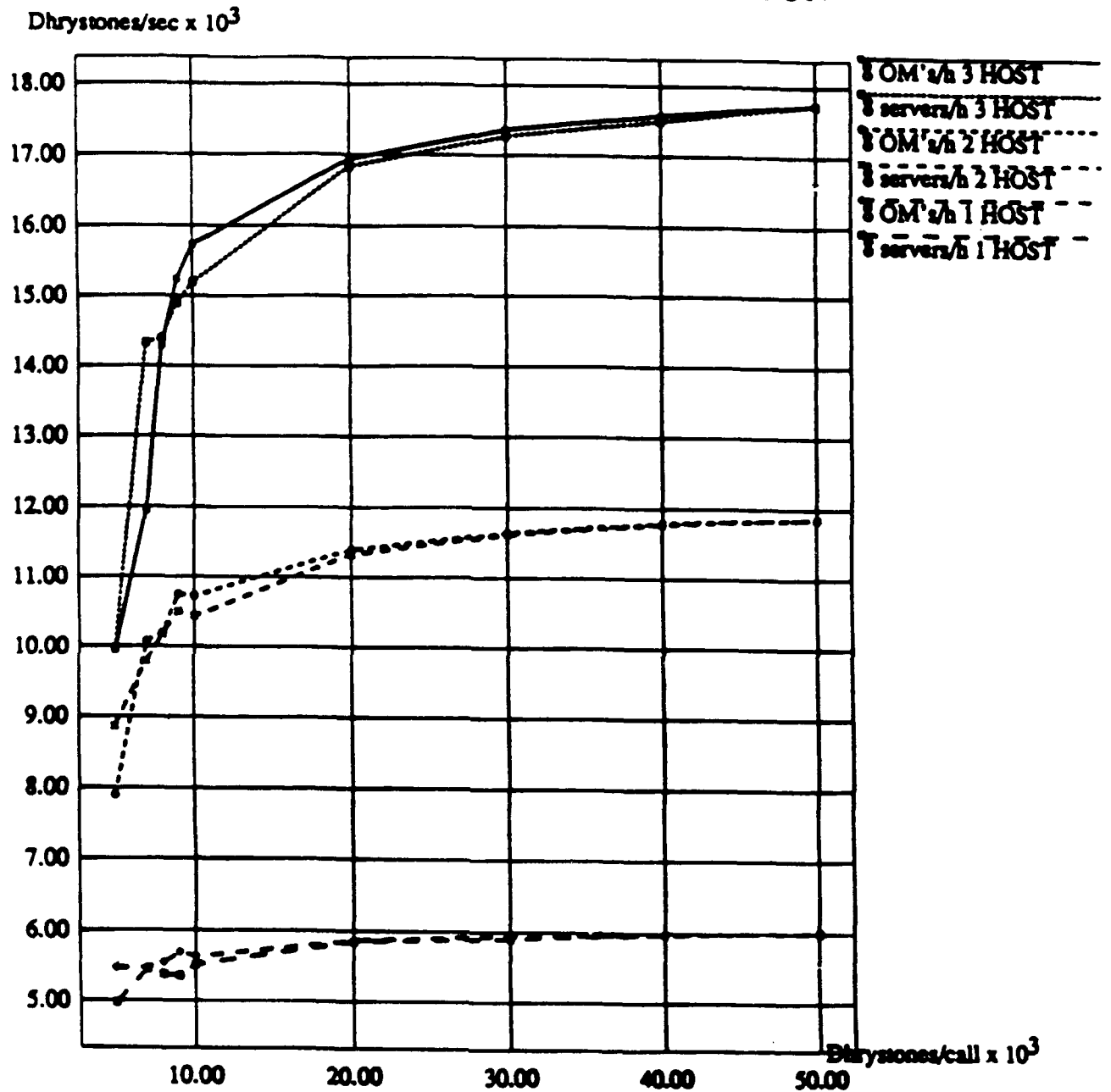
Fig. 3.2.3(f) Throughput of SUN RPC for various Servers/Hosts (3 hosts)

CRONUS AND RPC COMPARISON



Figures 3.2.4 Cronus and RPC Comparison

CRONUS AND RPC COMPARISON



Figures 3.2.5 Cronus and RPC Comparison

3.3 Benchmarking Availability and Survivability: As described earlier, there are many different attributes that characterize a distributed operating system (DOS) environment. In this section we will look at the DOS environment's ability to make data and/or processes more available and survivable using replication. There is however a price to be paid in using replication. An example of which is the DOS environment's overhead in keeping the replicated copies consistent. For this we used a performance metric to compare read and write latency times when accessing replicated copies as opposed to non-replicated copies. This metric is generic and can be easily implemented on any DOS environment that has the ability to replicate data. Cronus provides mechanisms to support replication where as Sun RPC does not, therefore we will only discuss Cronus in this section.

3.3.1 The Proposed Model: The model is essentially a distributed application consisting of a "client" and "servers". The application will enable us to compare read and write latency times for accessing replicated objects as opposed to non-replicated objects. The data gathered using this application will allow us to determine a performance cost that must be paid for the added benefits of replication.

The application first creates a non-replicated copy of some data type. It should be noted that the data should not be located on the same host that is issuing the invocations. This forces the non-replicated portion to use remote communication mechanisms, thus making a better comparison with the replicated portion. The basic flow of the non-replicated portion of this test is as follows:

- 1) Obtain the local system time in order to record the start time of the experiment.
- 2) Invoke a read operation.
- 3) Wait for response that read completed successfully.
- 4) Obtain the local system time in order to record the finishing time of the experiment.

These steps are then executed "N" times and the average read latency time is then determined. The same basic flow is followed for the write operation and the average write latency time is also determined. This will give us a baseline for comparison with the replicated case. For the replicated portion of the model it is necessary to create a number of replicated copies of some data type. It should be noted that the same data type, data, and operations used to manipulate the data are used in both the non-replicated portion as well as the replicated portion of this model. The replicated portion follows the same basic flow as the non-replicated portion (as outlined above). The only real difference between the two portions of the model (replicated vs non-replicated) is that we are using replicated mechanisms that have to cooperate with all of the replicated copies. This cooperation may be necessary to maintain consistency and detect any would be inconsistencies among the replicated copies or some other reasons specific to the replication mechanisms implemented within the DOS. Once the average read/write latency times for both the replicated and non-replicated portion are determined a comparison of the results can be made to determine the cost that must be paid in achieving various levels of desired availability and consistency.

3.3.2 Choice of Data: Since Cronus is an object oriented system, data items within the Cronus environment are referred to as object instances. An object manager is used by Cronus to manage objects of one or more object types. Within object types there can be any number of object instances (data items). The fact that there may be any number of object instances maintained by a Cronus object manager spawned concerns about how our results could potentially be affected. For example suppose the data in question is of an employee record type, in this example the object instances would be the information about each employee. The concern is whether or not Cronus incurs greater overhead when accessing different object instances (i.e. the 1st as opposed to the 1000th instance). This concern was laid to rest by some companion work done by MITRE which determined there was no appreciable difference in accessing the 1000th object instance as compared to accessing the 1st object instance. For this reason and for simplification we decided to access (read/write) one object instance for both the replicated and non-replicated portions of our model. For the object instance we decided to create a user defined non-trivial canonical type built from Cronus provided canonical types. The user defined canonical type selected for the object instance is a single record containing Cronus provided canonical types which is used to store information about an employee. A detailed description is given in Figure 3.3.1.

3.3.3 Replication in Cronus: When a replicated object is updated it is necessary to bring all the replicated copies to a consistent state. Within the Cronus replication mechanisms there are two update strategies; one being update by replacement, the other being update by operation. The update by replacement strategy is performed by Cronus in the following way. An operation is invoked on an object manager. This object manager performs the operation and copies the entire object instance by way of object managers to other replicated copies of that object instance. The update by operation strategy is performed by Cronus in the following way. An operation is invoked on an object manager. This object manager performs the operation and invokes the same operation on all the replicated object managers that manage that object instance. Obviously update by replacement should be used for small data instances while update by operation can be used for large data instances. The update strategies are very easy to use in Cronus. To implement the different update strategies, a simple one line addition is added to the type definition file describing which strategy is to be used. A precompiler is then used to generate all of the necessary code to implement replication mechanisms/techniques specified.

Cronus implements a form of version voting strategy to detect inconsistencies in replicated copies. Every replicated object has a version vector. The version vector for a specific object type contains a list of hosts that support object managers of the specified type and an associated version number (ref. fig 3.3.2). When an operation is performed on the object Cronus replication mechanisms collect the read or write quorum necessary and locks those copies until the operation has been performed on all of the replicated copies it can locate. If the read or write quorum can not be obtained Cronus will return an error message to the application programmer. Cronus allows the application programmer to select read and write quorums depending on his needs for data availability or consistency. To ensure maximum availability the application programmer should select read and write quorums of one; this however will sacrifice consistency. To ensure data consistency the sum of the read and write quorums must be greater than the total number of replicated copies. Cronus does not however

CANTYPE REPITEM /* User defined canonical data type */

REPRESENTATION IS REPITEM

RECORD

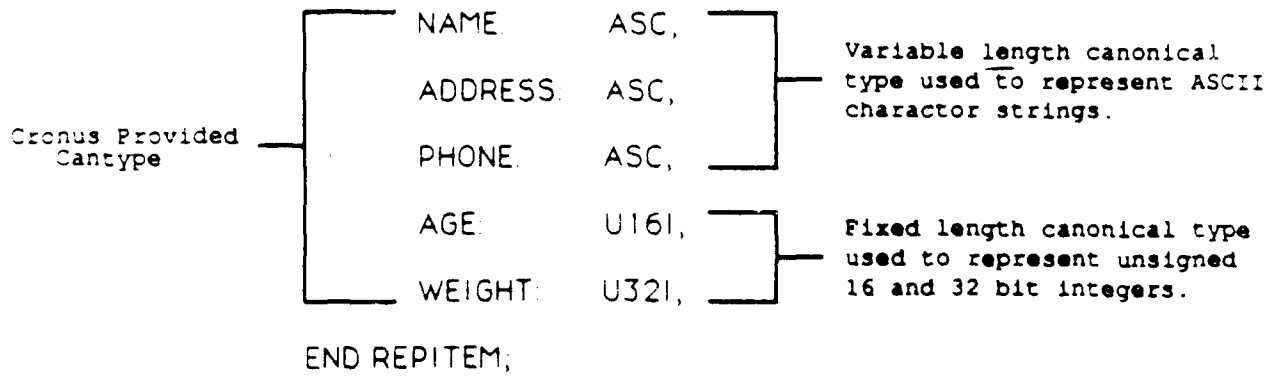


Figure 3.3.1 Detailed description of object instance.

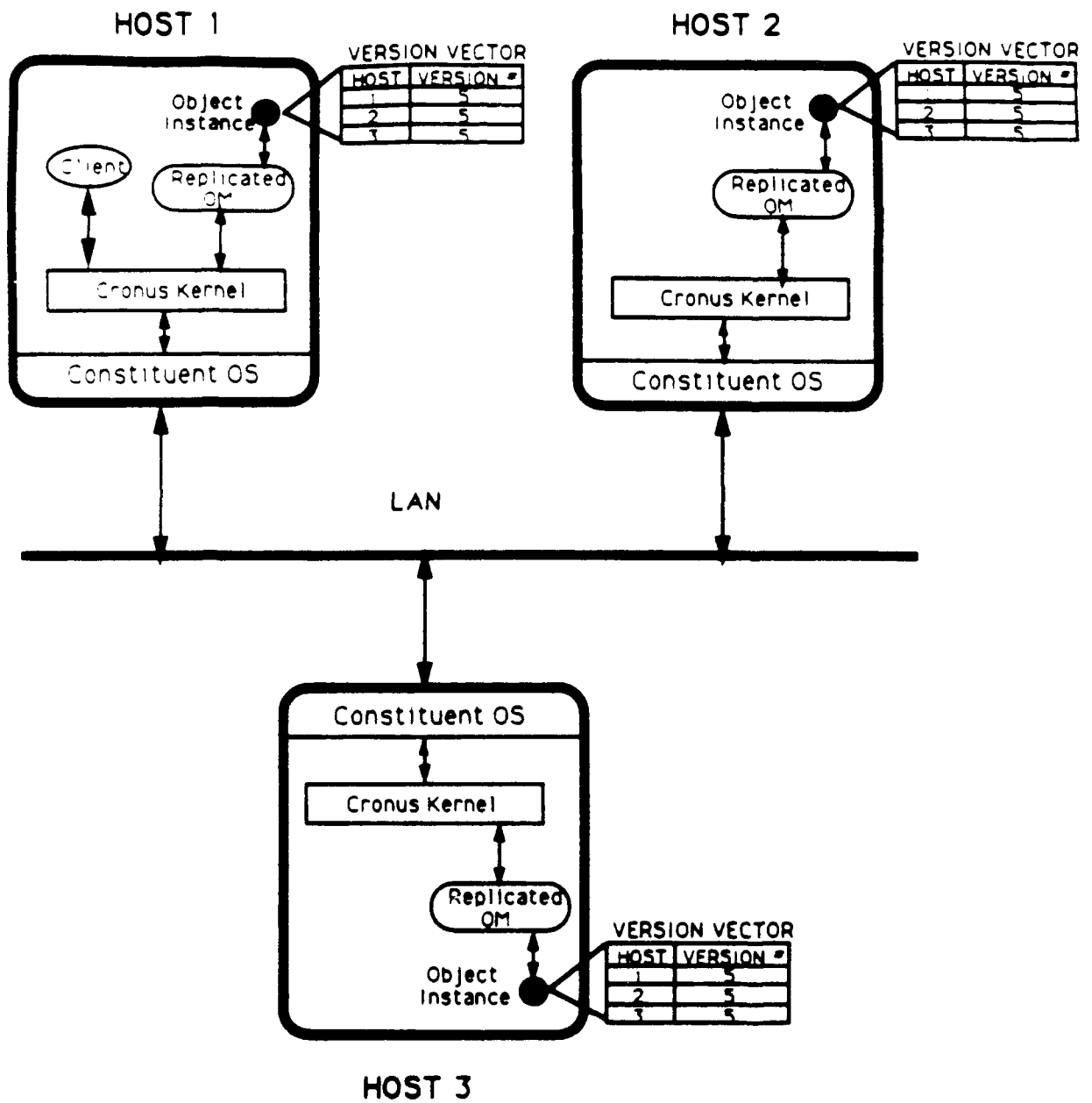


Figure 3.3.2 Version Vectors

correct inconsistencies; it detects them and locks all copies to prevent further inconsistencies. Cronus replication mechanisms detect inconsistencies by comparing the version vectors. Cronus does provide a command to manually repair and unlock replicated copies called "fixobject". To implement the different number of read and write necessary, a simple three line addition to the type definition file can be added.

Object location is implemented in Cronus as an operation on type object. Every object created in Cronus is a subtype of type object. Therefore every object in Cronus including replicated objects will inherit the locate operation from type object. When an object needs to be located, possibly to have an operation invoked upon it, the locator first checks its local object cache. If the location of the object is stored in the object cache the locate does not need to be performed. If the object location is not in the local object cache, the operation switch invokes the kernel locate mechanisms which broadcast the locate operation to all operation switches (Cronus kernels). The operation switches in turn route the locate invocation to the object manager (of the type object to be located) if it exists on its host. If the object is managed by the object manager the host name is returned to the originating locator. All hosts whose object managers manage the object respond. The results of the locate are now stored in the originating locator's object cache to be used when necessary. In our model we obtain read and write access latency data with and without the use of Cronus kernel locate mechanisms. To obtain the read and write data without incurring the overhead of performing object location we simply invoke the appropriate operation on the object prior to the actual test. This insures that the object locations will be stored in the object caches of all Cronus kernels that are participating. To obtain the read write data using the object location mechanisms we use a Cronus command "clear object cache" on all hosts prior to the actual test. We used this command to insure the object locations would not be stored in the kernel's object caches thereby quantifying the additional overhead incurred in performing object location.

3.3.4 Benchmarking Cronus Replication: The high level design of this model consists of an application that invokes the read and write operations on the object manager that maintains and manipulates the object instance. The configurations used for the implementation of our model for both the non-replicated and replicated cases are presented in figures 3.3.3 and 3.3.4 respectively. For the non-replicated case we obtained the average read and write latency time for the application. The operations were invoked on a remote object manager which manages the object instance. This latency time can be used as a baseline for comparison with the replicated case. The replicated case is a bit more complex because of the many possible choices that can be made in tailoring the performance of the replication mechanisms provided within Cronus. As previously mentioned, Cronus provides the application designer with two ways of updating replicated copies: the first is by replacement, and the second is by operation. Another function Cronus has is a locate mechanism; this is used by replication to locate objects. Cronus also can vary read and write quorums. This allows the applications programmer to decide if the data should be consistent or available or a combination of the two. All of the aforementioned replication mechanisms have been discussed in greater detail in section 3.3.3. It is apparent that the design for the replicated case has become quite complex due to the flexibility of replication provided in Cronus. It is important to assess the cost of doing replication in Cronus using as many of the possible combinations of features

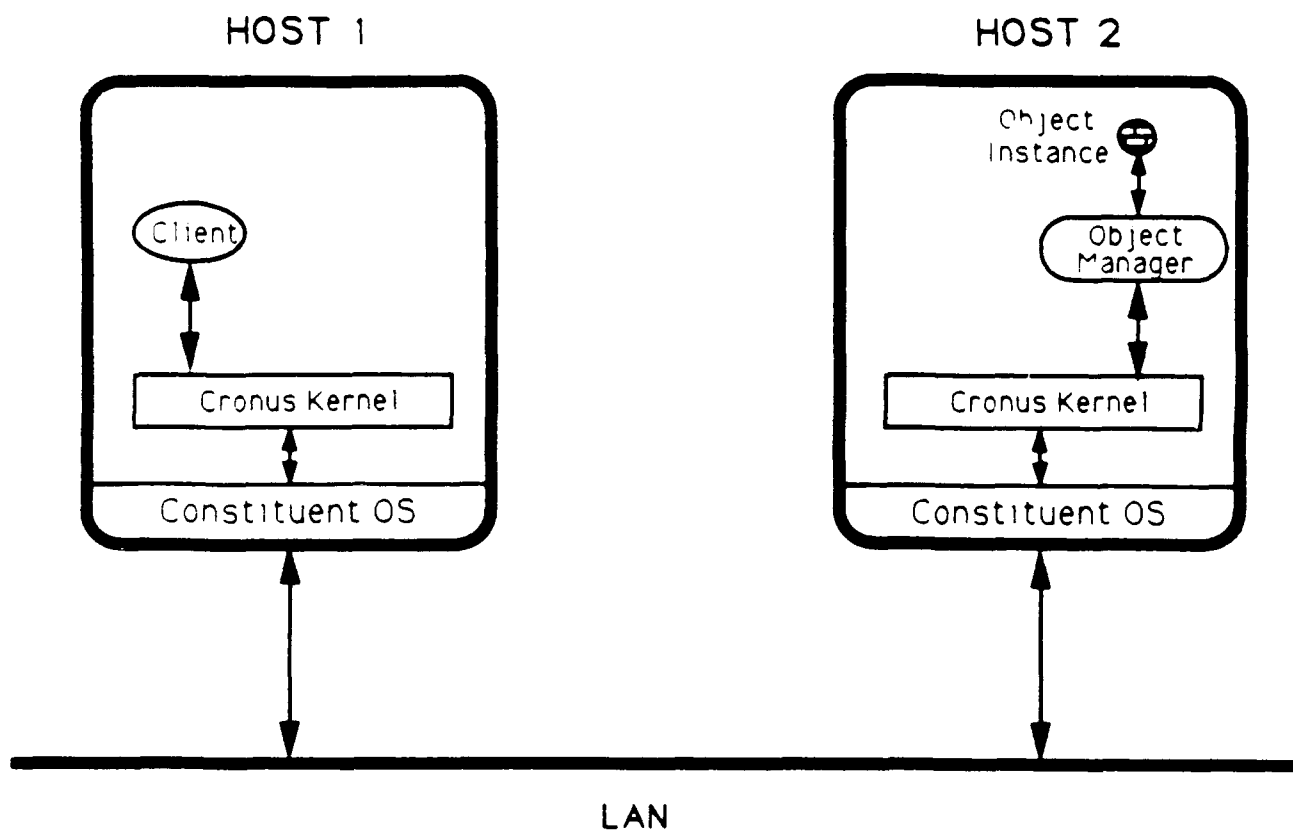


FIGURE 3.3.3 Model For The Non-replicated Case

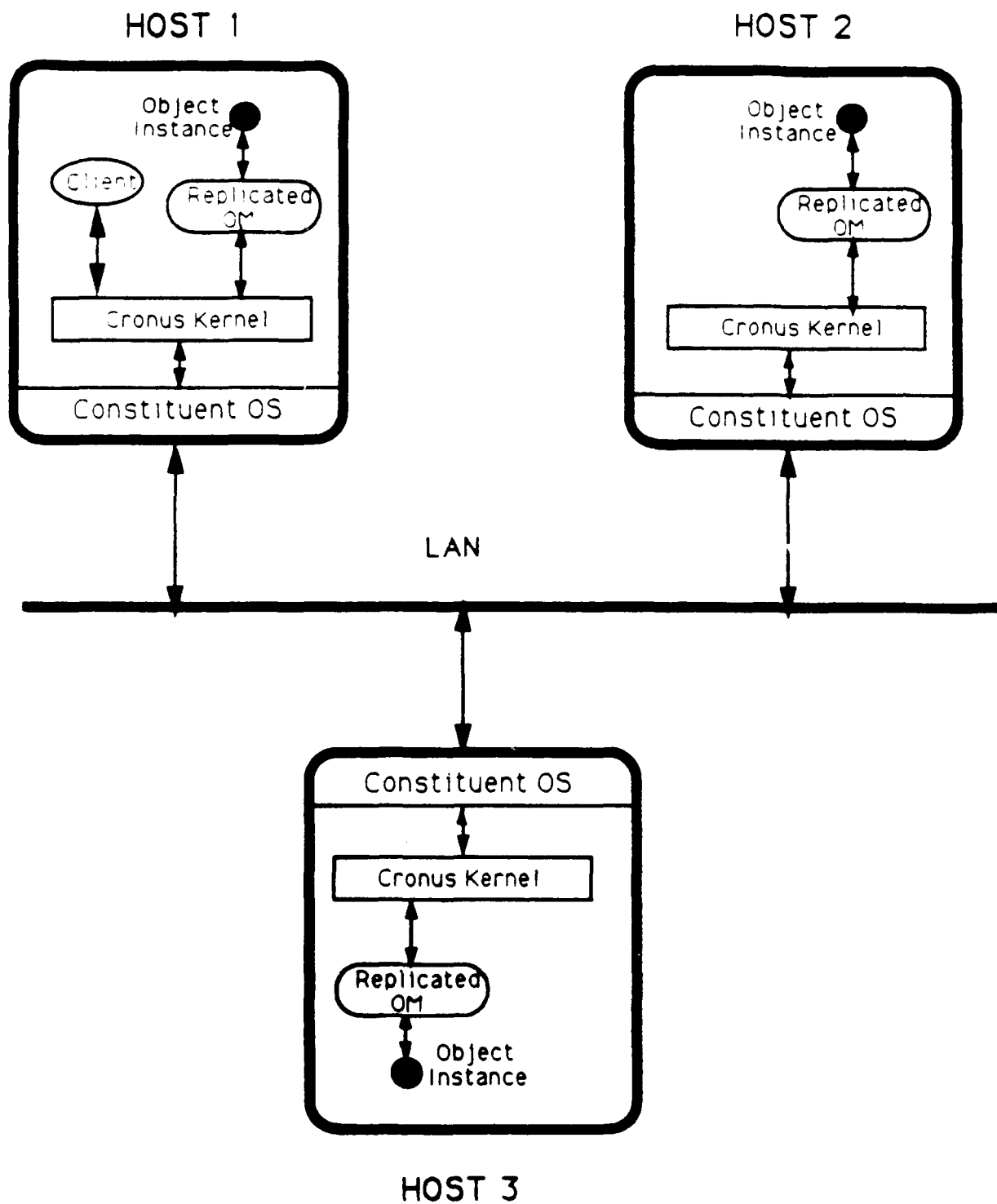


Figure 3.3.4 Model For The Replicated Case

supported. Therefore we design the high level model to obtain read and write latency times for the following cases.

- 1) Every combination of read and write quorums (1-3), using object location mechanisms, and update by operation.
- 2) Every combination of read and write quorums (1-3), without using object location mechanisms, and update by operation.
- 3) Every combination of read and write quorums (1-3), using object location mechanisms, and update by replacement.
- 4) Every combination of read and write quorums (1-3), without using object location mechanisms, and update by replacement.

While the experiments described above will not allow us to make any sweeping statements as to how "good" or "bad" Cronus implements replicated mechanisms, it will allow us to make some qualitative statements as to why certain subsystems in Cronus may be more efficient under certain conditions. We are unable to make any quantitative statements about Cronus until this generic metric is applied to another DOS environment and the results are compared.

3.3.5 Discussion of Results: Figure 3.3.5 contains all data collected during this portion of the evaluation. Essentially all read and write access latency times for various replication strategies are provided. We decided to look at the case where the object defined has been replicated at two other nodes (three copies in all). The data was collected with read and write quorums varied (from 1 to 3) and the mode of update used was both "by operation" and "by replacement" (for more discussion of how replication mechanisms are used in Cronus see section 3.3.3). The data produced does not really provide any results that were surprising. We notice that the overhead increases incrementally as we increase the number of votes that must be collected by the local node's manager (the manager managing the replicated copy on the host from which the read or write invocation originated). This is not necessarily the case as the Cronus kernel may cache the location of the manager on another host and route the invocation there. However, it usually turns out that the local node's manager services request issued locally. Since the ability to successfully perform a write operation is dependent on the read quorum as well as the write quorum in Cronus (we must read the object from the object database before we can write it), the write latency (with the write quorum held constant) will increase as we increase the number of votes necessary to do a read (read quorum). As mentioned previously the Cronus kernel maintains a cache of the locations of recently addressed objects. If the object cache is empty or does not include the address of the object of interest (object type of interest) the kernel uses a location mechanism which broadcasts a message to all the Cronus kernels in the configuration. When the objects of the type requested are located the replication mechanism interactions can take place between the replicated object managers. The data for the read and write latency described above both with and without kernel locate are also included in Figure 3.3.5. There was one result observed that was a bit surprising. The data for the case when we updated remote copies "by operation"

Replication Data					
Update by replacement & by operation (in seconds)					
Read Quorum	Write Quorum	With Locate		Without Locate	
Read	Write	Read	Write	Read	Write
1	1	.15	.24	.04	.19
1	2	.15	.30	.04	.22
1	3	.15	.34	.04	.24
2	1	.25	.31	.15	.23
2	2	.26	.31	.15	.23
2	3	.26	.33	.16	.25
3	1	.27	.34	.19	.24
3	2	.27	.34	.19	.25
3	3	.27	.34	.19	.24
Without Replication:		.04	.06	.03	.04

Figure 3.3.5 Data: Survivability and Availability Section

was nearly identical to the results obtained using "by replacement". To explain what was happening in both cases it was necessary to study the Cronus source code. The interactions that take place in a standard read or write among replicated managers are as follows:

- 1.) After one of the replicated objects has been read or written to the following set of arbitrations occur.
- 2.) `SendVoteRequestsOut`: A vote request is sent out to each of the hosts listed in the version vector associated with the object. If the update strategy is "by replacement" a copy of the new object instance is sent along at this time. If the update strategy is "by operation" the invocation information for the operation invoked locally is sent along at this time.
- 3.) `CollectVotes`: The coordinating manager waits for the number of vote responses corresponding to the appropriate quorum.

If a write was requested the following are executed as well:

- 4.) `SendCommitsOut`: If the coordinator receives a quorum of votes it then sends out commit messages to all the hosts listed in the version vector associated with this object. This is a signal to all hosts that they may restore the new object to the disk or schedule and perform the operation requested if the update strategy is "by replacement" or "by operation" respectively.
- 5.) `BumpVersionVector`: The coordinator now increments its copy of the version vector associated with the object.

The key implementation issue that effects the data is that the coordinator does not wait for a signal from the hosts listed in the version vector after it sends out the commit messages (`SendCommitsOut`). We would as a consequence expect the data to be approximately the same since we cannot measure the differences in overhead for each of the replicated managers to update their copies by replacement versus by operation (Cronus does not wait until this is done). From an application perspective this means that, if we use replication in Cronus and perform a write operation, we cannot be sure that all copies have in fact been updated once we return to the application. This should not be a problem since any inconsistencies should be automatically detected the next time the replicated data is accessed (through the use of version vectors).

3.4 Benchmarking Interprocess Communication (IPC):

It was decided during the evaluation to study the individual components within Cronus to determine not only how they operate but how efficiently they operate in the distributed environment. As an attempt to achieve both goals we decided to look at a standard invocation and response cycle within Cronus. Simply stated we looked at a read operation done on an employee record consisting of name, address, telephone number, age, and weight. The record was maintained by a Cronus object manager on a remote host. We then benchmarked segments of the Cronus source code throughout the invocation/response cycle.

To further illustrate the work done refer to Figure 3.4.1 for the following discussion. First a Cronus application (denoted API.) is used to start the experiment (i.e. invokes an operation on object manager #1 {denoted OM1}). Next OM1 invokes an operation on OM2 to read the employee record given his or her name. This is the point where we begin our benchmarking (i.e. we benchmark a standard invocation/response cycle between two object managers). OM1 forms an invocation message to be sent to OM2 (i.e. message formation, canonical translation, other bookkeeping) and ships the message off to Cronus kernel #1 (denoted K1). Next K1 does message routing and object location (in our case the kernel does not have to do a kernel locate as we force the address to be in its internal object cache). K1 then sends the invocation message to the Cronus kernel on the appropriate host (in our case kernel #2 {denoted K2}). K2 then processes the message to determine where to send the invocation (i.e. to what Cronus entity running on the host). The invocation message is then sent to object manager #2 where the message is translated into the local machines internal data representation. A task is then created internal to OM2 to service the request (perform the requested operation). After OM2 has completed the task it then forms the response message (message formation, canonical translation, etc.) and sends it to the invoking manager (OM1). The response message is processed by kernel #2 and then kernel #1 in very much the same way as the original invocation message. Finally kernel #1 (K1) passes the response message to the appropriate Cronus entity on its host (in our case OM1). Object manager #1 then extracts the information requested from the message structure and translates the information from its canonical forms to the local machine's internal data representation. It is at this point that the benchmarking of the invocation and response cycle is concluded.

The data structure (object definition) read from the object database at the remote site is as follows:

InName:	ASC; (15 octets)
InAddress:	ASC; (21 octets)
InPhone:	ASC; (8 octets)
InAge:	U16I; (2 octets)
InWeight:	U32I; (4 octets)

where ASC refers to the Cronus canonical representation used for character strings, U16I refers to the canonical representation of an unsigned 16 bit integer, and U32I refers to an unsigned 32 bit integer. In our set of benchmarks the total size of the object, from the user's perspective, is 50 octets.

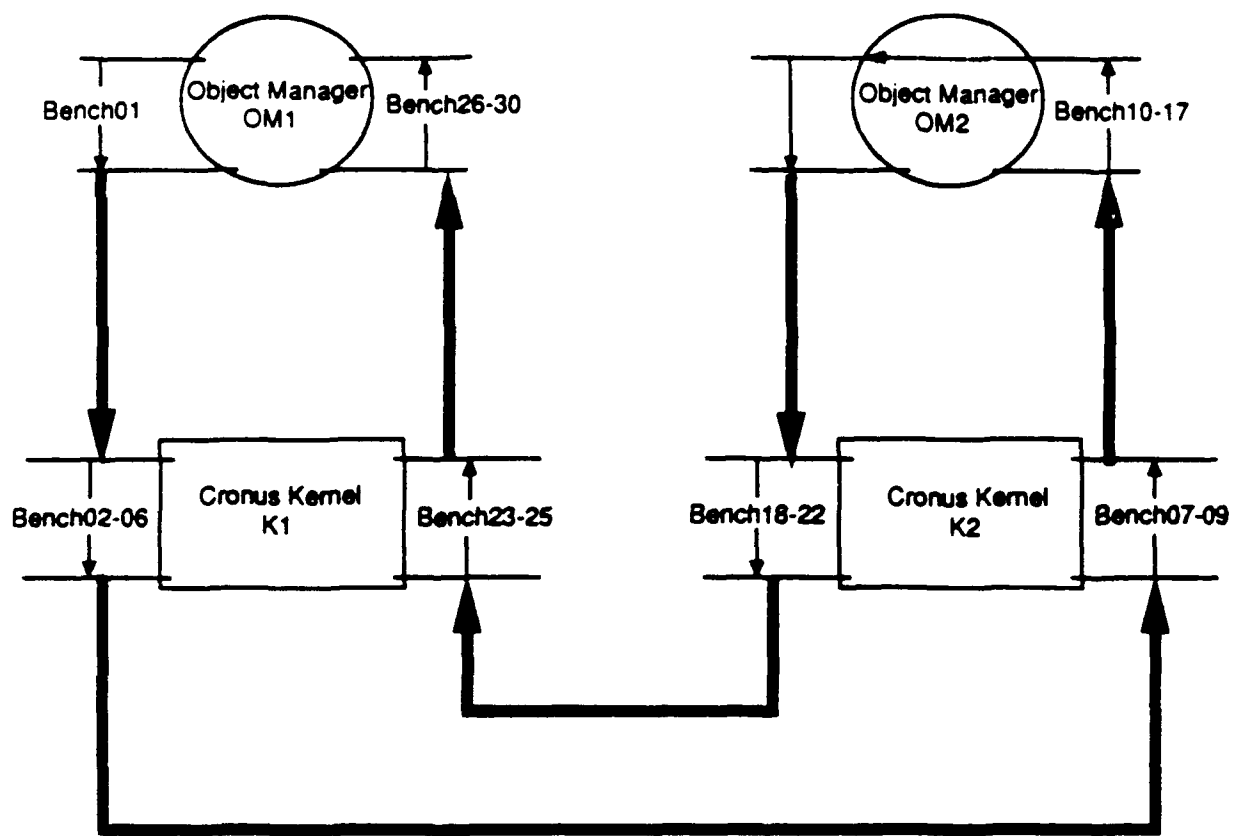


Figure 3.4.1

3.4.1 Results from Benchmarking Interprocess Communication: The results obtained in this section are included in Figure 3.4.2. The data is arranged by benchmark number, the Cronus entity in which the benchmark data is obtained, % time spent in the entity, and % of the overall invocation/response time. It should be noted that all percentages based on overall invocation/response time are taken *without considering the overhead imposed for message transportation between kernels and between object managers and kernels*. A discussion of the effect of transport mechanisms and their overhead will be treated as a separate issue. A discussion of the individual benchmarks follows:

1.) *Bench01:* This benchmark measures the percentage of time spent in forming and processing the invocation message up to the point where object manager #1 (OM1) is about to send the message to the local Cronus kernel (K1).

The manager allocates a message structure, canonically translates the name of the employee, inserts that information into the message structure, and inserts other necessary information into the request message structure (i.e. message type, request identifiers, operation id of operation to be invoked, operation name of operation to be invoked). The manager then calls a routine called Invoke which in turn calls other routines to form and maintain structures used to describe the message to be sent to manager #2 (invocation) (destination host, source host, request type, message type, unique identifier (UID) of the object to be manipulated, message ids, protocols, etc.) It should be noted that this information is canonically translated before being stored in the message structure. This information as well as the data to be provided to the remote manager (name of employee) are sent to the kernel #1 to be routed. The manager then issues a Unix system call (sendto) to send the invocation message to the kernel (via UDP). The measurement stops immediately before the call to Unix.

2.) *Bench02:* This benchmark measures the percentage of time that is spent in kernel #1 to detect an event and determine what type of event has caused it to awaken. Additional processing done before the message is pulled in from the Unix socket buffers.

We begin benchmarking from the point where the kernel has awakened due to some event. In our case this event corresponds to detected file activity in Unix (a message has been sent to a socket managed by the kernel that has a socket (file) descriptor associated with it). The Cronus kernel recognizes three types of events: a message event which represents a message being sent from one process within the kernel to another (e.g. operation switch to object locator), a timeout event which signals the passage of some quanta of time, and a file activity event which represents the detection of file activity (file activity associated with some socket or message port) which indicates that an IPC message has been received from outside the kernel. Items to be processed in the Cronus event queues first start out on a waiting list. The items are essentially waiting to be awakened by some event. The item maintains information on the type of event for which it is waiting as well as other information to distinguish this item from others on the wait queue that are waiting for the same type of event. For example the Cronus kernel must discern between a request for a connection (kernel to kernel connection), a low effort message, and a process request all of which are file

Interprocess Communication Data			
Benchmark	Entity	% Entity	% Overall
Bench01	OM1 (I)	100.0	8.3
Bench02	K1 (I)	14.8	1.0
Bench03	K1 (I)	13.6	0.9
Bench04	K1 (I)	17.2	1.2
Bench05	K1 (I)	51.7	3.6
Bench06	K1 (I)	2.8	0.2
Bench07	K2 (I)	20.1	0.9
Bench08	K2 (I)	22.3	1.0
Bench09	K2 (I)	57.6	2.6
Bench10	OM2 (I)	5.4	2.3
Bench11	OM2 (I)	0.3	0.1
Bench12	OM2 (I)	17.8	7.6
Bench13	OM2 (I)	16.6	7.1
Bench14	OM2 (I)	12.5	5.4
Bench15	OM2 (I)	10.6	4.5
Bench16	OM2 (I)	21.7	9.3
Bench17	OM2 (I)	15.2	6.5
Bench18	K2 (R)	13.8	0.9
Bench19	K2 (R)	13.9	1.0
Bench20	K2 (R)	17.3	1.2
Bench21	K2 (R)	52.1	3.6
Bench22	K2 (R)	2.8	0.2
Bench23	K1 (R)	21.1	0.9
Bench24	K1 (R)	24.8	1.0
Bench25	K1 (R)	54.1	2.2
Bench26	OM1 (R)	9.3	2.5
Bench27	OM1 (R)	0.5	0.1
Bench28	OM1 (R)	31.1	8.2
Bench29	OM1 (R)	27.1	7.2
Bench30	OM1 (R)	32.0	8.5

Figure 3.4.2 Data: Interprocess Communication Section

activity events. In our case the invocation message received by kernel #1 is interpreted as a process request. Next the kernel reads the local request in from the Unix socket buffers. The measurement for this benchmark stops immediately before the call to Unix to ship the message in from the Unix message buffers (Unix *recvfrom*).

3.) *Bench03*: This benchmark measures the percentage of time spent internal to the kernel in categorizing the incoming message and obtaining information about the invoking process.

After storing the invocation message internally (shipped in from Unix socket buffers) kernel #1 first searches through its internal list of known remote processes to determine what process has tried to communicate with it and to fill in a process structure with the vital information it needs to know about the process. The kernel then allocates a new message buffer and associates it with the process file from which the last message (invocation message) was received (preparing for subsequent activity on this port). Finally a pointer to the message and the process structure (invoking process) are passed to the main routine used by the operation switch.

4.) *Bench04*: This benchmark measures the percentage of time spent in determining and setting more information internally about the invoking process (OM1), determining what process the invocation message is bound for, and setting the message up to facilitate the operation switch's routing of the message.

The operation switch portion of the kernel begins to handle the request by canonically translating and copying additional information into the structure describing the invoking process (see *Bench03*) in addition to checking and, if necessary, setting timeout information associated with the message. Next the operation switch tries to determine where the message should go by searching the kernel's internal object cache. If the object address information is not in the cache then the operation switch sends a message to the locator within the kernel. The locator then broadcasts a message to all known kernels to determine the address of the object in question. In our experiments the address of the object is forced to be in the cache so the locator mechanism is not invoked. Next the kernel looks up the host structure describing the destination host and its activity with the operation switch (given the address of the host). Finally the operation switch copies the message into the appropriate buffer associated (internal to the kernel) with a file descriptor. The file descriptor is then marked as though a write had been done to the socket (port) associated with it. This is done so that when the kernel returns to the kernel's main processing loop *activity* on the socket will be detected and control will be passed to the operation switch which will service (route) the message. The measurement for this benchmark ends immediately after returning to the kernel's main processing loop.

5.) *Bench05*: This benchmark measures the percentage of time spent by kernel #1 in detecting the signal generated internally by the operation switch to bring the external request to the attention of the kernel for further processing.

The kernel, now in its main processing loop, waits for any signal that will be trapped as a specific type of event and associated with an item on a queue of items waiting for such an event. At this point in our experiment the only event that has signaled file activity is the one caused internally by the kernel's operation switch (see discussion of Bench04). After the *file* activity has been detected and associated with a wakeup request item that is on the queue of items awaiting events, the kernel shifts the request to the ready queue of requests to be processed by the kernel. The kernel now returns to its main processing loop, pops the new request from its ready queue and invokes the proper kernel process to service the request (in our case the operation switch). The measurement for this benchmark ends immediately after the operation switch has been invoked to service the request.

6.) *Bench06*: This benchmark measures the percentage of time spent by the operation switch in message processing immediately before sending the message out using the local operating system (Unix system call). The operation switch has now been tasked to handle the rest of the processing to be done by the kernel in order to send the invocation on its way to the kernel running on the remote host (the host running OM2). The operation switch first calls a routine that will search through a list of structures that describe the hosts that are associated with a specific active file descriptor (host connected to a socket that is associated with the file descriptor). The operation switch then ships the message to kernel #2 by invoking the Unix system call *write* on the appropriate file descriptor. The measurement for this benchmark ends immediately before the call to the constituent operating system (*write*).

7.) *Bench07*: This benchmark measures the percentage of time spent by kernel #2 in detecting the signal generated by the incoming invocation message and the subsequent processing done by the kernel before invoking the operation switch. The percentage of time spent by the operation switch in processing that occurs immediately before the message is transferred from Unix message buffers to the kernel internal buffers is also measured.

The description of what is happening in kernel #2 is essentially the same as the detailed description of what occurred in kernel #1 for benchmark #2 (see discussion for Bench02). The obvious difference is that the signal detected is being caused by the incoming message from kernel #1. The operation switch within kernel #2 parses certain fields within the request event to determine that the activity at the port was caused by a *reliable message*. Next the structure describing the host that has caused activity at the port is found. The operation switch then checks to see if its kernel (K2) has a message to send. Since it does not, the operation switch determines that a message must be received from the Unix internal socket (port) buffers (this is the incoming invocation message). The message is then transferred into the kernel's internal buffers by invoking a constituent operating system call (Unix *read*). The measurement for this benchmark ends immediately before the Unix system call (*read*).

8.) *Bench08*: This benchmark measures the percentage of time spent by the operation

switch in initially processing the incoming invocation message and in updating statistics pertaining to each host that communicates with the kernel.

After the message is copied into the kernel's (operation switch) internal message structures, initial message validity checks are made and information (such as source host address, destination host address, request type, etc.) is canonically translated and stored in the message buffer. Finally the host information pertaining to the invoking host (host structure mentioned in discussion of Bench07) such as count of messages received from the host, when the last message was received from the host, etc. It is at this point that a new message buffer is allocated for the next message that is received from the host. The kernel then returns to the main operation switch routine so that the message can be further processed and routed. It is at this point that the measurement for this benchmark ends.

9.) Bench09: This benchmark measures the percentage of time spent in the actual routing of the invocation message to the object manager that will ultimately be tasked to service the request.

The operation switch now routes the invocation to the appropriate local process (OM2) by first searching through its hash tables (search done based on the unique identifier for OM2) to obtain the structure that describes the process. With this information the operation switch sends the invocation message (via UDP) to the appropriate local process (OM2) by invoking a constituent operating system call (Unix *sendto*). The measurement for this benchmark ends immediately before the system call.

Before continuing a little more needs to be said about the fundamental structure and operation of the Cronus object manager. The object manager uses a lightweight tasking package to schedule and perform all of its work. There are two permanent tasks that are used most frequently called *idle* and *dispatch* (there are others but we will not concern ourselves with them for now). The *idle* task essentially waits for activity (IPC, timeout) and, in our case, after processing the incoming message, the *idle* task places itself at the end of a list of runnable tasks and does a lightweight contextual swap to allow the next task on the queue to run (in our case this is the *dispatch* task). The *dispatch* task processes new requests made to the object manager (operation invocations) or responses (replies) to outstanding requests made by the OM. If there has been a new request, *dispatch* creates a new task called *invokerequest* which determines on which object the operation is invoked. Next it does an access check, calls the appropriate operation handler routine (to perform the actual operation), and sends the reply back to the invoker.

10.) Bench10. This benchmark measures the percentage of the time spent by object manager #2 in transferring the message internally, decoding the message, and scheduling the next task to be run. This is essentially the time spent (initially) in the *idle* task (ref. discussion above).

The object manager (OM2) is normally waiting (within the *idle* task) for some event to

wake it up. In this case the object manager is awakened by the invocation message sent to its port by the kernel (K2). A *message in progress* structure is found (created before the event occurred) that is used to keep track of vital information associated with the incoming message (hash table search). The manager then checks the status of the message in progress. Since the message is *waiting for receive* the manager transfers the message from the constituent operating system buffers to the object manager's internal message buffer (Unix *recvfrom*). The object manager now begins to decode incoming message by translating header information into local structures (i.e. source address, destination address, priority information, etc.). The rest of the data is then copied into internal structures without translation. The status of the message in progress is now marked as *done not reported* and the manager returns to the *idle* task's main routine. The manager then searches through the list of blocked tasks (those tasks that are waiting for external activity) for the task that is waiting for an external invocation. The task is found and placed at the head of the queue of runnable tasks. Finally the idle task places itself at the end of the runnable tasks queue and a lightweight contextual swap is performed to run the next task at the head of the queue. The measurement for this benchmark ends immediately before the contextual swap.

11.) Bench11: This benchmark measures the percentage of time spent by the object manager performing the light weight contextual swap. It should be noted that there are several contextual swaps that occur in the manager but their effects are included within other benchmarks.

12.) Bench12: This benchmark measures the percentage of time spent in the *dispatch* task within object manager #2 (ref. discussion above).

After the contextual swap (within the manager) the manager is within the dispatch task. The dispatch task now extracts vital information from the incoming message and translates it from its canonical to its internal representation (message type, request identifiers, operation code, etc.). The dispatch task then declares (creates) another task called *invokerequest* (mentioned above) and places the task at the tail of the queue of runnable tasks. Finally the task ships the invocation message information into its internal buffers, adds itself to the list of blocked tasks, and does a lightweight contextual swap to schedule the next task on the queue of runnable tasks. The measurement for this benchmark ends immediately after the contextual swap.

13.) Bench13: This benchmark measures the percentage of time spent back in the *idle* task determining whether or not there has been additional activity (events) since the invocation message had been received.

After the contextual swap the manager is back in the main routine within the *idle* task. The idle task essentially determines whether a task has expired (timeout) and then goes on to try to receive a message that may have been sent to object manager #2 since the invocation message had been processed. In our case there is no message to be processed at this point so the idle task places itself at the end of the queue of runnable tasks and does a lightweight contextual swap to allow the next runnable task to execute within the manager's address space. The measurement for this benchmark

ends immediately after the contextual swap.

14.) Bench14: This benchmark measures the percentage of time spent by the *invokerequest* task in verification of the validity of the request and in the formation of a response message.

The object manager has now scheduled the *invokerequest* task (created by the *dispatch* task). The *invokerequest* task first determines whether or not the object is managed by manager #2. Next the task reads in the invocation message data and determines whether or not the operation is a valid one and if the invoker has the right to request that the operation be performed. The task then creates a reply buffer and initializes certain reply codes. The measurement for this benchmark ends immediately after the reply buffer is allocated and initialized.

15.) Bench15: This benchmark measures the percentage of time spent by the *invokerequest* task in parsing (and translating) the parameters passed for the invoked operation.

The *invokerequest* task now calls a function that begins to parse the requested operation data. This function is unique to the operation requested (i.e. there is a separate parsing function for each operation implemented by the manager). The parsing function essentially searches the request buffer for required (and optional) operation parameters. The parameters are translated from their canonical representation to the machine's internal representation. A reply buffer for the operation (Read) is also initialized in the parse routine. The function that actually implements the operation requested is called from the parse routine next. The measurement for this benchmark ends immediately before this procedure call.

16.) Bench16: This benchmark measures the percentage of time spent by the *invokerequest* task in the actual implementation of the operation (i.e. reading the employee record in from the object manager's database).

We now begin to benchmark the invoked operation itself (Read). This procedure first searches through the object database to find the requested object instance (in our case the employee record that matches the supplied employee name). This is accomplished by searching through the manager's table of object unique identifiers (UID) based on a supplied identifier. Once a UID of an object of that type is found a copy of its *object descriptor* is brought in from the disk (object itself brought in as well) and certain fields are put into an internal format (the descriptor is *internalized*). The actual object data is then translated from the canonical format into the internal data representation that is appropriate for the machine. The measurement for this benchmark ends at this point.

17.) Bench17: This benchmark measures the percentage of time spent by manager #2 (*invokerequest* task) in forming the reply message to be sent to the invoking entity

(manager #1).

The manager now allocates space for the reply message (if necessary), canonically translates the object data fields (employee record), and stores the information in the reply buffer. Finally the *invokerequest* task does some cleaning up and bookkeeping (deallocation of space) and sends the reply message out to the kernel (K2) via UDP. The measurement for this benchmark ends at immediately before the message is sent to the kernel for routing.

18.) Bench18 - Bench25: Each of these benchmarks measure the percentage of time spent within kernel #2 and kernel #1 in routing the reply message to the invoking manager (M1). The activity benchmarked in Bench18 - Bench25 is analogous to the processing done in Bench02 - Bench09 respectively. The obvious differences are that the message is being routed through kernel #2 and then through kernel #1 and the message is a response message rather than an invocation message.

Before continuing a short description of the state of the invoking manager at this point in the invocation/response cycle is in order. After sending out the invocation message, manager #1 immediately executed a *receive reply* Cronus system call (recall that this is being done within the manager's *invokerequest* task). Since the manager must wait for the reply, the receive reply executes a call to *TaskSleep* which places the current task (*invokerequest*) on the queue of *blocked tasks*. When the manager then subsequently receives the reply message the task that requires the reply information is blocked and the permanent manager tasks, *idle* and *dispatch*, are in the queue of *runnable tasks*.

19.) Bench26: This benchmark measures the percentage of time spent by object manager #1 immediately after being signaled by the constituent operating system that an event has occurred (kernel #1 has sent the manager the response message). The manager transfers the message into its internal buffers, partially decodes the message and schedules the next task to be run (*dispatch*). The work done within the manager is analogous to the work done initially in object manager #2 during the invocation (ref. discussion in Bench10).

20.) Bench27: This benchmark measures the percentage of time spent by the object manager performing the light weight contextual swap. It should be noted that there are several contextual swaps that occur in the manager but their effects are included within other benchmarks.

21.) Bench28: This benchmark measures the percentage of time spent in the *dispatch* task within object manager #1.

After the light weight contextual swap the manager is within the *dispatch* task. The *dispatch* task then extracts information from the incoming message and translates it from its canonical to its internal representation (message type, request identifiers, operation code, etc.). Next the *dispatch* task determines that the message is a reply and searches through the items (workers in progress) to determine which blocked task the reply is bound for. The *dispatch* task then unblocks the appropriate blocked

invokerequest task by removing it from the blocked tasks list and places it on the end of the runnable tasks list. Finally dispatch adds itself to the list of blocked tasks and does a lightweight contextual swap to schedule the next task on the queue of runnable tasks. The measurement for this benchmark ends immediately after the contextual swap.

22.) *Bench29*: This benchmark measures the percentage of time spent back in the *idle* task determining whether or not there has been additional activity (events) since the response message had been received. The work done by the idle task is analogous to the work done by manager #2 while processing the invocation message (ref. Bench13 discussion).

23.) *Bench30*: This benchmark measures the percentage of time spent by the *invokerequest* task processing the reply message and presenting the requested information in the proper format.

The *invokerequest* task is now scheduled by the manager. The task next transfers the reply and associated information from the *worker in progress* structures into local structures. The *worker in progress* structures are used to keep track of those tasks that are awaiting a reply message within the manager's queues. After the *worker in progress* entry is freed, the *invokerequest* task translates the reply code sent in the reply message. If the reply code does not signify a successful completion then the error fields within the reply message are presented. Finally the reply information requested by the invoking manager is translated from its canonical representation to its internal representation (i.e. address, telephone number, age, and weight). The measurement for this benchmark ends at this point.

Notice that the overall time spent during the invocation and response cycle was 36.52 milliseconds. This includes the UDP message transmission times (intrahost communication) and the TCP transmission times (interhost communication). The overall time spent within Cronus (i.e. invocation and response without transmission time) was found to be 26.79 milliseconds.

4.0 Overall Remarks and Conclusions: The evaluation was an attempt to characterize and analyze the performance of a distributed operating system called Cronus. Our attention focused on three main areas, namely, computational throughput (concurrent processing capability), survivability and availability, and finally interprocess communication.

In measuring the concurrent processing capability of Cronus (ref. section 3.2 -- **Benchmarking Computational Throughput**) a number of interesting results were obtained. First, it was noted that the throughput, in terms of aggregate Dhrystones/sec., increases as the workload placed on the servers increases. That is to say that as we increase the number of Dhrystones that are to be computed by each server, the overall computational throughput increases dramatically. This is to be expected as, with the increase in load, a greater percentage of the overall time spent in the environment is used to calculate Dhrystones as opposed to communication between client and servers, canonical translations, etc. It appears as though the environment is beginning to behave more like a multiprocessor system than a distributed system. Next we notice from the data that, as we add more hosts to the task, there is always a payoff in terms of computational throughput. Communication overhead does increase, however, as we increase the number of hosts used. There exists a tradeoff between the amount of computation performed per server (Dhrystones/call), the number of servers running per host, and the number of hosts integrated into the system. Finally we noticed that adding more servers (object managers) running per host causes an initial increase in computational throughput up to some maximum level. As we continue to increase this number performance begins to drop off. The range over which performance remains at a maximum level depends on the number of hosts integrated and the load placed on each of the servers (Dhrystones/call). This drop in performance may be attributed to an increasing cost of interaction in the distributed environment (i.e. message formation, canonical translations, etc.). Another possibility is that, with an increase in servers running per host, we may be witnessing the effects of increased constituent operating system overhead in terms of process scheduling, paging, etc. The trends described above manifested themselves in the data collected for the Sun RPC implementation of the benchmark application as well. It should also be noted that the results obtained for the Cronus implementation were, in general, comparable to those obtained for the Sun RPC implementation.

In benchmarking the replication mechanisms within Cronus (ref. section 3.3 -- **Benchmarking Availability and Survivability**) read and write access latency times were obtained for an object while varying the read and write quorums needed to access the data, and the mechanisms used to maintain consistency among replicated objects (update by replacement, and update by operation). While the data obtained allows us to make few global assessments of the performance of the Cronus replication mechanisms, it does allow an application designer familiar with his application and hardware/software environment to estimate the effect/impact using replication may have on the performance of his application. It is interesting to note, however, that there does not seem to be a difference in the data collected for using the two update mechanisms provided in Cronus, namely, update by operation and update by replacement. This can be attributed to the way in which the protocol is implemented in both cases. In Cronus, the coordinator in the aforementioned two phase update protocols does not wait for a signal from the participants listed in the version vector (other managers of replicated objects of this type) after it sends out commit messages (this occurs after a quorum has been achieved). As a consequence, we do not have the opportunity

to measure the amount of time it actually takes the participants to bring their copies to a consistent state. This would lead us to expect that, under most circumstances, the latency data collected would be the same (ref. section 3.3.5). If we made the object size considered very large, however, we would expect that some differences in the data would begin to appear. This would be caused solely by the increase in latency involved in shipping a copy of the object instance to the participant's nodes. It should be noted that, by using this implementation, replication mechanisms in Cronus do not guarantee that consistency is achieved among the replicated copies upon returning to the application (in other words, the commits may be sent out and one or more of the participants may fail). This should not be a problem as inconsistencies would be detected upon accessing the replicated object again (detection is done by comparing version vectors).

In benchmarking Cronus Interprocess Communication (ref. section 3.4) we studied the individual components that are involved in a typical operation invocation in Cronus. This analysis provided data on the percentage of time spent within a distinguishable section of code (based on function) within a Cronus subsystem within the invocation/response cycle as well as the percentage of the overall invocation/response time that was spent within the code section. A detailed analysis of the code executed during this cycle was completed as well (ref. section 3.4.1). It can be seen from the data collected (ref. Figure 3.4.2) that the greatest percentages of overall time spent (invocation and response) were within one of the two Cronus managers involved. In particular, the two Cronus kernels consumed 11% and 11.4% of the overall time while the two Cronus managers (invoker and invokee) consumed 34.8% and 42.8%. We have concluded that most of the time spent within the manager's was in actually running the operations requested. This statement however does not yield useful information about the internal functions associated with a Cronus manager (i.e. lightweight task creation and scheduling, data translations to and from the canonical data types, etc.). In other words we could easily make the operation invoked do very little and, as a consequence, the percentage of time spent in actually executing the operation would decrease. Most of the overhead spent within the manager was in message formation and extraction. Most of this time was spent in translating data to and from Cronus' canonical data representations. Also a great percentage of time was spent in lightweight task creation and management.

It is our overall opinion that Cronus performed quite well in all areas while providing a great number of features that are desirable in a distributed environment. In general we found that benchmarking or studying the behavior of a distributed environment is an exceedingly difficult and time consuming venture. It is not sufficient to run a series of *canned* routines and, based on performance indices, assess the overall capabilities of the system. The software and hardware components that comprise the essence of the distributed environment can be placed in many configurations (each of which can alter the results of any predefined or static benchmark). We tried to keep this in mind during each phase of the evaluation. In short, the rules of common sense apply. No benchmark may be taken in isolation to determine overall system performance in a centralized or a distributed environment.

References:

- [1] G. Popek, and B. Walker, *The LOCUS Distributed System Architecture*, Cambridge, Mass., MIT Press, 1985.
- [2] M.A. Dean, R.M. Sands, and R.E. Schantz, "Canonical Data Representation in the Cronus Distributed Operating System", Proceedings of the IEEE Infocom '87, March 1987, pp. 814-819
- [3] R. Gurwitz, M.A. Dean, and R.E. Schantz, "Programming Support in the Cronus Distributed Operating System", Proceedings of the 6th International Conference on Distributed Computing Systems, May 1986, pp. 486-493
- [4] D.R. Cheriton, W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations", Report No. STAN-CS-83-973, Stanford University, CA., July 1983.
- [5] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "MACH: A New Kernel Foundation for UNIX Development" in Proceedings of USENIX 1986 Summer Conference, pp. 93-112.
- [6] R.F. Rashid, "Threads of a New System", UNIX Review, Aug 1986, pp. 37-49.
- [7] P.A. Bernstein, J.B. Rothnie, N. Goodman, and C.A. Papadimitriou, "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)", IEEE Trans. on Software Engineering, Vol. SE-4, No. 4, May 1978, pp. 113-127.
- [8] B.C. Lindsay, L.M. Hass, C. Mohan, F.F. Wilms, and B.A. Yost, "Computation and Communication in R*: A Distributed Database Manager", ACM Trans. on Computer Systems, Vol. 2, No. 1, Feb. 1984, pp. 24-38.
- [9] M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", IEEE Trans. on Software Engineering, Vol. SE-5, No. 3, May 1979, pp. 188-194
- [10] "Networking on the Sun Workstation: Network File System Protocol Specification", Part No: 800-1342-03, Revision B of 17 February 1986, SUN Microsystems Inc. CA.
- [11] A. Ghafoor, C.Y.R. Chen, and P.B. Berra, "A Distributed Multimedia Database Architecture", Proc. of IEEE Int. Workshop on Distributed Computing Systems in 90's, Hong Kong, Sep. 1988, pp.461-469
- [12] K. Ramamritham, D. Stemple, and S. Vinter, "Decentralized Access Control in a Distributed Systems", Proc. of IEEE Fifth Int. Conf. on Distributed Computing Systems, Denver, Colorado, May 1985, pp. 524-531.

- [13] K.-Y., Whang, and S. Brady, "High Performance Expert Systems - DBMS Interface for Network Management and Control", IEEE Journ. on Selected Areas in Communications, Vol. 7, No. 3, April 1989, pp. 408-417.
- [14] C. Sernada, H. Coelho, and G. Gaspar, "Communicating Knowledge Systems: Part I and Part II- Big Talk among Small Systems", Applied Artificial Intelligence, Vol 1, 1987, pp. 233-260
- [15] V.R. Lesser, and L.D. Erman, "Distributed Interpretation: A Model and Experiment", IEEE Trans. on Comp. Vol. C-29, No. 12, December 1980, pp. 81-99.
- [16] "Internet Transport Protocols", Report X SIS 028112, Xerox Corp., Palo Alto, CA. 1981.
- [17] "Networking on the Sun Workstation: Remote Procedure Call Programming Guide and Remote Procedure Call Protocol Specification", Part No: 800-1342-03, Revision B of 17 February 1986, SUN Microsystems Inc., CA.
- [18] R.P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark", Communications of the ACM, Vol 27, Number 10, Oct 1984, pp. 1013-1030.
- [19] D. Wilson, "Tested Mettle: The Soulbourne 4/601 Workstation", UNIX Review, Vol 7, Number 6, Jun 1989, pp. 105-117
- [20] A.D. Birell, and B.J. Nelson, "Implementing Remote Procedure Calls", ACM Transactions on Computer Systems, Vol. 2, No. 1, Feb. 1984, pp. 39-59.
- [21] R. Schantz, R. Thomas, and G. Bono, "The Architecture of the Cronus Distributed Operating System", Proc. IEEE 6th Int. Conf. on Distributed Computing Systems, Cambridge, Mass., May 1986.
- [22] A.S. Tanenbaum, *Computer Networks*, Prentice-Hall, Inc. Englewood Cliffs, NJ, 1981.
- [23] Northcutt, J.D., *Mechanisms for Reliable Distributed Real-Time Operating Systems - The Alpha Kernel*, Academic Press, Inc., 1987.

Appendix A:

Example type definition file:

```
type NBITest = 311      globally unique type number
  abbrev is nbi
  subtype of Object; type hierarchy description

cantype ITEM
  representation is Item:      persistent state (object)
  record                      defined but not manipulated
    COne:    U32I;
  end ITEM;
```

*the next few lines are definitions for the Dhrystone
calculation operation (INV0).*

```
generic operation INV0
  (Iterations:  U32I;      input parameters. number of
   Tid:  U16I;)           Dhrystones calculated (Iterations)
  returns        and invocation tag (Tid)
  (ResultONE: U32I;      result Flag returned (ResultONE)
   Tidres: U16I);        and invocation tag returned (Tid)

end type NBITest;
```

Note: U32I and U16I refer to canonical data representations used within Cronus for the variables in question (unsigned 32-bit integer and unsigned 16-bit integer respectively).

Example manager definition file:

```
manager "Non-Blocking Invoke Manager"
  abbrev is nbi
```

```
type nbi
  variable representation is ITEM      persistent state (object)
  nbi implements all from nbi      identification of inherited
  obj implements rest              operations (from parents in type
                                   hierarchy).
```

Appendix B:

{ Specification of Client and Server processes for Cronus }

The formal specification of the client and the server processes are given as follows. It can be noted that the client application process essentially uses two communication functions of the Cronus IPC layer which are Invoke and Receive as mentioned previously. Similarly the server process relies on the underlying IPC layer and uses Send and Receive primitives.

**** Specification of the Client Application Process ****

```
procedure ClientApplication(NumOfServers, NumOfNodes, NumOfBenchmarks)
  NumOfServers, NumOfNodes, NumOfBenchmarks : integer;
  NodeNumber, ServerNumber : integer; Success : boolean;
begin

  /*** First take a time hack locally ***/

  StartTime = TimeHack();

  /*** Next send out all invocations ***/

  for ServerNumber = 1 to NumOfServers loop
    for NodeNumber = 1 to NumOfNodes loop
      InvokeServer(NodeNumber, ServerNumber, NumOfBenchmarks);
      NodeNumber = NodeNumber + 1;
    end loop;
    ServerNumber = ServerNumber + 1;
  end loop;

  /*** Next receive all responses ***/
  for ServerNumber = 1 to NumOfServers loop
    for NodeNumber = 1 to NumOfNodes loop
      Success = ReceiveResponse(NodeNumber, ServerNumber);
      NodeNumber = NodeNumber + 1;
    end loop;
    ServerNumber = ServerNumber + 1;
  end loop;

  /*** Take another time hack ***/
  FinishTime = TimeHack();

  /*** Finally calculate the aggregate rate of calculating benchmarks ***/
  BenchMarksPerSecond = (NumOfNodes*NumOfServers*NumOfBenchmarks)/
    (FinishTime-StartTime)
```

```
end ClientApplication;
```

```
function InvokeServer(NodeNumber, ServerNumber, NumOfBenchmarks)
  NodeNumber, ServerNumber, NumOfBenchmarks : integer;
begin
  /* This function forms the invocation message (including any */
  /* canonical translation if necessary) and sends the message out */
  /* to the appropriate server process. This function is very much */
  /* dependent on the distributed environment used (i.e. most */
  /* all of its implementation may be shielded from the user). */
end InvokeServer;
```

```
function ReceiveResponse(NodeNumber, ServerNumber) return Success
  NodeNumber, ServerNumber : integer; Success : boolean;
begin
  /* This function receives (and, if necessary, canonically */
  /* translates) the results sent back from the server processes. */
  /* Once again, how this is done and how much of the implementation */
  /* is left to the user is dependent on the distributed environment */
  /* used. */
end ReceiveResponse;
```

*** *Specification of the Server Process* ***

```
procedure ServerProcess()
  NumOfBenchmarks, BenchMark : integer;
begin
  loop FOREVER
    NumOfBenchmarks = ReceiveMessageAndExtractArguments();
    for BenchMark = 1 to NumOfBenchmarks loop
      ExecuteBenchmark();
    end loop;
    FormAndSendResponse();
  end loop;
end ServerProcess;

function ReceiveMessageAndExtractArguments() return NumOfBenchmarks
  NumOfBenchmarks : integer;
begin
  /* This routine receives the invocation message from the client */
```

```

/* application, extracts the number of benchmarks to be executed */
/* from the message structure, and, if necessary, translates this */
/* information into a representation suitable for interpretation at */
/* the local node. In implementation some or all of this overhead */
/* may be handled transparently by the distributed environment. */
end ReceiveMessageAndExtractArguments;

```

function **FormAndSendResponse()**

```

begin
/* This routine is called to send the appropriate success message to */
/* the invoking client application. This includes formation, canonical */
/* translation (if necessary), and transmission of the message. Some */
/* or most of the above may be handled transparently by the distributed */
/* environment. */
end FormAndSendResponse;

```

{ *Specification of Client and Server processes using Sun RPC*}

The formal specification of the client and the server processes using Sun RPC is given as follows.

Pseudocode for Client Application:

```

procedure RPC_ClientApplication(NumOfServers, NumOfNodes, NumOfDhrystones)
  NumOfServers, NumOfNodes, NumOfBenchmarks : integer;
begin
  RegisterServerProcedure("ProcessResults");
  StartTime = TimeHack();
  for ServerNumber = 1 to NumOfServers loop
    for NodeNumber = 1 to NumOfNodes loop
      CallServer(NodeNumber, ServerNumber, NumOfDhrystones);
      NodeNumber = NodeNumber + 1;
    end loop;
    ServerNumber = ServerNumber + 1;
  end loop;
  BecomeAServer();
end RPC_ClientApplication;

```

function **RegisterServerProcedure**(ProcedureName)

```

  ProcedureName : string;
begin

```

```

/* After all invocations are made by the client application it will */
/* service invocations made by the server processes. The server */
/* processes must invoke operations on the client to register their */
/* results after doing their benchmark calculations. In other words */
/* the client and server switch roles (the client becomes a server */
/* and visa versa). This function registers the name of the procedure */
/* that will be scheduled by the dispatcher when a server registers */
/* its results with the client (Success Flag). The new service is */
/* registered with the local node's portmapper daemon and a TCP socket */
/* is allocated for communication with this new service. */
end RegisterServerProcedure;

```

```

function CallServer(NodeNumber, ServerNumber, NumberOfDhrystones)
  NodeNumber, ServerNumber, NumberOfDhrystones : integer;
begin
  /* This function establishes a TCP connection between the client and */
  /* server, forms a message structure containing canonically translated */
  /* data to be passed to the server (NumberOfDhrystones), and sends the */
  /* message over the TCP connection to the server (dispatcher). */
end CallServer;

```

```

function BecomeAServer()
begin
  /* This function is called after the client application has made all */
  /* necessary invocations and must become a server process in order to */
  /* receive the Success flags sent by the server processes after they */
  /* complete the required number of Dhrystone calculations. This */
  /* function never returns. A dispatcher (infinite loop) waits for */
  /* the invocations and schedules the registered procedure "Process */
  /* Results" to service the invocation. */
end BecomeAServer;

```

```

procedure ProcessResults()
begin
  /* First translate data supplied in the invoking message structure into */
  /* a format suitable for the local node. */

  Success = ExtractAndTranslateMessageData();

  ResultsReceived = ResultsReceived + 1;
  if ResultsReceived = NumOfServers then
    FinishTime = TimeHack();
    DhrystonesPerSecond = (NumberOfNodes*NumberOfServers*
                          NumberOfDhrystones)/(FinishTime-StartTime);

```

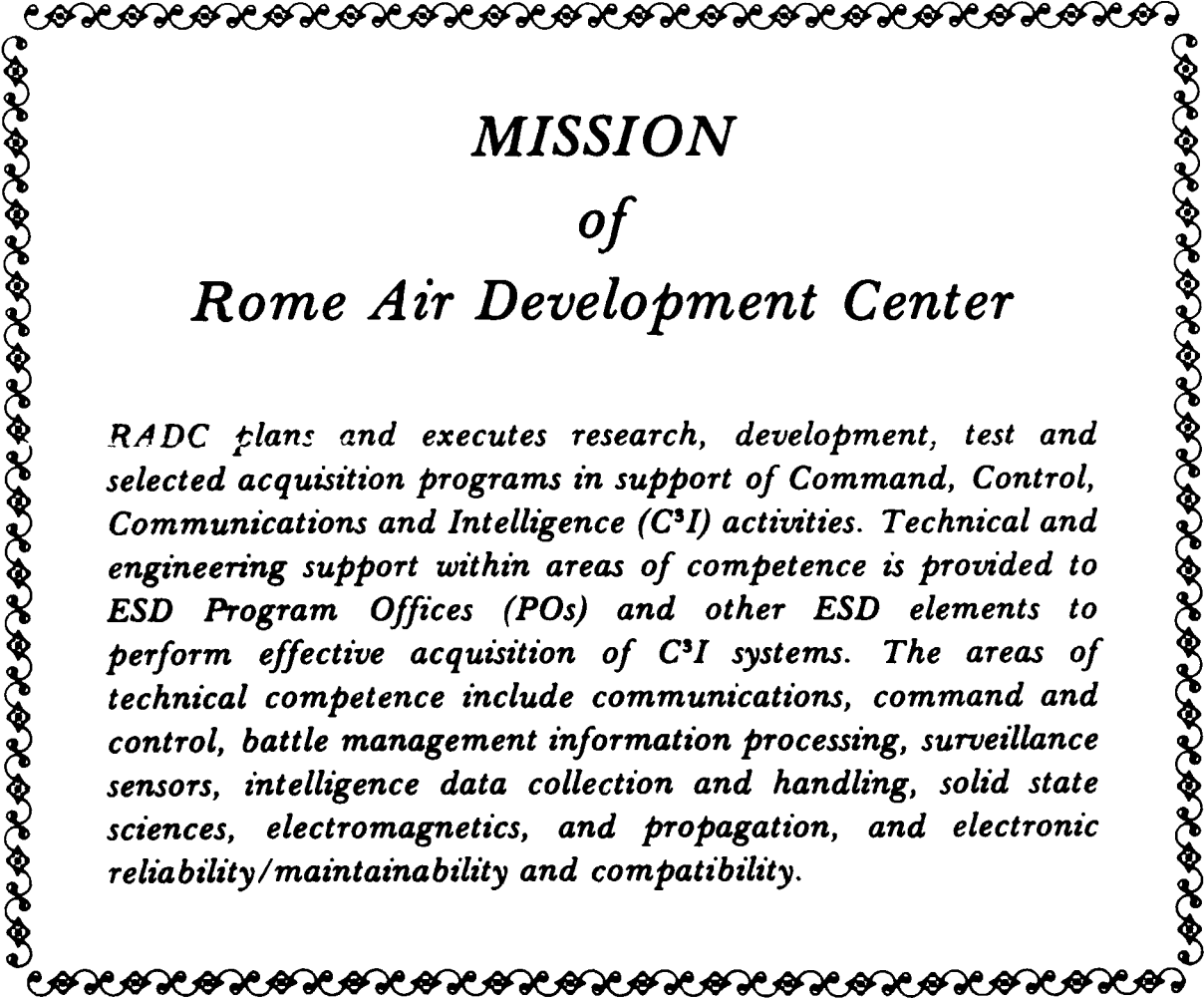
```
endif  
end ProcessResults;
```

Pseudocode for the Server Process:

```
procedure ServerProcess()  
begin  
  /* register the procedure that is to be scheduled upon client invocation */  
  RegisterServerProcedure("ServiceClient");  
  
  /* server's dispatcher takes over (supplied by Sun RPC libraries). The */  
  /* dispatcher runs continuously (in other words we never return from */  
  /* BecomeAServer)and services invocations by scheduling ServiceClient */  
  /* to run in the server's address space. */  
  BecomeAServer();  
  
end ServerProcess;
```

```
procedure ServiceClient()  
begin  
  /* determine the number of Dhrystones to be calculated (from client's */  
  /* invoke message). Number is translated to be compatible with local */  
  /* node's internal representation. */  
  NumOfBenchmarks = ExtractAndTranslateMessageData();  
  for Benchmark = 1 to NumOfBenchmarks loop  
    ExecuteBenchmark();  
    Benchmark = Benchmark + 1;  
  end loop;  
  CallClient(Success);  
end ServiceClient;
```

```
function CallClient(Flag)  
  Flag : boolean;  
begin  
  /* This call creates a local TCP socket, connects it to the client */  
  /* application's socket (client is now really a server), and invokes */  
  /* an operation on the client application to register the result of */  
  /* the requested Dhrystone calculations (Flag). */  
end CallClient;
```

MISSION of Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C³I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.