A Workload Emulator Architecture for Distributed Systems

Marc Abrams, John Arnesen, Alan Batongbacal, Chockanath Chandraseka, and Jay Wang

TR 91-35

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

December 31, 1991

A Workload Emulator Architecture for Distributed Systems*

Marc Abrams[1], John Arnesen[2], Alan Batongbacal[1],
Chockanath Chandrasekar[1], and Jay Wang[1]

[1]Department of Computer Science,
Virginia Tech, Blacksburg, VA 24061-0160
[2]ISSC, Capacity Planning IBM Corporation,
Bethesda, MD 20817

TR 91-35

January 3, 1992

Abstract

This report presents the initial design of a general purpose workload emulator that emulates workstations, terminals, and communication equipment attached to host computers and servers in a distributed computing environment. The workload emulator can be programmed to model the time dependent behavior of a workload for performance evaluation of distributed systems.

1 Introduction

Emulation is often used to evaluate the performance of a complex distributed computing environment which consists of workstations, terminals, and communication equipment connected to one or more host computers. Our objective is to design and build a tool called a Workload Emulator (WE), which is capable of emulating arbitrarily large numbers of heterogeneous workstations, terminals, and

*This project is supported by an IBM Innovative Solutions grant.

communication equipment attached to host computers and servers. Workstations are assumed to have a mouse and use a graphical user interface.

The initial purpose of this project is to build a tool that predicts and evaluates the performance of two environments. The first is a business support system developed by IBM, called ADMIN/EASE (release 6.0)[1]. The ADMIN/EASE application runs on IBM PS/2 model 80's (OS/2) and provides front-end proposal and billing applications to IBM sales staff. ADMIN/EASE users can create and modify maintenance quotes and initiate billing activities for their customers, using a database under CICS in a host computer. The ADMIN/EASE system provides a window-driven graphical interface. The second is to model a mixture of interactive Unix terminal sessions and clients on Unix workstations that access server machines. In this documentation a *target system* refers to the system to be emulated.

There are six objectives in the design of the Workload Emulator:

1. *Scalable*: There must be no inherent limit on the size (e.g., the number of workstations) of the configuration that it can emulate.
2. *Modular*: The Workload Emulator must be modular so that program modules emulating equipment from any vendor using any operating system and communication protocols can be added. The Workload Emulator will be built using the object model and C++ to achieve modularity.
3. *Adaptable*: The Workload Emulator must be adaptable such that if the target system to be emulated is upgraded, the Workload Emulator still works with the upgraded system without any or at least no major modifications.
4. *Programmable*: The Workload Emulator must be programmable through a scripting language. A *script* is an input to the Workload Emulator, which defines configuration of emulated system and instances of workload, for example, number and interconnection of workstations, number of user sessions, and sequence of screens that each user transmits in a session.
5. *Unobtrusive*: The Workload Emulator must help a user to formulate a workload model by monitoring one or more workstations and terminals in the system to be emulated to collect statistics on, for example, how frequently users enter different commands or use different application screens, and how long users think before entering. The Workload Emulator must not alter the workload being observed.
6. *Versatile*: The Workload Emulator can be used with different user-end operating systems, application programs, and communication protocols.

The rest of this documentation is organized as follows. Section 2 decides what an ideal emulator would provide. Section 3 examines the technical problems in

The rest of this documentation is organized as follows. Section 2 decides what an ideal emulator would provide. Section 3 examines the technical problems in building this ideal emulator. Section 4 defines the emulator that we will initially build.

2 Functions of an Ideal System

Two design alternatives for the Workload Emulator are the following:

1. Run the actual application programs from the target system on the Workload Emulator.
2. Program the Workload Emulator to generate and decode the same bitstreams that are introduced into and removed from the network(s) to which the target system is attached.

The first alternative is simple to build, but has two severe disadvantages: (1) Executing each application program requires certain computer resources, such as memory, CPU, and disk space, thereby limiting the number of user programs that may be executed by the emulator simultaneously. This leads to a bound on the number of users that can be emulated. (2) The user application programs may not be able to run on the Workload Emulator due to the differences in machine architectures and operating systems. This is true especially when the target system involves heterogeneous workstations.

We use alternative (2), however it does necessitate the learning facility described below. To achieve the objectives discussed in the first section, the Workload Emulator has to perform the following functions:

1. *Learning*: In a distributed system where users communicate with host computer(s) or server(s) by a network, when users enter commands or data from their workstations, the commands or data are converted into a set of *bitstreams*. These bitstreams are then transferred to the host computer(s) or server(s) through the network. Our Workload Emulator must be able to *learn* about the mapping of user keystrokes and mouse selections (for systems with graphical user interfaces) to the bitstreams and be able to produce such bitstreams.

   However, except for a few cases (such as remote terminal emulation in which the bitstreams are basically character streams with only X-on and X-off protocols), learning the mapping function of an application program (e.g. an X windows application) is usually not straightforward. Learning may need to proceed iteratively between the evaluation and validation phases. In the

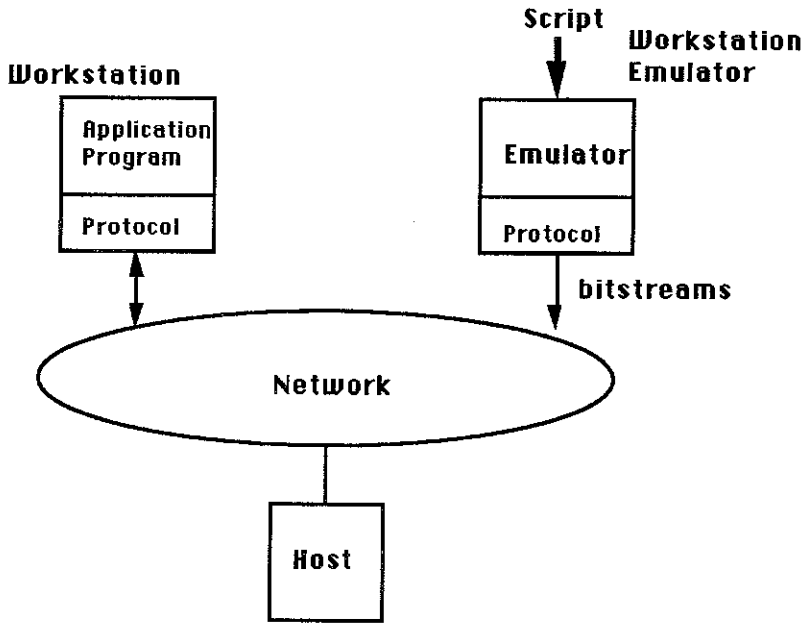

Figure 1: System Configuration

scripts. If errors are introduced by the Workload Emulator, we return to the evaluation phase and re-calculate the mapping function through additional learning and then test the resulting function again. The procedure proceeds until a *reasonable* mapping is achieved[1].

2. *Monitoring a Network*: To help characterize the workload of a target system, the Workload Emulator should be able to monitor the network and collect statistics (e.g., frequency of commands) of user activities.

3. *Generating Workloads*: The Workload Emulator produces a set of bitstreams based on a script that specifies the workloads of the system, provided by the user of the Workload Emulator and is based on the statistics from step 2. The Workload Emulator then sends the corresponding bitstreams to the host as if they were the tasks generated by the workstations and terminals of the real system. This function is illustrated in Figure 1.

[1]For a complex system, an *exact* mapping function may not be obtainable. However, system performance evaluation is indicated by the average of the statistics. Therefore an approximate mapping, which occasionally introduces errors, may suffice.

3 Approaches and Problems

As discussed in the last section, determining how user keystrokes and mouse selections that drive an application are mapped into a bitstream is usually not straightforward. Leaning a bitstream mapping is thus challenging and requires the theory of machine learning in the area of Artificial Intelligence. Many machine learning methodologies are discussed in literatures. Some methodologies [2] are listed below.

Learning

1. *Learning by Example*: In this method, the domain application is run repeatedly with different combinations on its input values. The solution to the problem is established through some analysis on the serial inputs and outputs. For example, the ADMIN/EASE application or commands in a Unix terminal session is executed for a number of times with different field values and the resulting bitstreams are examined by the Workload Emulator to determine the mapping function.
2. *Learning by Rote*: This is a simple kind of machine learning. Here the machine records all data and situations it comes across and applies it when the same or similar situation arises. The system tries to solve the problem and when it cannot go farther, it applies its static evaluation function to the problem and continues from there. In the ADMIN/EASE case, we can execute for a number of times and store all the situations and results. Later the same results can be emulated in similar situations. In contrast an X-window session may permit too rich a variety of behaviors to permit rote learning.
3. *Concept Based Learning*: Begin with a structural description of one known instance of the concept. Call that description the concept definition. Examine descriptions of other known instances of the concept. Generalize the definition to include them. Examine descriptions of near-misses of the concept. Restrict the definition to exclude these. Thus we can approximately form a structure description of the mapping and revise it as we go through the examples.
4. *Learning by Analogy*: Analogy is a powerful inference tool. It allows similarities between objects to be stated succinctly. It is finding correspondences between aspects of two situations. Assume that knowledge about objects is represented in a collection of frames. Then learning by analogy can be described as the transfer of values from the slots of one frame (the source)

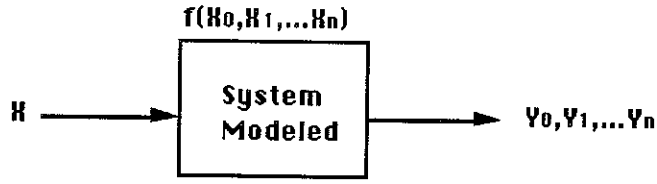

Figure 2: Bitstream Conversion

to the slots of another (the target). This transfer is done in two steps: (1) Use the source frame to generate proposed slots whose values can then be transferred to the target frame. (2) Use the existing information in the target frame to filter the analogies proposed in the previous step. If we know the mapping of one frame in our problem, we can use this technique and compare with other frames, in the subsequent executions.

Problems

Several questions may arise from the learning methodologies discussed above:

1. How can the Workload Emulator observe user key strokes from a particular workstation?
2. Can the Workload Emulator passively observe another machine and learn everything?
3. Is it always possible to estimate the mapping function, say f(x), with a finite number of observations of input and output (Figure 2)? For example, obtaining the mapping function appears impossible if f(x) encrypts x, or finds the root of a function.
4. Observing the bitstreams, how does the Workload Emulator determine the boundaries between data items sent by the application?
5. What restrictions will we need to permit learning? For example, if the mapping function permutes its arguments in using a deterministic rule, the Workload Emulator could learn the function.[2]
6. How much information do we need from the software designer of the application program? For example, does the program encrypt or optimize the data? If yes, are the algorithms available? What separates the fields in a bitstream?

[2]A deterministic mapping appears to be the case for ADMIN/EASE and Unix terminal session.

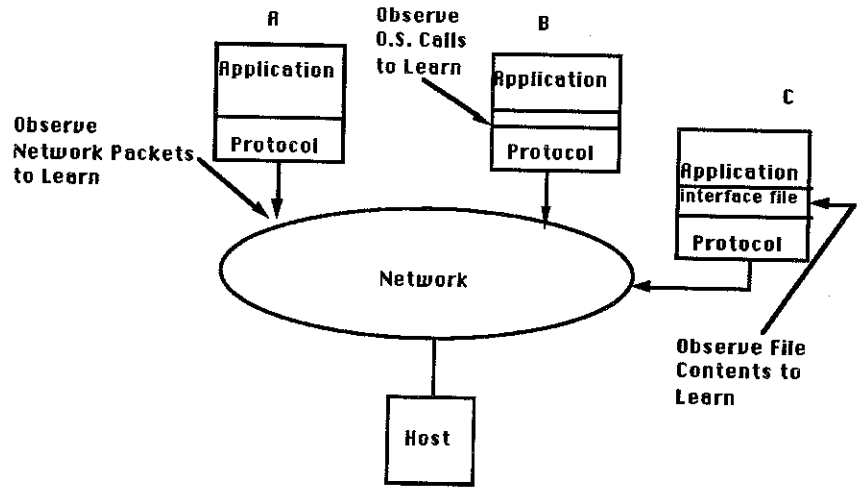

Figure 3: Three Approaches of Collecting Bitstreams

In collecting the bitstreams, basically there are three approaches (Figure 3). That is, we can tap the wire at the physical network data links (method A), listen to the service calls made by the application program to the operating system or protocol (method B) by instrumenting the operating system, or collect the bitstreams from any intermediate data storage (e.g., a file) created by the application program (method C). Among these, method C is the easiest because it need not deal with operating system calls or network protocols. However, method C only applicable to the cases that an intermediate data storage exists. For others, namely method A and B, each of them has its advantages and drawbacks. Basically, one's advantage is the disadvantage of the other. The following are some pro's and con's of method B.

1. We need to deal with only application-generated bitstreams and need not worry about the data appended to the bitstreams by the communication protocol. This simplifies the learning and the generation of bitstreams, because the Workload Emulator need not learn how the protocol functions.
2. Actual calls to the operating system are visible, which is what should be reproduced in the emulator.
3. Some existing tools (e.g., *truss* which lists the system calls used and the actual parameters for a run of a program, available under Amiga UNIX System V Release 4) provide functions to monitor system calls.

If a tool like *truss* is not available then there are some problems in this approach:

1. We need a special C library that can monitor on every platform.
2. We need to re-link the application program to a special C library unless a dynamic library (which is present on OS/2, and SVR4) is offered in the operating system.
3. For the above reasons, the source code to C library may be required.

In the example of ADMIN/EASE, however, bitstreams are stored in a system created file called *request.blk*. For every request sent to the host, the corresponding bitstream is stored in this file, and then a lower level network file service is invoked to transfer this file to a host. In this case, the bitstreams can be collected directly from request.blk.

4 Initial Implementation

The workload emulator consists of three major modules. They are, a *learning module*, an *emulation module*, and a *monitoring* module. Given a system, the learning module determines the bitstream mapping function, and the emulation module converts a user workload script into bitstreams based on this mapping function. The monitoring module analyzes the traffic on the network to characterize the workload of the target system. We divide the development of this workload emulator into two phases. In the first phase, we develop the emulation module. Building the learning module and the monitoring module involves issues in different domains, namely artificial intelligence and low level communication protocols; thus the development of the learning module and the monitoring module is deferred to the second phase.

An emulating module of the Workload Emulator is now under-construction on IBM RS/6000 using C++. The emulating module includes the following.

1. Bitstream Specification: A bitstream specification is required for the Workload Emulator to generate outgoing bitstreams and decode the incoming bitstreams. In our initial implementation, bitstream specification is controlled by hand rather than by learning in the first phase. A bitstream specification language is defined to provide a way to specifying bitstream formats for different applications.
2. Scripting Language: A C++ based scripting language is defined. This choice makes the integration of the components of the emulator easier since the emulator itself is built in C++.

3. Statistics Representation: The Workload Emulator maintains a time stamped log of when bitstreams are sent to or received from a host, along with information describing the bitstreams. For the ADMIN/EASE sessions and Unix terminal sessions, the log keeps the round trip delay of each CICS transaction and the response time of each Unix command respectively.

4. Target Applications: In our initial implementation, the target systems of the Workload Emulator include the followiing: (1) SNA-based ADMIN/EASE systems with an IBM 370 as a server and with file transfer as application protocol (2) TCP/IP based interactive Unix terminal sessions with Unix file server and NFS and with socket as application protocol.

5. Graphical User Interface: The Workload Emulator provides a graphical user interface using X windows. A prototype of the graphical user interface of the Workload Emulator is implemented in Nextstep, an interface building environment for the NEXT.

In our initial implementation, the workload emulation is handled by a single RS/6000. In order to meet the objective of scalability, in the latter phase multiple loosely synchronized RS/6000s shall be used in parallel to divide the workload.

5 Concluding Remarks

The initial purpose of our Workload Emulator is to emulate the IBM ADMIN/EASE system. To make this project more valuable to research work in system performance evaluation, our goal is to build a tool which is adaptable to any system. However, the implementation becomes more complex as the requirement of the adaptability and the complexity of the target system increases (Figure 4a).

To make an emulator more adaptive, there should be less correlation between the emulator and the target system. Hence, in this case, learning plays a more important role and more understanding about the emulated system is required (Figure 4b, 4c).

References

[1] *ADMIN/EASE Release 6.0 Design Document*, IBM, 1989.

[2] E. Rich. *Artificial Intelligence.* Mcgraw-Hill, 1983.

[3] C. Sauer, M. Chandy. *Computer System Performance Modeling.* Prentice Hall. 1981.

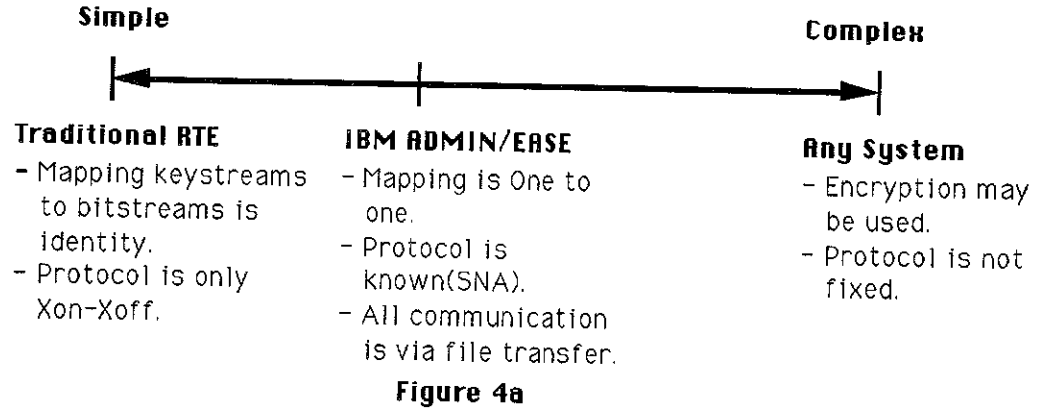

Figure 4a

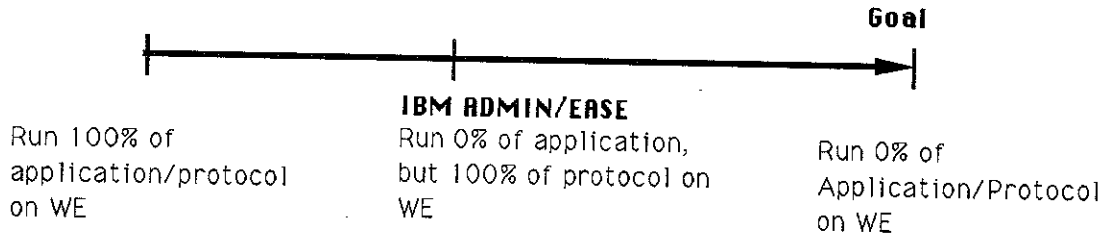

Figure 4b

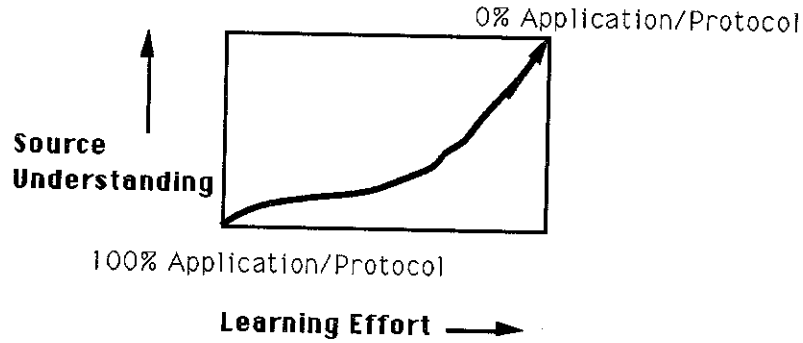

Figure 4c

Figure 4: A Scale of Complexity