The "Worm" Programs—Early Experience with a Distributed Computation

John F. Shoch and Jon A. Hupp
Xerox Palo Alto Research Center

I guess you all know about tapeworms. . . ? Good. Well, what I turned loose in the net yesterday was the . . . father and mother of all tapeworms. . . .
My newest-my masterpiece-breeds by itself. . . .
By now I don't know exactly what there is in the worm. More bits are being added automatically as it works its way to places I never dared guess existed. . . .
And-no, it can't be killed. It's indefinitely self-perpetuating so long as the net exists. Even if one segment of it is inactivated, a counterpart of the missing portion will remain in store at some other station and the worm will automatically subdivide and send a duplicate head to collect the spare groups and restore them to their proper place.
—John Brunner, *The Shockwave Rider*
Ballantine, New York, 1975

SUMMARY: The "worm" programs were an experiment in the development of distributed computations: programs that span machine boundaries and also replicate themselves in idle machines. A "worm" is composed of multiple "segments," each running on a different machine. The underlying worm maintenance mechanisms are responsible for maintaining the worm—finding free machines when needed and replicating the program for each additional segment. These techniques were successfully used to support several real applications, ranging from a simple multimachine test program to a more sophisticated real-time animation system harnessing multiple machines.

1. Introduction

In *The Shockwave Rider*, J. Brunner developed the notion of an omnipotent "tapeworm" program running loose through a network of computers—an idea which may seem rather disturbing, but which is also quite beyond our current capabilities. The basic model, however, remains a very provocative one: a program or a computation that can move from machine to machine, harnessing resources as needed, and replicating itself when necessary.

In a similar vein, we once described a computational model based upon the classic science-fiction film, *The Blob*: a program that started out running in one machine, but as its appetite for computing cycles grew, it could reach out, find unused machines, and grow to encompass those resources. In the middle of the night, such a program could mobilize hundreds of machines in one building; in the morning, as users reclaimed their machines, the "blob" would have to retreat in an orderly manner, gathering up the intermediate results of its computation. Holed up in one or two machines during the day, the program could emerge again later as resources became available, again expanding the computation. (This affinity for nighttime exploration led one researcher to describe these as "vampire programs.")

These kinds of programs represent one of the most interesting and challenging forms of what was once called *distributed computing*. Unfortunately, that particular phrase has already been co-opted by those who market fairly ordinary terminal systems; thus, we prefer to characterize these as *programs which span machine boundaries* or *distributed computations*.

An earlier version of this paper was prepared for the Workshop on Fundamental Issues in Distributed Computing, ACM/SIGOPS and ACM/SIGPLAN, Pala Mesa Resort, December 1980.
Authors' present address: J.F. Shoch and J.A. Hupp, Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

CR Categories and Subject Descriptors: C.2.4 and C.2.5 [**Computer Communication Networks**]: Distributed Systems and Local Networks.
General Terms: Design, Experimentation.
Additional Key Words and Phrases: multimachine programs, Ethernet local network, Pup internetwork architecture.

In recent years, it has become possible to pursue these ideas in newly emerging, richer computing environments: large numbers of powerful computers, connected with a local computer network and a full architecture of internetwork protocols, and supported by a diverse set of specialized network servers. Against this background, we have undertaken the development and operation of several real, multimachine "worm" programs; this paper reports on those efforts

In the following sections, we describe the model for the worm programs, how they can be controlled, and how they were implemented. We then briefly discuss five specific applications which have been built upon these multimachine worms.

The primary focus of this effort has been obtaining real experience with these programs. Our work did not start out specifically addressing formal conceptual models, verifiable control algorithms, or language features for distributed computation, but our experience provides some interesting insights on these questions and helps to focus attention on some fruitful areas for further research.

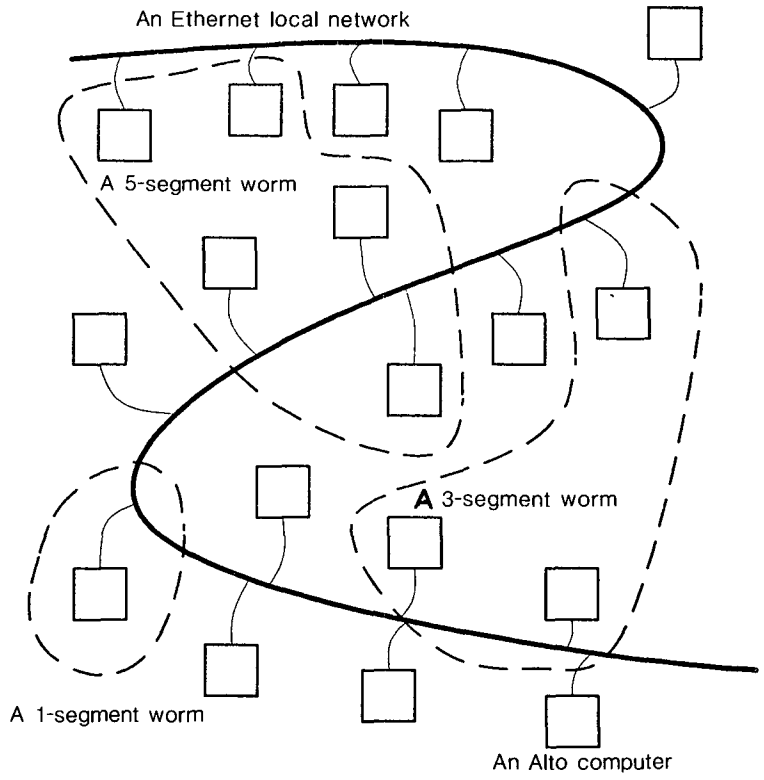

Fig. 1. Schematic of Several Multisegment Worm Programs.

2. Building a Worm

A *worm* is simply a computation which lives on one or more machines (see Figure 1). The programs on individual computers are described as the *segments* of a worm; in the simplest model each segment carries a number indicating how many total machines should be part of the overall worm. The segments in a worm remain in communication with each other; should one segment fail, the remaining pieces must find another free machine, initialize it, and add it to the worm. As segments (machines) join and then leave the computation, the worm itself seems to move through the network. It is important to understand that the worm mechanism is used to gather and maintain the segments of the worm, while actual user programs are then built on top of this mechanism.

Initial construction of the worm programs was simplified by the use of a rich but fairly homogeneous computing environment at the Xerox Palo Alto Research Center. This includes over 100 Alto computers [10], each connected to an Ethernet local network [4, 6]. In addition, there is a diverse set of specialized network servers, including file systems, printers, boot-servers, name-lookup servers, and other utilities. The whole system is held together by the Pup architecture of internetwork protocols [1].

Many of the machines remain idle for lengthy periods, especially at night, when they regularly run a memory diagnostic. Instead of viewing this environment as 100 independent machines connected to a network, we thought of it as a 100-element multiprocessor, in search of a program to run. There is a fairly straightforward set of steps involved in building and running a worm with this set of resources.

2.1 General Issues in Constructing a Worm Program

Almost any program can be modified to incorporate the worm mechanisms; all of the examples described below were written in BCPL for the Alto. There is, however, one very important consideration: since the worm may arrive through the Ethernet at a host with no disk mounted in the drive, the program must not try to access the disk. More important, a user may have left a disk spinning in an otherwise idle machine; writing on such a disk would be viewed as a profoundly antisocial act.

Running a worm depends upon the cooperation of many different machine users, who must have some confidence in the judgment of those writing programs which may enter their machines. In our work with the Alto, we have been able to assure users that there is not even a disk driver included within any of the

worm programs; thus, the risk to any spinning disk is no worse than the risk associated with leaving the disk in place while the memory diagnostic runs. We have yet to identify a single case in which a worm program tried to write on a local disk.

It is feasible, of course, for a program to access secondary storage available through the network, on one of the file servers.

2.2 Starting a Worm

A worm program is generally organized with several components: some initialization code to run when it starts on the first machine; some initialization when it starts on any subsequent machine; the main program. The initial program can be started in a machine by any of the standard methods, including loading via the operating system or booting from a network boot-server.

2.3 Locating Other Idle Machines

The first task of a worm is to fill out its full complement of segments; to do that, it must find some number of idle machines. To aid in this process, a very simple protocol was defined: a special packet format is used to inquire if a host is free. If it is, the idle host merely returns a positive reply. These inquiries can be broadcast to all hosts or transmitted to specific destinations. Since multiple worms might be competing for the same idle machines, we have tried to reduce confusion by using a series of specific probes addressed to individual machines. As mentioned above, many of the Altos run a memory diagnostic when otherwise unused; this program responds positively when asked if it is idle.

Various alternative schemes can be used to determine which possible host to probe next when looking for an additional segment. In practice, we have employed a very simple procedure: a segment begins with its own local host number and simply works its way up through the address space. Figure 2, an Ethernet source-destination traffic matrix (similar to the one in [8]), illustrates the use of this procedure. The migrating worm shows up amid the other network traffic with a "staircase" effect. A segment sends packets to successive hosts until finding one which is idle; at that point the program is copied to the new segment, and this host begins probing for the next segment.

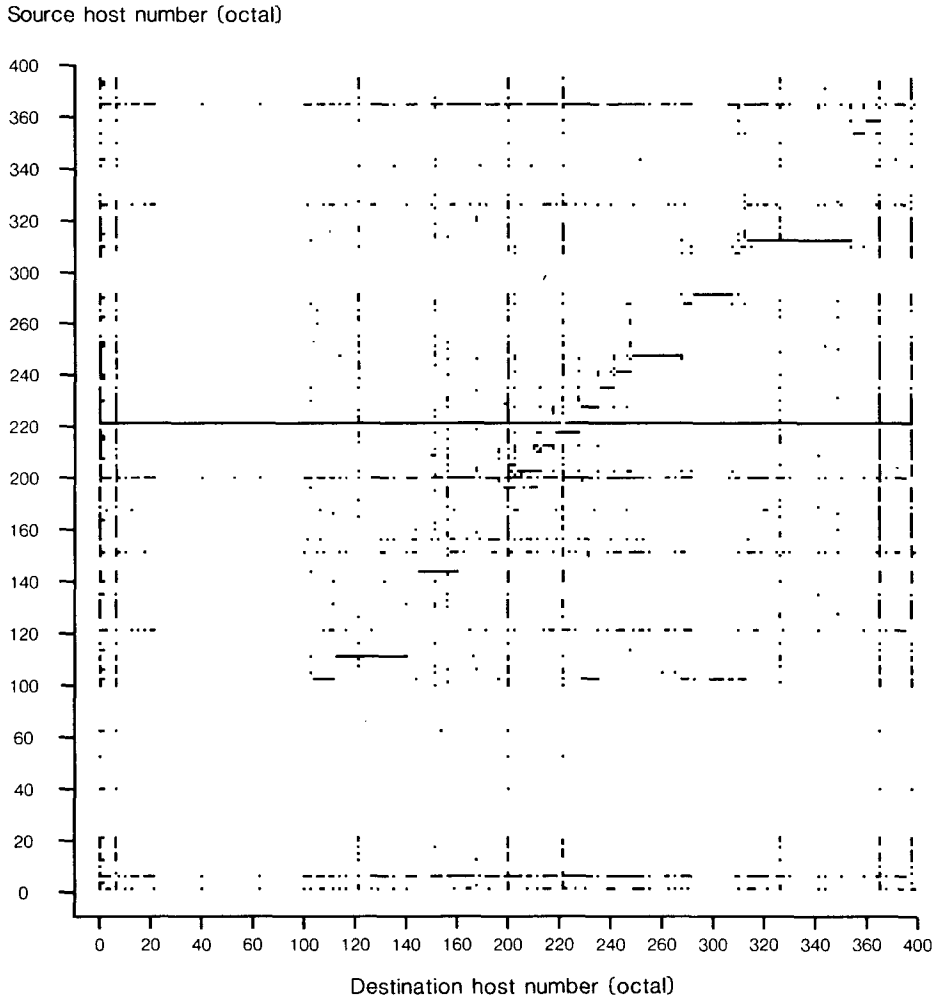

Fig. 2. Ethernet Source-Destination Traffic Matrix with a "Worm" Running. (Note the "staircase" effect as each segment seeks the next one.)

2.4 Booting an Idle Machine

An idle machine can be located through the Ethernet, but there is still no way in which an Alto can be forced to restart through the network. By design, it is not possible to reach in and wrench away control from a running program; instead, the machine must willingly accept a request to restart, either by booting from its local disk or through the network.

After finding an idle machine, a worm segment then asks it to go through the standard network boot procedure. In this case, however, the specified source for the new program is the worm segment itself. Thus, we have this sequence:

(1) Existing segment asks if a host is idle.
(2) The host answers that it is.
(3) The existing segment asks the new host to boot through the network, from the segment.
(4) The newcomer uses the standard Pup procedures for requesting a boot file [1].
(5) The file transfer protocol is used to transfer the worm program to the newcomer.

In general, the program sent to a new segment is just a copy of the program currently running in the worm; this makes it easy to transfer any dynamic state information into new segments. But the new segment first executes a piece of initialization code, allowing it to reestablish any important machine-dependent state (for example, the number of the host on which it is running).

2.5 Intra-Worm Communication—The Need for Multidestination Addressing

All segments of the worm must stay in communication, in order to know when one of their members has departed. In our experiments, each segment had a full model of its parent worm—a list of all other segments. This is a classic situation in which one host wants to send some information to a specified collection of hosts—what is known as *multidestination addressing* or *multicasting* (also called *group addressing*) [2, 5]. Unfortunately, the experimental Ethernet design does not directly support any explicit form of multicasting. There are, however, several alternatives available [6]:

(1) *Pseudo-multicast ID*: An unused *physical* host number can be set aside as a special *logical* group address, and all participants in the group set their host ID to this value. This is a workable approach (used in some existing programs), but does require advance coordination. In addition, it consumes one host ID for each worm.

(2) *Brute force multicast*: A copy of the information is sent to each of the group's other members. This is one of the techniques which was used with the worms: each segment periodically sends its status to all other segments.

The latter approch does require sending $n*(n-1)$ packets for each update; other techniques reduce the total number of packets which must be sent. Many of the worms, however, were actually quite small, requiring only three or four machines to ensure that they would not die when one machine was lost. In these cases, the explicit multicast was very satisfactory. When an application needs a substantial number of machines, they can be obtained with one large worm or with a set of cooperating smaller worms.

This state information being exchanged is used by each independent segment to run an algorithm similar to the one for updating routing tables in store-and-forward packet-switched networks and internetworks: if a host is not heard from after some period of time, it is presumed dead and eliminated from the table. The remaining segments then cooperate to give one machine responsibility for finding a new segment, and the process continues.

2.6 Releasing a Machine

When a segment of a worm is finished with a machine, it needs to return that machine to an idle state. This is very straightforward: the segment invokes the standard network boot procedure to reload the memory diagnostic program, that test is resumed, and the machine is again available as an idle machine for later reuse.

This approach does result in some unfortunate behavior should a machine crash, either while running the segment or while trying to reboot. With no program running, the machine cannot access the network and, as we saw, there is no way to reach in from the net to restart it. The result is a stopped machine, inaccessible to the worm. The machine is still available, of course, to the first user who walks up it it.

3. A Key Problem: Controlling a Worm

> No, Mr. Sullivan, we can't stop it! There's never been a worm with that tough a head or that long a tail! It's building itself, don't you understand? Already it's passed a billion bits and it's still growing. It's the exact inverse of a phage—whatever it takes in, it adds to itself instead of wiping . . . Yes, sir! I'm quite aware that a worm of that type is theoretically impossible! But the fact stands, he's done it, and now it's so goddamn comprehensive that it can't be killed. Not short of demolishing the net!
>
> —John Brunner, *The Shockwave Rider*

We have only briefly mentioned the biggest problem associated with worm management: controlling its growth while maintaining stable behavior.

Early in our experiments, we encountered a rather puzzling situation. A small worm was left running one night, just exercising the worm control mechanism and using a small number of machines. When we returned the next morning, we found dozens of machines dead, apparently crashed. If one restarted the regular memory diagnostic, it would run very briefly, then be seized by the worm. The worm would quickly load its program into this new segment; the program would start to run and promptly crash, leaving the worm incomplete—and still hungrily looking for new segments.

We have speculated that a copy of the program became corrupted at some point in its migration, so that the initialization code would not run

properly; this made it impossible for the worm to enlist a new, healthy segment. In any case, some number of worm segments were hidden away, desperately trying to replicate; every machine they touched, however, would crash. Since the building we worked in was quite large, there was no hint of which machines were still running; to complicate matters, some machines available for running worms were physically located in rooms which happened to be locked that morning so we had no way to abort them. At this point, one begins to imagine a scene straight out of Brunner's novel—workers running around the building, fruitlessly trying to chase the worm and stop it before it moves somewhere else.

Fortunately, the situation was not really that grim. Based upon an ill-formed but very real concern about such an occurrence, we had included an emergency escape within the worm mechanism. Using an independent control program, we were able to inject a very special packet into the network, whose sole job was to tell every running worm to stop no matter what else it was doing. All worm behavior ceased. Unfortunately, the embarassing results were left for all to see: 100 dead machines scattered around the building.

This anecdote highlights the need for particular attention to the control algorithm used to maintain the worm. In general, this distributed algorithm involves processing incoming segment status reports and taking actions based upon them. On one hand, you may have a "high strung worm": at the least disturbance or with one lost packet, it may declare a segment gone and seek a new one. If the old segment is still there, it must later be expunged. Alternatively, some control procedures were too slow in responding to changes and were constantly operating at less than full strength. Some worms just withered and died, unable to promptly act to rebuild their resources.

Even worse, however, were the unstable worms, which suddenly seemed to grow out of control, like the one described above. This mechanism is not yet fully understood, but we have identified some circumstances that can make a worm grow improperly. One factor is a classic failure mode in computer communications systems: the *half-up link* (or one-way path) where host *A* can communicate with host *B*, but not the other way around. When information about the state of the worm is being exchanged, this may result in two segments having inconsistent information. One host may think everything is fine, while another insists that a new segment is necessary and goes off to find it.

Should a network be partitioned for some time, a worm may also start to grow. Consider a two-segment worm, with the two segments running on hosts at opposite ends of an Ethernet cable, which has a repeater in the middle. If someone temporarily disconnects the repeater, each segment will assume that the other has died and seek a new partner. Thus, one two-part worm becomes two two-part worms. When the repeater is turned back on, the whole system suddenly has too many hosts committed to worm programs. Similarly, a worm which spans different networks may become partitioned if the intermedite gateway goes down for a while and then comes back up.

In general, the stability of the worm control algorithms was improved by exchanging more information, and by using further checks and error detection as the programs evaluated the information they were receiving. For example, if a segment found that it continually had trouble receiving status reports from other segments, it would conclude that it was the cause of the trouble and thereupon self-destruct.

Furthermore, a special program was developed to serve as a "worm watcher" monitoring the local network. If a worm suddenly started growing beyond certain limits, the worm watcher could automatically take steps to restrict the size of the worm or shut it down altogether. In addition, the worm watcher maintained a running log recording changes in the state of individual segments. This information was invaluable in later analyzing what might have gone wrong with a worm, when, and why.

It should be evident from these comments that the development of distributed worm control algorithms with low delay and stable behavior is a challenging area. Our efforts to understand the control procedures paid off, however: after the initial test period the worms ran flawlessly, until they were deliberately stopped. Some ran for weeks, and one was allowed to run for over a month.

4. Applications Using the Worms

In the previous sections we have described the procedures for starting and maintaining worms; here we look at some real worm programs and applications which have been built.

4.1 The Existential Worm

The simplest worm is one which runs a null program—its sole purpose in life is to stay alive, even in the face of lost machines. There is no substantive application program being run (as a slight embellishment, though, a worm segment can display a message on the machine where it is running).

This simple worm was the first one we constructed, and it was used extensively as the test vehicle for the underlying control mechanisms. After the first segment was started, it would reach out, find additional free machines, copy itself into them, and then just rest. Users were always free to reclaim their machines by booting them; when that happened, the customary worm procedure would find and incorporate a new segment.

As a rule, though, this procedure would only force the worm to change machines at very infrequent intervals. Thus, the program was equipped with an independent self-destruct timer: after a segment ran

on a machine for some random interval, it would just allow itself to expire, returning the machine to an idle state. This dramatically increased the segment death rate, and exercised the worm recovery and replication procedures.

4.2 The Billboard Worm

With the fundamental worm mechanism well in hand, we tried to enhance its impact. As we described, the Existential worm could display a small message; the "Billboard worm" advanced this idea one step further, distributing a full-size graphics image to many different machines. Several available graphics programs used a standard representation for an image—pictures either produced from a program or read in with a scanner. These images could then be stored on a network file server and read back through the network for display on a user's machine.

Thus, the initial worm program was modified so that when first started, it could be asked to obtain an image from one of the file servers. From then on, the worm would spread this image, displaying it on screens throughout the building. Two versions of the worm used different methods to obtain the image in each new segment: the full image could be included in the program as it moved, or the new segment could be instructed to read an image directly from one of the network servers.

With a mechanical scanner to capture an image, the Billboard worm was used to distribute a "cartoon of the day"—a greeting for workers as they arrived at their Altos.

4.3 The Alarm Clock Worm

The two examples just described required no application-specific communication among the segments of a worm; with more confidence in the system, we wanted to test this capability, particularly with an application that required high reliability. As a motivating example we chose the development of a computer-based alarm clock which was not tied to a particular machine. This program would accept simple requests through the network and signal a user at some subsequent time; it was important that the service not make a mistake if a single machine should fail.

The alarm clock was built on top of a multimachine worm. A separate user program was written to make contact with a segment of the worm and set the time for a subsequent wake-up. The signalling mechanism from the worm-based alarm clock was convoluted, but effective: the worm could reach out through the network to a server normally used for out-going terminal connections and then place a call to the user's telephone!

This is an interesting application because it needs to maintain in each segment of the worm a copy of the database—the list of wake-up calls to be placed. The strategy was quite simple: each segment was given the current list when it first came up. When a new request arrived, one machine took responsibility for accepting the request and then propagating it to the other segments. When placing the call, one machine notified the others that it was about to make the call, and once completed, notified the others that they could delete the entry. This was, however, primarily a demonstration of a multimachine application, and not an attempt to fully explore the double-commit protocols or other algorithms that maintain the consistency of duplicate databases.

Also note that this was the first application in which it was important for a separate user program to be able to find the worm, in order to schedule a wake-up. In the absence of an effective group-addressing technique, we used two methods: the user program could solicit a response by broadcasting to a well-known socket on all possible machines, or it could monitor all traffic looking for an appropriate status report from a worm segment.

4.4 Multimachine Animation Using a Worm

So far, the examples described have used a distributed worm, with no central control. One alternative way to use a worm, however, is as a robust set of machines supporting a particular application—an application that may itself be tied to a designated machine. An example which we have explored is the develoment of a multimachine system for real-time animation. In this case, there is a single *control node* or *master* which is controlling the computation and playing back the animation; the multiple machines in the worm are used in parallel to produce successive frames in the sequence, returning them to the control node for display.

The master node initially uses the worm mechanisms to acquire a set of machines. In one approach, the master first determines how many machines are desired and then recruits them with one large worm. As we just discussed, however, a single large worm may be slow to get started as it sequentially looks for idle machines, and it may be unwieldy to maintain. Instead of using one large worm to support the animation, the master spawns one worm with instruction on how many other worms to gather. This starting worm launches some number of secondary worms, which in turn acquire their full complement of segments (in this experiment, three segments per worm). Thus, one can very rapidly collect a set of machines responding to the master; this collection of machines is still maintained by the individual worm procedures.

Each worm segment then becomes a "graphics machine" with a pointer back to the master, and each reports in with an "I'm alive" message after it is created; the master itself is not part of any worm. The master maintains the basic model of the three-dimensional image and controls the steps in the animation. To actually produce each frame, though, it only has to send the coordinates for each object; the "worker" machine then performs the hidden-line elimination and half-tone shading, computing the finished frame. With this approach, all of the worm segments work in parallel, performing the computationally intensive

tasks. The master supplies descriptions of the image to the segments and later calls upon them to return their result for display as the next image.

The underlying worm mechanism is used to maintain the collection of graphics workers; if a machine disappears, the worm will find a new one and update the list held by the control program. The worm machines run a fairly simple program, with no specific knowledge about the animation itself. The system was tested with several examples, including a walk through a cave and a collection of bouncing and rotating cubes.

4.5 A Diagnostic Worm for the Ethernet

The combination of a central control machine and a multipart worm is also a useful way to run distributed diagnostics on many machines. We knew, for example, that Alto Ethernet interfaces showed some pair-wise variation in the error rates experienced when communicating with certain other machines. To fully test this, however, would require running a test program in all available machines—a terribly awkward task to start manually.

The worm was the obvious tool. A control program was used to spawn a three-segment worm, which would then find all available machines and load them with a test program; these machines would then check in with the central controller and prepare to run the specified measurements. Tests were conducted with as many as 80, 90, or even 120 machines.

In testing pair-wise error rates, each machine had a list of all other participants already loaded by the worm and registered with the control program. Each host would simply try to exchange packets with each other machine thought to be a part of the test. At the end of the test each machine would report its results to the control host—thus indicating which pairs seemed to have error-prone (or broken) interfaces.

Figure 3 is the Ethernet source-destination traffic matrix produced during this kind of worm-based test. To speed the process of gathering all available machines, a three-segment worm would be spawned, and these segments could then work in parallel. Host 217 was the control Alto, and

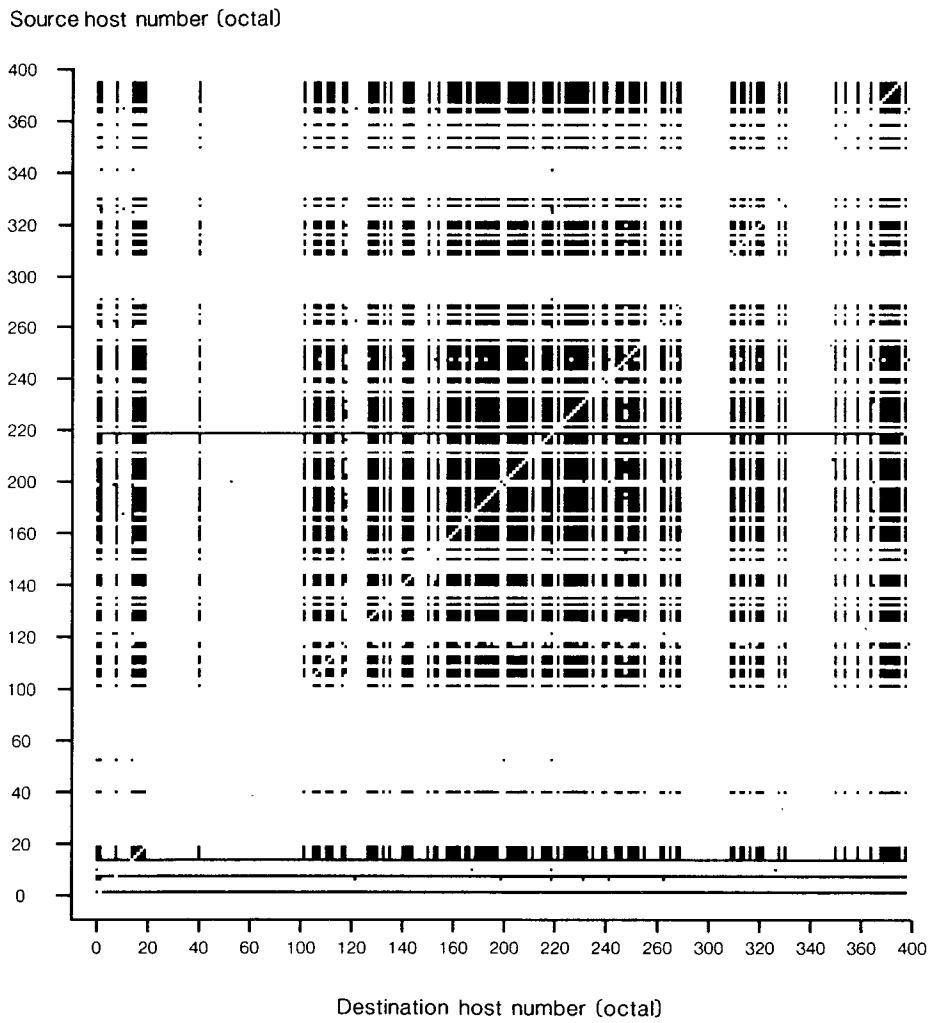

Fig. 3. Ethernet Source-Destination Traffic Matrix When Testing Ethernet Connectivity. (Total number of source-destination pairs = 11,396.)

it found the three segments for its worm on hosts between 0 and 20. Those three segments then located and initialized all of the other participants. As described earlier, a simple linear search through the host address space is used by each segment to identify idle machines. To keep the multiple segments from initially pinging the same hosts, the starting point for each segment could be selected at intervals in the address space. Each segment does make a complete cycle through the address space, however, looking carefully for any idle machines.

To avoid any unusual effects during the course of the test itself, the worm maintenance mechanism was turned off during this period. If hosts had died, the worm could later be reenabled, in an effort to rebuild the collection of hosts for a subsequent test.

At the conclusion of the tests, all of the machines are released and allowed to return to their previous idle state—generally running the memory diagnostic. These machines would boot that diagnostic through the network, from one of the network boot file servers; 120 machines trying to do this at once, however, can cause severe problems. In particular, the boot server becomes a scarce resource that may not be able to handle all of the requests right away, and the error recovery in this very simple network-boot procedure is not foolproof. Thus, all of the participants in the measurements coordinate their departure at the end of a test: each host waits for a quasi-random period before actually attempting to reboot from the network boot server.

5. Some History: Multimachine Programs on the Arpanet

The worm programs, of course, were not the first multimachine experiments. Indeed, some of the worm facilities were suggested by the mechanisms used within the Arpanet or demonstrations built on top of that network:

(1) The Arpanet routing algorithm itself is a large, multimachine distributed computation, as the Interface Message Processors (IMPs) continually exchange information among themselves. The computations continue to run, adapting to the loss or arrival of new IMPs. (Indeed, this is probably one of the longest-running distributed computations.)

(2) In a separate procedure, the Arpanet IMPs can be individually reloaded through the network, from a neighboring IMP. Thus, the IMP program migrates through the Arpanet, as needed.

(3) In late 1970, one of the earliest multimachine applications using the Arpanet took place, sharing resources at both Harvard and MIT to support an aircraft carrier landing simulation. A PDP-10 at Harvard was used to produce the basic simulation program and 3-D graphics data. This material was then shipped to an MIT PDP-10, where the programs could be run using the Evans & Sutherland display processor available at MIT. Final 2-D images produced there were shipped to a PDP-1 at Harvard, for display on a graphics terminal. (All of this was done in the days before the regular Network Control Program (NCP) was running; one participant has remarked that "it was several years before the NCPs were surmounted and we were again able to conduct a similar network graphics experiment.")

(4) "McRoss" was a later multimachine simulation built on top of the NCP, spanning machine boundaries. This program simulated air traffic control, with each host running one part of the simulated air space. As planes moved in the simulation, they were handed from one host to another.

(5) One of the first programs to move by itself through the Arpanet was the "Creeper," built by B. Thomas of Bolt Beranek and Newman (BBN). It was a demonstration program under Tenex that would start to print a file, but then stop, find another Tenex, open a connection, pick itself up and transfer to the other machine (along with its external state, files, etc.), and then start running on the new machine. Thus, this was a relocatable program, using one machine at a time.

(6) The Creeper program led to further work, including a version by R. Tomlinson that not only moved through the net, but also replicated itself at times. To complement this enhanced Creeper, the "Reaper" program moved through the net, trying to find copies of Creeper and log them out.

(7) The idea of moving processes from Creeper was added to the McRoss simulation to make "relocatable McRoss." Not only were planes transferred among air spaces, but entire air space simulators could be moved from one machine to another. Once on the new machine, the simulator had to reestablish communication with the other parts of the simulation. During the move this part of the simulator would be suspended, but there was no loss of simulator functionality.

This summary is probably not complete or fully accurate, but it is an impressive collection of distributed computations, produced within or on top of the Arpanet. Much of this work, however, was done in the early 70s; one participant recently commented, "It's hard for me to believe that this all happened seven years ago." Since that time, we have not witnessed the anticipated blossoming of many distributed applications using the long-haul capabilities of the Arpanet.

6. Conclusions

We have the tools at hand to experiment with distributed computations in their fullest form: dynamically allocating resources and moving from machine to machine. Furthermore, local networks supporting relatively large numbers of hosts now provide a rich environment for this kind of experimentation. The basic worm programs described here demonstrate the ease with which these mechanisms can be explored; they also highlight many areas for further research.

Acknowledgments

This work grew out of some early efforts to control multimachine measurements of Ethernet performance [6, 7, 8]. E. Taft and D. Boggs produced much of the underlying software that made all of these efforts possible. In addition, J. Maleson implemented most of the graphics software needed for the multimachine animation; his imagination helped greatly to focus our effort on a very real, useful, and impressive application. When we first experimented with multimachine migratory programs, it was S. Weyer who pointed out the relevance of John Brunner's novel describing the "tapeworm" programs. (Readers interested in both science fiction and multimachine programs might also wish to read *The Medusa Conspiracy* by Ethan I. Shedley and *The Adolescence of P-1* by Thomas J. Ryan.) Finally, our thanks to the many friends within the Arpanet community who helped piece together our brief review of Arpanet-related experiments, and our apologies to anyone whose work we overlooked.

References

1. Boggs, D.R., Shoch, J.F., Taft, E.A., and Metcalfe, R.M. PUP: An internetwork architecture. *IEEE Trans. Commun. 28*, 4 (April 1980). Describes the Pup internetwork architecture, used to tie together over 1,200 machines on several dozen different networks.

2. Dalal, Y.K. Broadcast protocols in packet switched computer networks. Tech. Rep. 128, Stanford Digital Syst. Lab., Stanford, Calif., April 1977. Discussion of alternative techniques for broadcast addressing.

3. Dalal, Y.K., and Printis, R.S. 48-bit Internet and Ethernet host numbers (to be published in the Proc. 7th Data Comm. Symp., Oct. 1981). Describes the use of broadcast and multicast addresses in an internet design, and how this influenced the development of the Ethernet addressing scheme.

4. Metcalfe, R.M., and Boggs, D.R. Ethernet: Distributed packet switching for local computer networks. *Comm. ACM 19*, 7 (July 1976), 395–404. The original Ethernet paper, describing the principles of operation and experience with the Experimental Ethernet.

5. Shoch, J.F. Internetwork naming, addressing, and routing. Proc. 17th IEEE Comp. Soc. Int. Conf. (Compcon Fall '78), Washington, D.C., Sept. 1978.General discussion of addressing modes, including the use of multicast addressing.

6. Shoch, J.F. *Local Computer Networks.* McGraw-Hill, New York (in press). A survey of alternative local networks and a detailed description of the Ethernet local network.

7. Shoch, J.F., and Hupp, J.A. Performance of an Ethernet local network–a preliminary report. Local Area Comm. Network Symp., Boston, Mass., May 1979 (reprinted in the Proc. 20th IEEE Comp. Soc. Int. Conf. (Compcon Spring '80), San Francisco, Calif., Feb. 1980). Description of the measured performance of the Ethernet.

8. Shoch, J.F., and Hupp, J.A. Measured performance of an Ethernet local network. *Comm. ACM 23*, 12 (Dec. 1980), 711–721. Detailed discussion of the measured performance of the Ethernet, including several source-destination traffic graphs similar to the ones presented here.

9. Shoch, J.F., Dalal, Y.K., Crane, R.C., and Redell, D.D. Evolution of the Ethernet local computer network. Xerox Tech. Rep. OPD–T81–02, Palo Alto, Calif., Sept. 1981. The basic paper on the revised and improved Ethernet Specification, including comparisons with the original Experimental Ethernet.

10. Thacker, C.P., McCreight, E.M., Lampson, B.W., Sproull, R.F., and Boggs, D.R. Alto: A personal computer. In *Computer Structures: Principles and Examples, 2nd edition*, Siewiorek, Bell, and Newell, Eds., McGraw–Hill, New York, 1982, 549–572. Describing the Alto computer—a high-performance, single-user machine—which was used for running the worm programs.

The attention of Computing Practices readers is called to a letter on spelling checkers by Raben in the ACM Forum, pp. 220–221.