IEEE Trans. Nucl. Sci. **NS–43** (1996) 2457.

A Simple Multiprocessor Management System for Event-Parallel Computing

Steve Bracker, Krishnaswamy Gounder,
Kevin Hendrix, and Don Summers

Department of Physics and Astronomy
University of Mississippi–Oxford
University, MS 38677 USA

Abstract — **Offline software using TCP/IP sockets to distribute particle physics events to multiple UNIX/RISC workstations is described. A modular, building block approach was taken, which allowed tailoring to solve specific tasks efficiently and simply as they arose. The modest, initial cost was having to learn about sockets for interprocess communication. This multiprocessor management software has been used to control the reconstruction of eight billion raw data events from Fermilab Experiment E791.**

I. THE E791 RECONSTRUCTION TASK

Fermilab Experiment E791 accumulated a large dataset (50 Terabytes, 20 billion events, 24 000 8mm Exabyte tapes) in 1991 and early 1992 [1,2]. As might be expected, the reconstruction and analysis of this data challenged available computing resources; event reconstruction alone required over 10 000 mips-years of processing power. The computing task was larger than that faced at high luminosity collider experiments, which are compared to E791 in Table 1. For $2\frac{1}{2}$ years, reconstruction processing was underway at four different locations [3]. The three largest sites used clusters or *farms* of commercial UNIX/RISC workstations connected together by Ethernet. Within each farm, many processors operated together; data management and system control were exerted from a single point via multiprocessor management software. Here we describe the multiprocessor management software developed and used at the University of Mississippi [4]. The Mississippi farm hardware is shown in Figs. 1 and 2.

Most of the large-scale computing needs encountered in particle physics – reconstruction and analysis of recorded data and generation of simulated data – are event-oriented. Each event's data packet is extracted from an input stream and processed in isolation from other events. The results from each event's analysis are merged into an output stream. The computing power required to process each data packet is significant relative to the time needed to transport the data, even over datapaths of modest throughput. Many other scientific computing problems (e.g. small-object recognition in astronomical images) conform to the same model. In all such problems, it is trivial to divide the total computing task over many processors; each processor is simply given a share of the events to process independently of its peers.

Most reconstruction and simulation tasks are developed and tested as single processor programs. They can be outlined in the following manner:

initialize data structures
initialize input and output data streams
read run startup information from input data stream
read and store run calibration constants

for every event from the input stream...
   unpack and check the input data
   PROCESS THE EVENT (most of the work)
   for good events,
      pack the processed event into output stream

produce a report characterizing the processing job
stop

When the program is "nearly working," it is then moved into the multiprocessing environment for final testing and production. If it has been written in a sensible fashion, with dataflow management tasks cleanly separated from event processing tasks, then the transition from single processor to multiprocessor system should be relatively painless. Dataflow management is generally vested in a single ***server process***, while the event processing is performed by numerous ***client processes*** running in many processors. The server grants the clients access to calibration data, reads the event input stream, parcels out events to the many client processors, gathers output data from the clients, and writes the output stream. The server processor is typically well endowed with peripheral devices (disks, tape drives), whereas the clients may be very simple (but powerful) processors with no peripherals except a network connection.

II. THE MULTIPROCESSOR MANAGER'S TASKS

In the first implementations of event-oriented multiprocessing for particle physics [5,6], the multiprocessor management software had to do a great deal of hard work. Clients frequently had no significant operating system, primarily because memory was expensive. Executable code and calibration data had to be formatted in the server and downloaded into the clients word by word. Extraction of input events from records and merging of output events into records often had to be done in the server because of limited client buffer memory, posing the danger of a processing bottleneck at the server. Reports had to be retrieved from the clients as tables of numbers, to be formatted and output by the server. Substantial effort was often required to move a single processor program onto a multiprocessor system, because the division between server code and client code was intricate. Developing the multiprocessor management code could become nearly as large a task as developing the event processing software: an odious burden.

With the advent of cheaper memory (enabling each client to have a real operating system and large event buffers) plus good networking software, most of the difficulties inherent in moving code to a multiprocessor vanished. Rather than relating "war stories" of heroic efforts and brilliant strategies leading to victory in the face of staggering difficulties, this paper celebrates the fact that multiprocessor management is now straightforward, that an effort tiny compared to the development of application code will suffice to distribute the workload efficiently and reliably among many processors, and that generic system software nearing completion promises to make the task even easier in the future.

III. A DISK-BASED MULTIPROCESSOR MANAGER

It is easiest to understand the multiprocessor strategy by considering first a disk-based system. The clients each have a real operating system (in our case ULTRIX, a flavor of UNIX) and thus can read and write disk files, but they have no disks attached directly to them. Instead, clients have access over the network to disks attached to the server. Using Network File System (NFS) software, disks or portions of disks can be "cross-mounted" so that they are accessible by multiple processors. For us, this meant the server itself and all of its clients. NFS is supplied as standard software with many types of workstations, and is available as an option on most of the rest.

The fact that clients, though diskless, have full access to disk services across the network, immediately solves several problems that burdened earlier systems:

(1) Clients can read executable code and start programs; it is no longer necessary for the server to micromanage the downloading and startup of clients, though high level control of client processes is still maintained in the server,

(2) Clients can read their own calibration files; it is no longer necessary for the server to explicitly

read, reformat, and transmit calibration data for the clients,

(3) Clients can write their own report files; it is no longer necessary for the server to explicitly extract, format, and write reports for the clients.

In many cases, cross-mounted disks can even be used to provide for the movement of event data (input and output) between server and clients:

(4) The server can read events from tape and write them to disk; the clients can read the events from disk and process them. For example, each client can own two input files; the server inserts events into one while the client consumes events from the other,

(5) The clients can write output events to disk; the server can read the disk files, and write the events to output tapes. Each client can own two output files; the client inserts processed events into one while the server removes events from the other and writes them to tape.

Referring back to the outline of the processing tasks, it is clear that the program running in each client of a multiprocessor system is very similar to the single processor program. Each client loads its own code and starts it. Each client reads its own calibration constants from disk. Each client writes its own reports to disk. Each client reads its input data from disk and writes its output data to disk, just as single processor code usually does during program development. The server's tasks now become the following:

(1) Keeping a list of the available processors and requesting them to start instances of the processing program,

(2) Making startup data (e.g. run number) available to clients in a disk file,

(3) Assuring that valid calibration constants are available to be read by the clients,

(4) Reading input data from tape and distributing events to the various client disk files,

(5) Fetching output data from the various client disk files and writing it to tape,

(6) Gathering reports from the client report files and producing a system-wide report.

This scheme was very simple to implement, maintain, and document. For applications in which dataflow is slow, it works very well. Clients are provided with certified events (or blocks of events) on a reliable medium, protected from the vagaries of tape reading, which is handled by the server. Likewise, clients have a reliable medium for writing output; problems with tape writing are handled in the server. If enough disk is available, whole tape files can be staged through the disk files, leading to more reliable performance from streaming tape drives [7].

IV. A TCP/IP SOCKETS BASED MULTI–PROCESSOR MANAGER

As a farm's data throughput is increased by adding processors, upgrading processors, or making application code run faster, passing the event stream through disk can become a bottleneck; clients may spend too much time waiting for input events to be read and output events to be written, and disk "thrashing" (incessant head motion) may set in. One possible solution is to move the event streams through cross-mounted *memory disks* using special software by which a portion of the server's memory is set aside and appears to the user as a very high-speed, low-latency, non-thrashing, disk drive. When we explored this option in 1992, such software was available but failed to work properly. Memory disks preserve the advantage that the application code for a multiprocessor system looks almost identical to that for a single processor; all data passing is done with normal Fortran reads and writes.

Instead, our multiprocessor manager bypasses disk altogether for those portions of the dataflow that are fast enough to challenge the disk's throughput; data is moved between processes using TCP/IP (Transmission Control Protocol/Internet Protocol) network services [8]. Using this facility, processes can establish a connection between themselves and pass data back and forth by the subroutine calls, *read_from_connection* and *write_through_connection.* Event input/output can no longer be implemented as simple Fortran reads and writes in the client (a disadvantage), but on the other hand, the high throughput of direct network data transfers becomes available.

Before deciding to implement a TCP/IP-based scheme, we had to answer three significant questions:

(1) Would passing data through TCP/IP yield a significant improvement in throughput over NFS disk-based data transfers?

(2) Would resulting modifications to the client code be tractable?

(3) Given very limited time to produce the management code, would TCP/IP be easy enough to learn and use, especially for those of us coming from a VAX/VMS—Fortran environment?

We wrote a test package to shuttle data between several processors to measure throughput and reliability. The results: average data throughputs in excess of 900 kilobytes per second (almost full Ethernet speed) could be maintained (we needed only 150 kilobytes per second); the impact on client efficiency was immeasurably low at maximum projected throughput; and data corruption was not seen.

In keeping with the Fortran orientation of the experiment's software, Fortran-callable functions were written in C for all of the system services needed to support TCP/IP data transfers. As a result, the changes to the client code were minimal and easily understandable to Fortran-only programmers.

Fig. 3 illustrates how the network I/O calls are used. As the server prepares to start a client, it uses *make_socket* to "have a phone put in", so that it will be able to connect to the client. When the client starts, it too uses *make_socket* to "have a phone put in". The server "lists its number" by binding its socket to a port (*bind_socket*), and "stays near the phone" listening for an attempt to connect (*listen_socket*). The client "calls up" the server (*connect_socket*) and the server "picks up the phone" establishing the connection (*accept_socket*).

When it needs input data, the client "places its order" by writing a message to the server (*write_socket*). The server is continually monitoring all of the client connections for requests (*select_socket*). When a request comes in, the server "writes down the order" (*read_socket*), and does its best to satisfy the client's request. The client and server shuttle messages back and forth (each writing to the other and reading from the other) until the input data is exhausted. At that point, the server notifies the clients that there is no more data. The clients then finish their tasks, close their connections (*close_socket*), and exit; the server finishes its tasks and exits.

The Fortran-callable routines that manipulate sockets and connections really are that simple to use. Only one routine (*read_socket*) is any more than a C to Fortran interface, and even *read_socket* is trivial. Almost all of the real work involved with network communications has already been done in UNIX, TCP/IP, and Berkeley Sockets.

Before writing our own multiprocessor software we had considered trying to extract the few features that we needed from large packages under development at the Fermi [9–11] and Argonne [12] National Laboratories. However, offsite support was unavailable. And our small group did not have the resources to modify the packages to meet our needs (e.g. running many clients per server effectively, ability to tolerate client crashes, ability to tolerate operating system upgrades, etc.). Instead we concentrated on writing software which could do a limited number of things very well. Six person–weeks over an 8 week period were required to write the farm control software. Farm operation typically required 5 hours of work spread out over each 24 hour day. After seeing our approach, the D0 experiment at Fermilab decided to follow a similar strategy to our own for moving farm data [13,14]. The D0 experiment has processed the second largest data set in high energy physics.

V. PERFORMANCE

Only the transport of input events from server to clients was implemented in TCP/IP; output events from clients to server needed to be staged to disk to make best use of our Exabyte tape drives, so they were written to disk through NFS as before. This output scheme was efficient because five out of six events were rejected by a filter after reconstruction and were not output.

Tape reading was multiply buffered, so that events were almost always available immediately when a client asked for them. The primary cause of inefficiency in multiprocessors of this type is compelling client processors to wait in line for new data to process, wasting some fraction of their cycles.

In our system, whenever the server obtained new data, clients were allowed to bid for it, according to their place in a fixed priority list. If the first processor in the list needed data, it received the new packet; otherwise the second processor in the list had a chance to grab the data packet, etc. By comparing the event processing rate of the most favored processor with that of the least favored processor, we could determine whether the system as a whole was being starved for data. In fact, there was no statistically significant difference between the data processing rates of most favored processors and least favored processors, lending strong support to the notion that no processor was wasting much time waiting for new data to work on.

By deliberately degrading the ability of the server to obtain new data or the ability of the network to transport it, we could test the sensitivity of this measure; as soon as the total data rate fell slightly below total needs, the relative processing rate of the least favored processor plummeted.

We also selected a large sample of typical events, reconstructed that dataset on a single processor using multi–buffered I/O, and measured the processing time. Then we reconstructed the same set of events on multiple processors, and again measured the processing time. Ideally one would expect the time required to reconstruct the whole dataset to vary inversely with the number of processors working on it. There was no statistically significant difference between the actual performance of multiple processors and that expectation.

Altogether, the performance measurements we made permit us to say with assurance that more than 97% of the client processing cycles were put to beneficial use; the data are not inconsistent with 100% efficiency.

Funding awarded in June 1993 allowed an expansion of the Mississippi computing facility from 1100 to 2900 mips. By July 1993, the increased computing power had been acquired and was processing data. E791 reconstruction was completed in September 1994. A total of eight billion events on 10 000 raw data tapes were processed at the University of Mississippi. Preliminary versions of the reconstruction software were run to find out which E791 algorithms would yield the most physics per tape. One result was a tripling in the yield of charm particles. When the final reconstruction software was ready, it was run. Overall efficiency, considering all cycles lost for *any* reason, exceeded 90%. Thus the multiprocessor manager has proven itself to be efficient and robust under varying conditions in what we believe to be a fairly typical university operating environment.

For the E791 experiment as a whole, the successful management of multiple processors has provided the full reconstruction of over 200 000 particles with a charm quark, the largest sample in the world. This large charm sample is in turn generating new physics results [15–22].

VI. SCAVENGING COMPUTING CYCLES

Although most of the processors in the Mississippi system are mounted in racks at a central location, some are at people's desks and serve as general purpose workstations. Many workstation activities – editing, compiling and running small programs, reading and writing e-mail, etc. – can coexist with the farm client process running in the background. However, there are some workstation activities that are incompatible with farm operations, and the client process must be removed from the workstation. Nevertheless, it is certainly helpful to be able to scavenge the desk workstation cycles when they are otherwise unused.

There are several approaches to using the desk workstations as farm clients. On one extreme, the workstation can be removed from the farm client list; it will never be used as a farm client. At the other extreme, one can forbid workstation activities that interfere with the farm operation. In between there is ample middle ground. For example, it is possible to have the farm server examine the workstation activities from time to time and adjust the priority or run status of the farm client process accordingly.

At Mississippi, we have found it satisfactory to allow users to abruptly kill the client process whenever they find its activities on their workstations to be troublesome. The server is quickly aware that the client has disappeared and adjusts its event distribution accordingly. If a workstation user knows that he will be engaged in a computationally intensive activity for a long time, he may edit the client list and remove his workstation from the farm. Once killed, a workstation cannot participate as a farm client again until the next job is started. Though crude, this scheme has proven highly effective in our operating environment.

One disadvantage of this approach is that a few input events may be "trapped" in a killed processor, and thus lost. We take a rather cavalier attitude toward such losses; with twenty billion events, we can afford to lose a few now and again. Although in principle the server can hold event buffers until successful processing is assured, and reassign the events to another processor if they become trapped in a disabled client, we don't do that. In E791, processing raw events is rather like hauling corn to market in a truck. If a few grains of corn fall out of the truck, no one becomes concerned until the loss becomes large enough to be economically important. The alternative approach – treating events as babies in a hospital nursery, where one normally expects a somewhat stricter accounting – only makes sense when the events have become greatly enriched in scientific significance, late in the reconstruction and analysis cycle [23].

VII. MULTIPROCESSOR OPERATIONS

The most vexing operational problems are those that one might expect in handling a dataset this large – ensuring that all tapes are processed exactly once, preparing and maintaining run-dependent calibration files, making sure that all of the output tapes are correctly labeled, preparing and examining the necessary report files, etc. During the first year of operation, job flow was controlled by scripts composed with the help of small interactive programs. In 1993, a farm job manager with an X Window graphical user interface was written. It took 20 days to program and was specifically targeted at preventing errors we had observed to occur in the setup of jobs and the management of tapes. The input tape control display is shown in Fig. 4. Most of the day-to-day system operation in later stages was performed by students, many of whom had little understanding of the internal workings of the system.

Farm management software consists of three independent programs:

(1) The server code, which provides system and dataflow control,

(2) The client code, linked to the application code routines,

(3) The tape-writing code, running on the server, which fetches events written to disk by clients and transfers them to tape.

The server, the tape writer, and each client produces report files for each job. Within each client, a table of about 200 counters is updated, as each event is processed, Typical quantities counted included the number of events for each trigger type, the number of events for which 3 or more tracks could be reconstructed in the tracking chambers, and the number of events in which both a primary and secondary vertex could be located. Each client writes these counters onto its own file on the server disk at the end of a job. The tape writer (the last program to handle the data) gathers reports together, totals counters, and produces two farm-wide reports. One is a standard statistics file common to all E791 computing sites; the other is a report tailored to the needs of the Mississippi site.

VIII. FUTURE DIRECTIONS

Event-parallel computing farms have come to dominate large-scale computations in experimental particle physics. Because computing paradigms seldom remain viable for more than a decade or so in this field, it is perhaps useful to ask "Whither computer farming?" in the next few years.

Two trends in the computer industry make it unlikely that "farming as usual" will continue much longer. First, individual workstations, PCs, and Macintoshes now coming to market offer prodigious computing power. A 200 MHz Pentium Pro PC, which Mississippi recently purchased, is 15 times faster than one of our 25 MHz MIPS R3000 based DECstations. This PC by itself is almost capable of saturating the 0.25 MB/s output rate of one Exabyte tape drive for E791 reconstruction. In such a case, there seems little point in concentrating dataflow through a server [24], though one might still choose to centralize job control, a comparatively trivial task.

Second, with increased popularity of object-oriented design and languages such as C++ which support it, the data packets now being explicitly passed between processes may be implemented as objects of classes. Before too long, it may be possible to define remote objects, providing a means to exploit the power of many processors within a single unified programming environment while shielding the user from the details of data transport. For programs written within such a system, there will be little difference between a single processor implementation and a multi-processor implementation, an important benefit for those developing application code.

These two trends suggest that one should be cautious about replicating conventional multiprocessor farms in the future. In the early days of "computer farming" in particle physics, there were not suitable commercial processors available, so we built our own [5,6]. After a relatively short time, we were put out of the processor building business by high-powered workstations offered by several vendors [9]. The early implementations of multiprocessor management software were complex, costly, and cumbersome to use; the need for them has been snuffed out by the widespread availability of interprocess communication tools such as TCP/IP sockets and NFS.

The multiprocessor system described here was inexpensive, quick to implement, easy to operate, and very robust by 1992 standards, nearly optimal for many particle physics data reconstruction and event simulation problems. We suspect that the set of computing problems for which such systems are optimal will diminish over the next few years.

ACKNOWLEDGEMENTS

We especially thank Lucien Cremaldi and Breese Quinn for their contributions to building and running the Mississippi farm. This effort was partially supported by the U. S. Department of Energy, DE-FG05-91ER40622.

REFERENCES

[1] D.J. Summers et al., "Charm Physics at Fermilab E791," Proceedings of the XXVII$^{\text{th}}$ Rencontre de Moriond, Electroweak Interactions and Unified Theories, Les Arcs, France, pp. 417–422, 15-22 March 1992; presents an overview of the experiment.

[2] S. Amato, J.R.T. de Mello Neto, J. de Miranda, C. James, D.J. Summers, and S.B. Bracker, "The E791 Parallel Architecture Data Acquisition System," ***Nucl. Instr. and Meth.,*** vol. A324, no. 3, pp. 535–542, January 1993; describes the system used to collect the E791 dataset.

[3] The Ohio State University (moved to Kansas State University in 1993); Fermi National Accelerator Laboratory; Centro Brasileiro de Pesquisas Físicas–Rio de Janeiro; the University of Mississippi.

[4] Although all four processing centers ran the same physics reconstruction code, three different multiprocessor management systems were used.

[5] Paul F. Kunz et al., "Experience Using the 168/E Microprocessor for Off-line Data Analysis," *IEEE Trans. Nucl. Sci.,* vol. NS–27, no. 1, pp. 582–586, February 1980.

[6] J. Biel et al., "Software for the ACP Multiprocessor System," Proceedings of the International Conference on Computing in High Energy Physics, Asilomar (2-6 Feb. 1987), Fermilab-Conf-87/22; *Computer Physics Communications,* vol. 45, pp. 331–337, 1987.

[7] Exabyte tape drives operate most reliably if they read and write data continuously rather than starting and stopping incessantly. In addition, head wear depends on the time the tape is tensioned over the rotating head, not the amount of data written. You obtain far more useful life (Terabytes per head change) from an Exabyte that reads and writes at nearly full speed.

[8] Sidnie Feit, *TCP/IP: Architecture, Protocols, and Implementation*, New York: McGraw-Hill, 1993, ISBN 0-07-020346-6, Chapter 17. There is also an extensive bibliography.

[9] C. Stoughton and D.J. Summers, "Using Multiple RISC CPUs in Parallel to Study Charm Quarks," *Computers in Physics,* vol. 6, No. 4, pp. 371–375, July/August 1992. Fermilab's first UNIX farm.

[10] Frank Rinaldo and Stephen Wolbers, "Loosely Coupled Parallel Processing at Fermilab," *Computers in Physics,* vol. 7, no. 2, pp. 184–190, March/April 1993.

[11] Aleardo Manacero, Jr., "Comparisons of CPS Performance Under Different Network Loads," FERMILAB–PUB–94–033, 8 pages, January 1994.

[12] R.J. Harrison, "Portable Tools and Applications for Parallel Computers," *International Journal of Quantum Chemistry,* vol. 40, pp. 847-863, 1991; describes the Theoretical Chemistry Group Message (TCGMSG) program — Argonne National Laboratory.

[13] David Oyler and Kirill Denisenko, "D0 Unix Farm Production," Computing Techniques Seminar, Fermilab Computing Division Note EN0050, 19 pages, 18 May 1993.

[14] Kirill Denisenko et al., "D0 Farm Production System," ***Proceedings of the Conference on Computing in High Energy Physics '94,*** San Francisco, LBL–35822, pp. 152–154, 21–27 April 1994.

[15] A.K.S. Santha, "Study of the Decay $D^0 \rightarrow K^0_S K^\pm \pi^\mp$," APS/AAPT Joint Meeting, Crystal City, VA (18–22 April 1994); ***Bull. Amer. Phys. Soc.,*** vol. 39, no. 2, pp. 1030–1031, April 1994.

[16] Milind Purohit and James Wiener, "Preliminary Results on the Decays $D^+ \rightarrow K^+\pi^+\pi^-$, $D^+ \rightarrow K^+K^+K^-$," Proc. of the 8th Meeting, Div. of Particles and Fields of the Amer. Phys. Soc., Albuquerque, vol. 1, pp. 969–973, 2–6 August 1994.

[17] Guy Blaylock, "A Search for $D^0\overline{D}^0$ Mixing at the Fermilab E791 Experiment," Proc. of the 8th Meeting, Div. of Particles and Fields of the Amer. Phys. Soc., Albuquerque, vol. 1, pp. 980–983, 2–6 August 1994.

[18] Krishnaswamy Gounder and Lucien Cremaldi, "D-Meson–Pion Production Correlations," APS/AAPT Joint Meeting, Washington (18–21 April 1995); ***Bull. Amer. Phys. Soc.,*** vol. 40, no. 2, p. 1022, April 1995.

[19] E.M. Aitala et al., "Search for D^0–$\overline{D}^0$ Mixing in Semileptonic Decays," KSU-HEP–96-01, Fermilab–Pub–96/109-E, hep–ex/9606016, (accepted by ***Physical Review Letters***).

[20] E.M. Aitala et al., "Mass Splitting and Production of Σ^0_c and Σ^{++}_c Measured in 500 GeV/c π^-–N Interactions," ***Phys. Lett.,*** vol. B379, no. 1–4, pp. 292–298, June 1996.

[21] E.M. Aitala et al., "Asymmetries Between the Production of D^+ and D^- Mesons from 500 GeV/c π^-–Nucleus Interactions as a Function of x_F and p_t^2," ***Phys. Lett.,*** vol. B371, no. 1–2, pp. 157–162, March 1996.

[22] E.M. Aitala et al., "Search for the Flavor-Changing Neutral Current Decays $D^+ \rightarrow \pi^+\mu^+\mu^-$ and $D^+ \rightarrow \pi^+e^+e^-$," ***Phys. Rev. Lett.,*** vol. 76, no. 3, pp. 364–367, January 1996.

[23] Event trapping was, in practice, rare. On average, about 30 events of 10 million events processed each day were trapped (lost to the analysis) through unrecovered client crashes and processors withdrawn from

the farm for other uses. Needless to say, a .0003% loss is of no statistical significance, and no experimenter cared that it happened.

[24] One caveat. I/O is getting faster but for now not as quickly as CPUs. We used Ethernet at 10 Mbit/s; and Exabyte 8200 and 8205 tape drives storing 2.2 Gigabytes at 0.25 Mbytes/s. Full Duplex Fast Ethernet now runs at 2×100 Mbits/s. The Quantum DLT4000 tape drive now stores 20 Gigabytes at 1.5 Mbytes/s. The Exabyte Mammoth Tape drive stores 20 Gigabytes at 3 Mbytes/s. The older Exabyte 8505XL stores 5 or 7 Gigabytes at 0.5 Mbytes/s. Unhappily, the cost of tapes over 5 Gigabytes has increased far more rapidly than capacity.

Experiment	Events Recorded Millions	Raw Data Recorded Terabytes	Recording Period
FNAL E791	20 000	50	7/91 - 1/92
FNAL CDF	95	10	10/85 - 12/95
FNAL D0	80	40	2/92 - 12/95
HERA H1	70	2.5	5/92 - 12/95
HERA ZEUS	50	5	5/92 - 12/95
LEP ALEPH	60	1.7	8/89 - 11/95
LEP DELPHI	~30	~5	8/89 - 11/95
LEP L3	83	3.4	8/89 - 11/95
LEP OPAL	102	1.5	8/89 - 11/95
CESR CLEO	600	5	10/79 - 12/95

Table 1. Comparison of the the size of raw data sets processed by Fermilab E791 with estimates from several $p\overline{p}$, e^-p, and e^+e^- collider experiments. D0 records digitized waveforms for its central detector which increases its data set size. The number of events recorded at LEP includes triggers such as Bhabhas as well as Z^0 triggers.

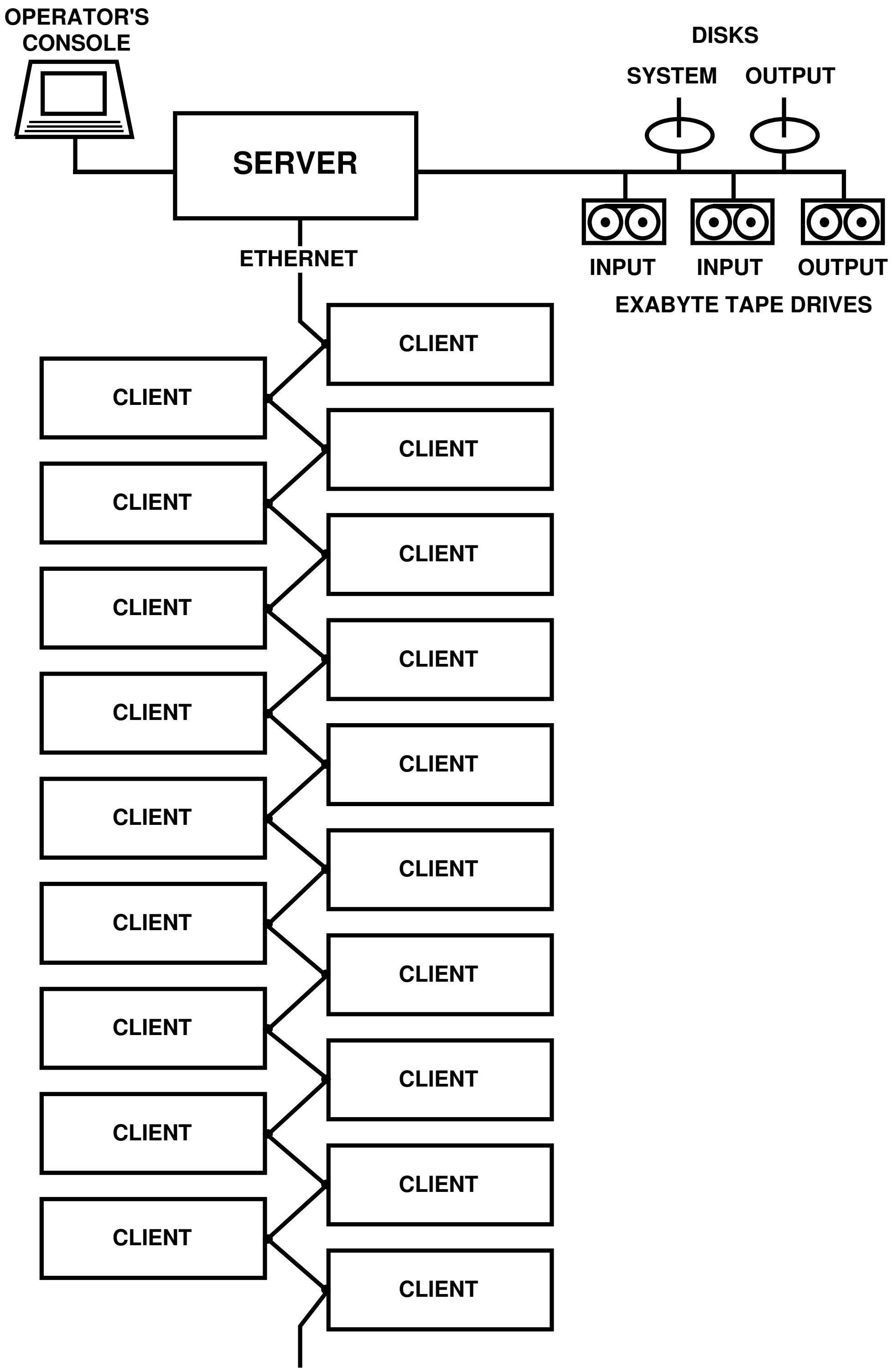

Fig. 1. Computing farm configuration at the University of Mississippi. Servers and clients are DECstation 5000 workstations running ULTRIX. Some have MIPS R3000 Processors; others have the more powerful MIPS R4000. Altogether, there are 68 processors organized into four farms, each with a separate job stream. One typical farm is shown in this diagram. The two input tape drives alternate automatically. Each drive is only wearing itself out half the time. If an input tape drive fails the next tape starts automatically. The output is staged through disk to tape. If the output tape drive fails output data can easily be recovered from disk. If a disk fills, processing is automatically paused until space *appears* on the disk. This I/O scheme avoids the need for continuous operator supervision. The total computing power is about 3000 mips.

Fig. 2. A photographic overview of the University of Mississippi computing farm. Servers are on the four tables. Clients are on the racks shown and on desktops which are not shown. If the room overheats, power is automatically cut to the computers and must be manually reset. The espresso machine is on the left.

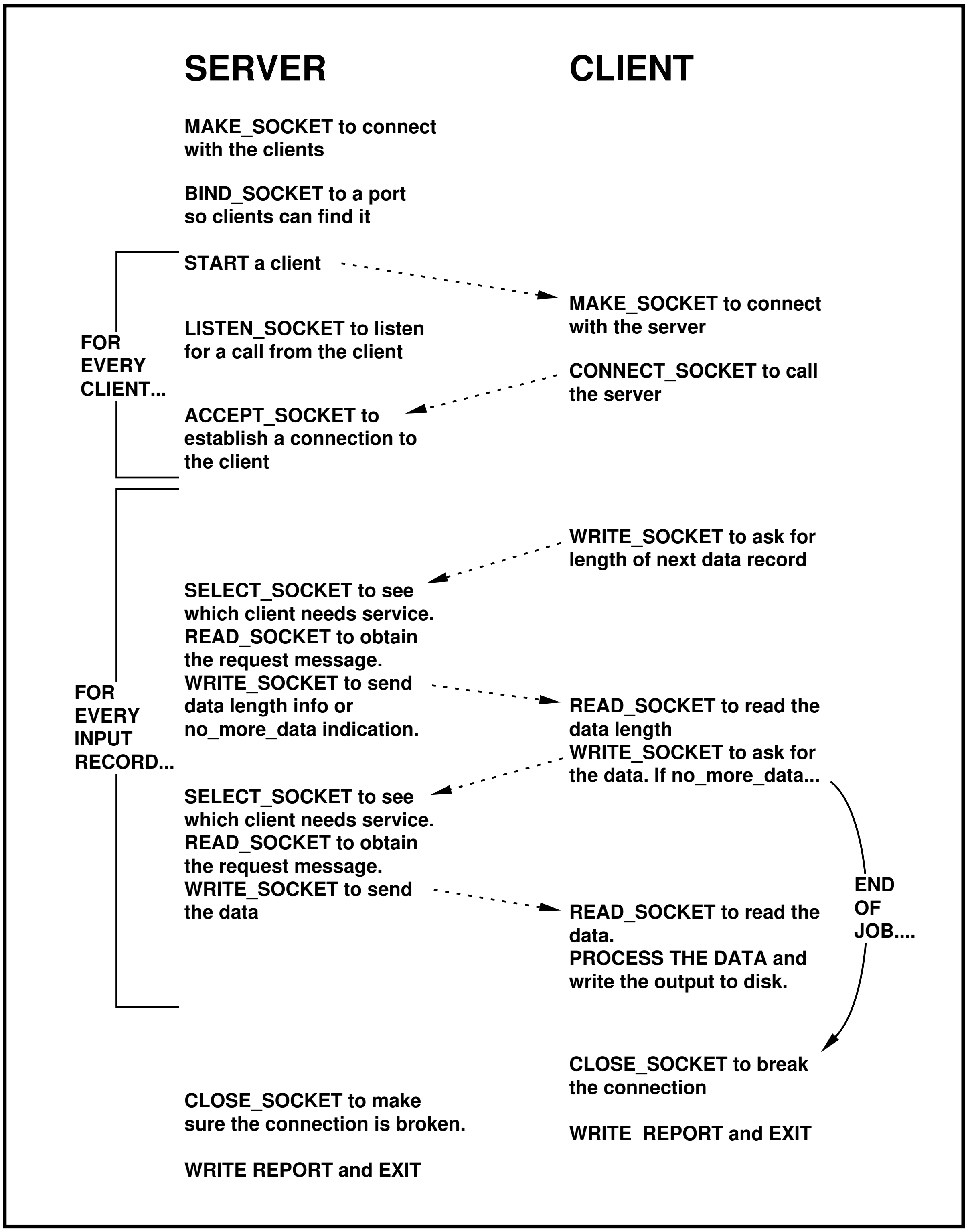

Fig. 3. Communication between server and clients using TCP/IP sockets.

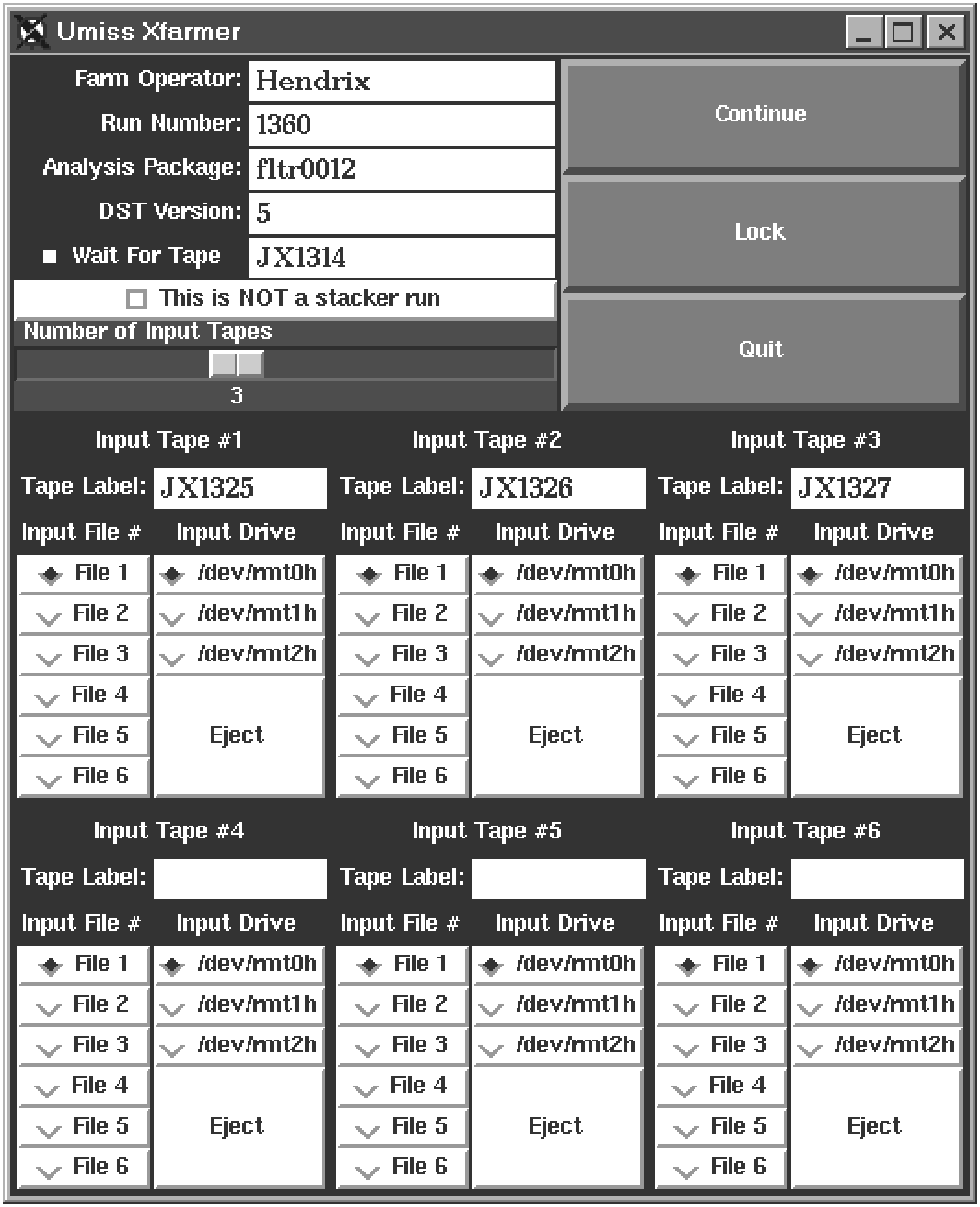

Fig. 4. X Window interface used to control input tapes. It was written in Tcl/Tk. Tapes numbers to be input for a whole run (up to 42 tapes) were prepared beforehand in a file. This minimized typing errors when running jobs.