Data Distribution Manager

C. Timmer, D. J. Abbott, V. H. Gyurjyan, W. G. Heyes, E. Jastrzembski, and E. Wolin

Abstract— **Jefferson Lab produces large amounts of data, up to 22 MB/sec. We developed a software package designed to manage and distribute all of this data as it is being produced - in real time. Called the Event Transfer (ET) system, it allows users to create data (events) and insert them into the system as well as allow other users to retrieve these events sequentially. The ET system has fast, local operation based on shared memory and POSIX threads and mutexes. Event transfer may also occur over the network to remote users. The ET system is designed to be easy to use as well as very robust. Although initially implemented in C on Solaris and Linux platforms, we completed a recent port to Java. This paper presents a description of this software package as well as some performance measurements.**

I. Introduction

The Thomas Jefferson National Accelerator Facility (Jefferson Lab) is a U.S. Department of Energy, nuclear physics research laboratory employing a 6 GeV electron accelerator . Of the various detectors used at our facility, the CEBAF Large Angle Spectrometer (CLAS) in experimental Hall B is the one whose operation currently places the highest demand on the data acquisition (DAQ) system. With over 40,000 channels and 30 FASTBUS/VME crates, a data rate of up to 22 MB/sec is possible. In the future, experiments in the proposed Hall D will produce an estimated 1000 MB/sec raw data rate and 100 MB/sec to tape. Hall D's raw data will have to be handled in several parallel streams.

To handle these data, the CODA data acquisition toolkit [1], [2] has been developed to run on Solaris and Linux systems. Briefly, CODA is composed of software components that communicate via the network and with a common database. The first of the four main CODA components is the readout controller (ROC) which runs in embedded controllers in FASTBUS or VME crates collecting raw data. ROCs send their data to the second component, the event builder (EB), which constructs complete events out of these data fragments. The EB, in turn, passes complete events to the event recorder (ER) which writes them to tape.

The Event Transfer (ET) system is responsible for passing these events between the EB and ER. The system is also used to pass the data to other users who may, for example, wish to monitor the data quality or do some physics analysis. In fact, the ET system is a general software package, which may just as easily be used independently of CODA. The PHENIX experiment at BNL's Relativistic Heavy Ion Collider as well as the MIT Bates accelerator also use the ET system to do their data transfer.

Manuscript received May 25, 2001. This work was supported by the U.S. Department of Energy.

II. ET System Design

Software for the transfer of data is such a basic and important building block for Jefferson Lab's DAQ system, it had to meet some stringent requirements. Namely, it had to be fast enough not to be a bottle-neck, extremely reliable, flexible, and physicist-friendly.

We achieved our speed requirements through a number of means. The POSIX thread (IEEE Std. 1003.1) library, also known as "Pthreads", was used to make the ET system a completely multithreaded, single Unix process. This allowed us to take full advantage of multi-processor computers. Currently CLAS runs their ET system on a Sun 3500 with 6 processors with excellent results. Another means was to use shared memory when transferring events between users on the same machine. Access to this shared memory is arbitrated by use of Pthread mutexes and condition variables, which are generally implemented in the users' memory space as opposed to the kernel and are therefore faster than SYS V semaphores.

A reliable system requires constant monitoring of users by the ET system and monitoring of the system by users. The ET system process and all users have a thread which provides a heartbeat and a second thread which monitors all other heartbeats. In this way, each can tell when the other has crashed or disappeared for some reason. A user is capable of waiting for the return of the ET system and continuing where it left off. The ET system, on the other hand, can recover the events that a crashed user was holding, place them where the user specifies, and continue on.

Flexibility and ease-of-use is due in part to making the ET system software completely reentrant, meaning that multiple copies may run on a single machine at the same time. No Unix environmental variables or static variables are used, eliminating a whole class of problems. In addition, because the ET library is thread-safe, there are no worries about the details of thread usage. Finally, users on remote nodes may receive events over the network with no change in their code.

III. ET System Implementation

Fig. 1 gives a general overview of the ET system. To start, empty data buffers are created in shared memory. In our case, these buffers are filled with event data and so we refer to these buffers as "events" for short. Arrows in the figure show the flow of events (actually pointers to events) through the system. The basic idea behind the flow is to have an ordered series of event repositories called "stations". Each station is primarily composed of two lists sitting in shared memory. The "input

All authors are with the Thomas Jefferson National Accelerator Facility, 12000 Jefferson Ave., Newport News, 23606, USA (telephone 757-269-5130, e-mail: timmer @jlab.org).

list" contains events available for use, and the "outut list" contains events users are finished with. The first station, called GrandCentral, is a special repository with all the unused events available to users to fill with data and put back into the system. Users that create these new events we call producers. Once produced, events are placed by the ET system process in other stations "downstream." Users wanting to read or modify the previously created events we call consumers and may "attach" themselves to stations downstream from GrandCentral where they can get and put these events as they please. Again, the ET system process moves the used events to the next station downstream. Once events reach the last station, they are recycled back to GrandCentral.

This flow of events is accomplished by multithreading the ET system process. Each station has its own event transfer thread, or "conductor", which is waiting for output events. When events are "put" by the user, the conductor wakes up and reads all events in the output list, determines which events go where, and writes them in blocks to each station's input list. A key optimization in the transferring of events from station to user and vice versa is to transfer a whole array at once. This reduces contention for mutexes in proportion to the number of events in the array and can result in the increase of speed by over an order of magnitude.

The use of threads has made complete error recovery possible. ET system and user processes each have a heartbeat thread which increments an integer in shared memory. Simultaneously, in other threads, the system monitors each user and each user monitors the system. If the system dies, users automatically return from any function calls that are currently pending. They can determine if the system is still alive, and can wait for the system's return. If a user's heartbeat stops, the system removes any trace of that process from the system while all events tied up by the dead process are returned to the system. These events can be placed in either: 1) the station's input list, 2) the station's output list, or 3) GrandCentral station (recycling them).

It is possible for multiple consumers to attach to a single station. In this case, each consumer receives only a fraction of the total flow of events through the station. One advantage of this configuration is that fewer stations means events flow through the system faster. Another advantage is that several identical consumers can operate simultaneously.

As for the actual events themselves, there are a number of ways to determine which are accepted into a station's input list. Each event has an associated header containing integers whose values may be set, effectively tagging them. Stations may choose to select events based on those tags using either a default algorithm or a user supplied routine. It is also possible to prescale so that every Nth event is chosen. Another means is to make a station "blocking" in which case it receives all events that match its selection criteria or "nonblocking" in which case it receives events only until its cue is full, at which time other events flow around it.

Occasionally, a user will need an event to hold a large amount of data - larger than the fixed space allocated for each event when the ET system was started and the event size was determined. In such cases, a request for a large event will cause a file to be memory mapped with all the requested space. When all users are done with it, this temporary event will be disposed of and its memory freed. This is all transparent to the user.

Events can be either high or low priority. High priority events that are placed into the system are always placed at the head of stations' input and output lists. That is, they are placed below other high priority, but above all the low priority items.

Part of the ET system's flexibility is its remote capabilities. Users can interact with ET systems over the network since each system has two threads dedicated to that purpose. One thread responds to the UDP broadcasts of remote consumers trying to find an ET system of a particular name somewhere on the network. The response simply sends back the port number of the socket that the second thread is listening on. The second thread, meanwhile, is the listening on a socket as part of a TCP server. That server, in turn, creates other threads which establish connections with consumers and handle general and event I/O with them.

Currently, the Linux kernel does not allow the sharing of Pthread mutexes and condition variables between processes. This makes it impossible to access the shared memory of the ET system safely between processes. However, this problem can be circumvented by treating local Linux producers and consumers as though they are remote. Thus it is the ET system's network capability that makes it possible to run on Linux (Redhat 6.0 and later). The server built into the ET system handles all ET routines that require handling these mutexes and send users pointers to events that can then be used to access events in shared memory. This makes ET systems on Linux somewhat slower than those on Solaris.

IV. ET System on Java

Although the words "Java" and "real time" are seldom spoken together, the performance of a Java-based ET (JET) system does not differ that markedly from that of a C-based ET system operating over the network. Since Java has no shared memory, an ET system running on it uses sockets for all communication. JET was initially implemented to be part of a slow controls system acting as a distributor of control information. It is currently being considered, however, to handle some of the CLAS experiment's main flow of data, because of the existence of data analysis software in Java. JET has only been used with Java version 1.3 from Sun, leaving open the possibility of performance increases through the use of Java compilers.

One benefit from having JET is the relative ease of implementing GUI's on Java. For example, creating a graphical ET system monitor was done quite easily using Java's built-in graphics and widgets available on the internet.

V. ET System Performance

Measurements of the ET system's speed in the handling of events can be seen in Fig. 2. We ran tests on both a 4 cpu, 250MHz Sparc UltraII Sun workstation running Solaris and on a 2 cpu, 450MHz Xeon PC running Linux. The conditions of the test were that the ET system had 3000 total events while a producer copied the event size amount of data into each event. A consumer created and attached to a blocking station so as to get and then put all events, but no manipulation of the data was done. The point was to simulate a bare-bones user application.

Notice that at the event size used in CLAS (5kB), the ET system can transfer about 200 MB/sec on the Sun and 160 MB/sec on the PC. That rate is limited primarily by the speed of copying data into the events. As the event size drops below 512 bytes, the inefficiencies of Linux mutex handling become apparent while Solaris does much better with these small events. Operating with the ET system connected to a consumer over 100 Mbit ethernet shows a rate of over 11 MB/sec, meaning that most of the available bandwidth is used. The Java ET system, at over 15MB/sec, does well in comparison to a true network-based consumer.

How the ET system holds up under multiple users can be seen in Fig. 3. Solid symbols denote consumers reading and writing events from a single station, while open symbols indicate performance when each consumer is attached to a different station. The conditions of the test were that the ET system had 3000 total events while a single producer was running. Stations were made to accept every event with each consumer's read and write containing 100 events. No copying or manipulation of data was done so as to clearly see the speed of the event handling itself.

As can be seen, with just a single consumer, an event rate of 550KHz is possible on the Sun, and a rate of 150KHz is possible on the PC. These numbers are somewhat arbitrary as the rate can be increased or decreased depending on how many events are written or read at one time. The large difference in the rates is due in part to the limitations Linux has in sharing mutexes between processes and the greater efficiency of mutex handling on Solaris. Notice also that when consumers all share the same station, the ET system operates much faster. This is because with fewer stations, the events travel through fewer conductors and have less ET system handling overhead.

Though JET performs well below C-based ET systems, when only one station is used for all consumers, the event rate actually stays at a very constant 25kHz. In fact, it performs at the same rate as a C-based system with 8 different stations and consumers.

The results of these tests, while accurate, do not reflect the conditions most users will impose on the system. Users of ET systems do more than just get and put events. Typically some analysis of the event data is done. When cpu intensive programs are figured into the mix, it is their speed and efficiency that determines at what rate the events flow through the system. In most cases, the overhead of the ET event handling is not the bottleneck.

One way to improve the efficiency of the ET system can be seen in Fig. 4. By simply transferring (getting or putting) an array of events at a time, instead of one-by-one, the performance can be boosted. At roughly 100 events per transfer most of the performance gains on Solaris have been achieved, while the Linux data suggest continued improvements with larger numbers of events per transfer.

VI. Conclusions

The ET system at Jefferson Lab has been running successfully for over 2 years. It is extremely reliable, is simple enough for inexperienced programmers to use, and meets all the demands placed on it at Jefferson Lab. Significantly, the bottleneck on the event rate due to the previous data distribution system has been removed, allowing rates currently limited only by front end hardware.

Future challenges facing the ET system will require handling the 1000MB/sec raw data rate of the proposed Hall D experiment. More immediate is the challenge of incorporating JET systems into the next generation of slow controls and data flow management.

VII. References

[1] G. W. Heyes, et al., "The CEBAF On-line Data Acquisition System," *Proceedings of the 1994 CHEP Conference*, pp. 122-126, Apr. 1994.

[2] D. J. Abbott, W. G. Heyes, E. Jastrzembski, R. W. MacLeod, C. Timmer, E. Wolin, "CODA Performance in the Real World," *Proceedings of the 1999 IEEE Real Time Conference*, pp. 119-122, June 1999.

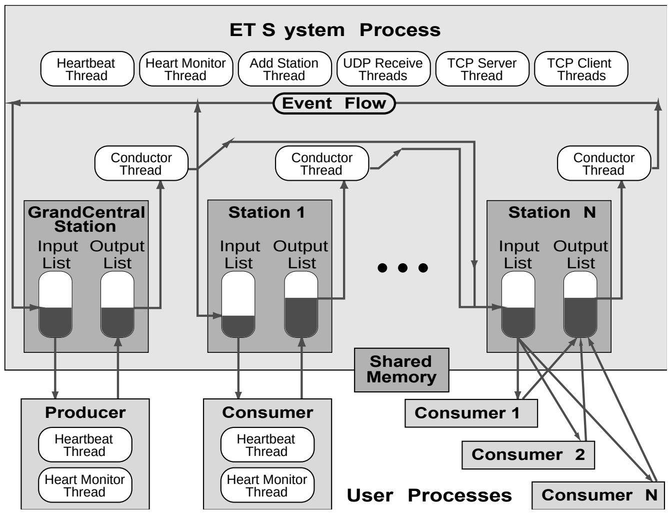

Fig. 1. The ET system architecture and event flow.

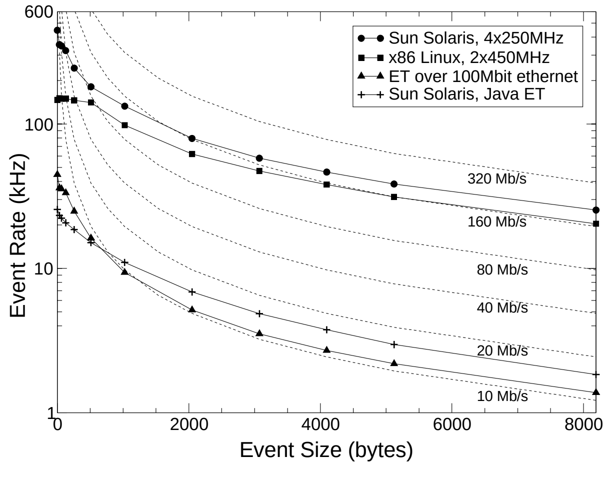

Fig. 2. The speed of the ET system in handling events is given as a function of the event size in bytes. Dotted lines mark fixed data transfer rates.

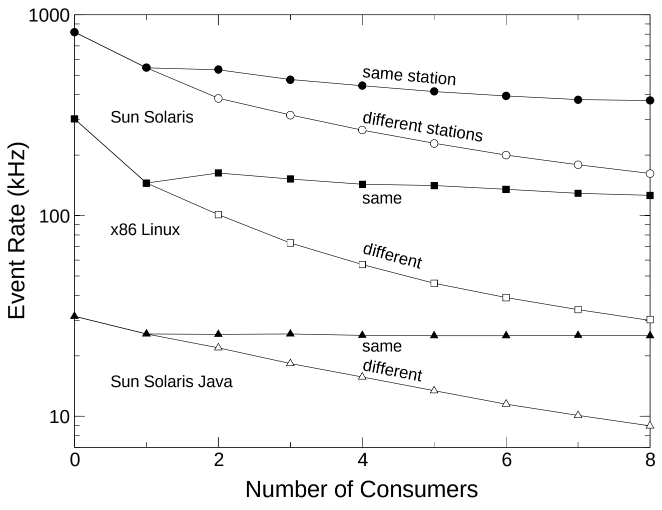

Fig. 3. The speed of the ET system as a function of the number of consumers. The performance for each platform is shown using the same station for all consumers and using a different station for each consumer.

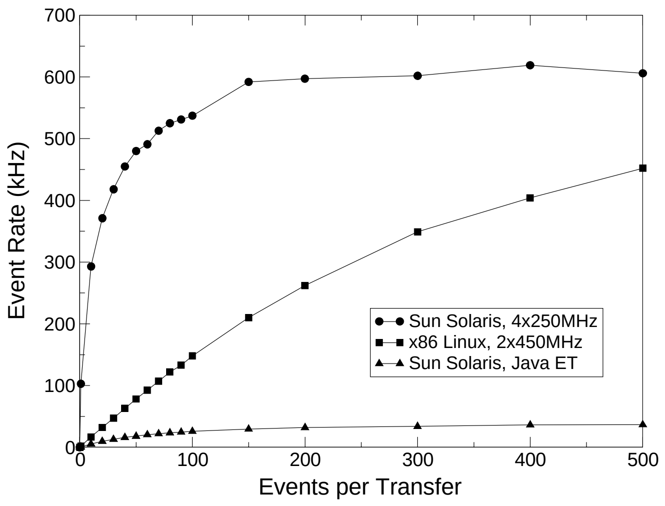

Fig. 4. The speed of the ET system as a function of the number of events in one transfer (one get or put call).