WWW Oriented Remote Job Submission, Monitoring and Management over Internet

G. A. Alves [a,1], M. Joffily [a,b], M. Miranda [a], A. Santoro [a,b], M. H. G. Souza [a]

[a] *LAFEX/CBPF*
R. Dr. Xavier Sigaud, 150
Rio de Janeiro, RJ, Brazil

[b] *Fermilab*
P.O.Box 500, Batavia, Il
60510, USA

We present a system which allows job submission over Internet with reliable data transfer using WWW graphical interface for job submission and monitoring and job management.

Key words: Network; Parallel Processing; WWW; Computing Farms

1 Introduction

The growing need for computational resources in big collaborations like Dzero, Zeus and LHC leads to the use of resources located in different sites. We have developed a system that allows one to submit and monitor a job over Internet using a WWW Interface. The system assumes there is a centralized server which receives job requests and sends them over Internet to a production site. The centralized server also functions like a temporary archiver for the input and output data.

The system is conceived to be used in any kind of job with the following characteristics:

- CPU intensive
- event oriented, with uncorrelated events

[1] gilvan@lafex.cbpf.br

– 1 input and 1 output event file (possible very large), besides calibration and control files

It is also assumed that the production site has been properly configured with calibration, control and other data files as well as any environment parameters. Also, the executable program must be available in the production site.

The system can be used for any program with the characteristics mentioned earlier. It can easily be customized, if required, to accommodate specific requirements of a particular system. Among the customizations that we can envision are elements for priorities, permissions, and parameters for the job. However, in most cases, almost no customization is required.

The system is composed of two main parts that will be explained in the following sections: the submission and monitoring system and the core system which implements the reliable data transfer and job control.

2 The Submission, Monitoring and Management System

The submission and monitoring system is based on a WWW interface, and so is the Management interface.

The submission system is layered in a three-tier architecture: a HTML page that holds the GUI, a CGI-BIN program that handles the requests generated by the user and makes entries in a requisition file and finally a daemon that controls the job submission.

The user submits a job by connecting to the WWW server in the Central Server. Information such as input data file, job parameters, user identification and priority are sent to the server, where the CGI-BIN program generates an entry in a request file. The submission daemon constantly examines the requests and starts the job when there is a production system available. Policies such as priority and user privileges can be customized. For such customization, a routine has to be written. A template with a FIFO policy is distributed with the system.

Once a job is selected for execution, the submission system starts the core system which in turn starts the job on the designated site. At the end of the job, the user is notified by e-mail.

Every job submitted receives a JobId which is used for monitoring the job. To monitor a job, the user connects to the status page of the WWW server. The monitoring system will tell if the job is waiting, running or finished. If the job is running, further information can be provided such as number of events already processed, status of the farm, and others. Also, it is possible for one to monitor the state of each CPU defined for a farm.

The management system allows one to check a specific node, add and remove nodes from a farm. The core system allows reconfiguration of the production site without having to stop any running job. By using the management interface, the manager of the site can add or remove nodes on the fly. When a node is removed, it finishes the processing of the event it is eventually working on, and then it is removed from the farm.

3 The Core System

The *Core* scheme, written in C++, implements *IO Classes* to deal with *Event* gathering, transmission, and submission. The primaries ingredients are the *IO Objects* which are defined as entities that *carry ONE Event*. They have three main "properties": a *Buffer*, where Event data are stored, a *Byte Count* that says about the actual size of the Event and an *IO Channel* which identifies the IO device where data will be read/write, which can be *file descriptors, pipes, Sockets*, etc. We can think them as a Class, which in C++ Language will read:

```
class io_operations {//This class defines objects with signatures:
protected:
   char *buf;
   int io_chan;        (or struct sockadd sock, for Sockets)
   int count;
                       // and methods acting upon :
public:
   io_operations (char *path, char *type);     //Constructor
   ~io_operations (void);     //Destructor

   void   set_buffer(char new_buf*, int new_count);
   char *get_buffer(void)  {return buf;}
   int    get_count(void) { return count;}

   void get_dat(void);
   void send_dat(void);
 };
```

These objects will be doing some IO, either reading *ONE* event and storing it on buffer, or writing *ONE* event from the buffer, from/to an IO device, which can be disk file, a Named Pipe or BSD Sockets.

As can be seen from the above code template, these informations are protected (encapsulated) from outside de Class scope. *Methods* are defined for a standard access to them, from outside. They are:

- Every time a Class member is defined, a *Constructor* method is invoked. It does all task of initialization, among them, an io_channel is opened. The argument *path* will indicate a *location*, for instance, a file name together of the directory or can be a *pipe* name or a *BSD Socket structure*. The second argument *type* will indicate when we will be doing reading or writing tasks. (Actually, in our implementation, we use different *Sub-classes* for each task, all inheriting the properties of the basic IO Class `io_operations`).

  In each case, the implementation is able, from these arguments, to make the correct identification of the device type and do everything needed to *open* it.
- *Methods* are defined to access encapsulated data in a standard protected way, for instance, the Event buffer: `char *get_buffer(void)` .
- Also, *Methods* are defined to make the appropriated IO operation: `get_dat` and `send_dat` . When reading, the buffer will be dynamically allocated in virtual memory. Of course, these methods should be able to know about event size. Also, they must know how to deal with different possibilities, other than experiment formats, following the *IO* device types.

An example template that will read data in some predefined format from tape and will write them to a *Named Pipe*, can be read as:

```
#include ...              // whatever includes you need
#include "io_header.h"    // IO Class definitions include file
main()
{
 class io_operations tape1("/dev/rmt0", "READ");
 class io_operations tape10("/local/userb/disk.dat", "WRITE");

 tape1.options("E740");
 tape10.options("PIPE");

  for (int k = 0 ; k < event ; k++)  {
    tape1.get_dat();

    cout << "Read phase byte count:" << tape1.get_count() << "\n";

    tape10.set_buffer( tape1.get_buffer(), tape1.get_count());
    tape10.send_dat();
  }
}
```

With the above defined *IO Classes*, we construct *IO Modules*, which will be the building blocks of the system. They are basically IO loops, following pretty much the above skeleton. There are two kinds of such modules:

- *Client Modules* read data available on *Named Pipes* or on disk files. To read each Event , an *IO object* will be created, which will hand over it to another *IO object* to carry this event and write it to a Socket.
- *Server Modules* are always alive, watching for income data on Sockets. When data is available on socket, *IO objects* are created, again in a event basis. When downstream *Named Pipes* are ready to accept events (which means that a *Client Module* is ready to accept data on the other side of the pipe), then another *IO objects* are activated to transmit them. This mechanism is the core of how the system do work: each event, in each module, is dealt by 2 different objects. When input data is present, objects will store them in virtual memory. When downstream IO is possible, *then and only then*, objects will be created to do data transmission. All allocations will be free and all objects destroyed.

From the core point of view, there are 3 elements: Central site, Server site and nodes.

- Nodes are CPUs that actually run the programs (huge fortran programs consuming several minutes of CPU time), expect input data (some hundreds of MB) and produce output data of same order of magnitude, roughly. Data is made available on Unix Sockets.
- In the Server site (a Unix machine) a file LOCAL_ RESOURCES defines completely the farm. The same *server* programs above are used to read input data (from the Central Site) on sockets, write it on named pipes, which are read by the *client* programs and shipped out to nodes (defined in LOCAL_RESOURCES) via Unix sockets. On the other hand, the *server* programs listen to sockets for returning data from nodes, write it on named pipes, which are read by *client* programs, in the same way as before. Data are then shipped back to Central Site. In short, there are 4 *Client/Server* programs running in this site (the same two binaries, with different parameters). They are all blocked on pipes and sockets, in absence of incoming/outgoing data. Once nodes need data or outgoing data is present on a socket, the transmission is made, in a event by event basis.
- Jobs are submitted from a Central Site, which may or may not be an Archive Site.

This model is very flexible for dealing with variable flux of data. Let us remind that each event are produced in a rate basis of (some) minutes/event and so, all overhead in making IO on intermediate fifos and sockets could be taken as negligible. So, in a normal scenario, we will have only one event transmitted over this system. Suppose now that several connections are done, each from a location that runs Monte-Carlo jobs. Several of those request can happen to arrive simultaneously. In this case, this flexibility shows its best qualities: all data will be accepted by the Server, freeing the upstream systems to resume its jobs with new fresh input. Data will then be transmitted from the Server to the Client through the fifo, in a steady event by event basis.

The system can catch and recover from Internet time-outs, broken pipes and so forth, without loosing events. The system also can sense for and recover from node program crashes, automatically. In this case, only the faulty event is lost.

4 Virtual Farms

It is worth to point out that the Central Manager and the Production site can be any UNIX machine connected through Internet. This means that one can build its local System using the available workstations. If the program's priorities are set low, one can build a production system using the otherwise unused cycles of the workstations on his institution. The nodes of the system can also be spread over the world, making a "Virtual Farm". Because it is assumed that the programs are CPU intensive, little load should be put on the network.

5 Conclusion

A pilot system is running now at Fermilab, as part of the overall Monte Carlo simulation effort of the Dzero experiment.

Prior experience in working with farms and farm's software, such as CPS [1] and Funnel [2], influenced this project. Specially, many good ideas, were borrowed from Funnel.

This project added to the well-succeeded CPS and Funnel projects the following characteristics:

- Object Oriented approach;
- Fault Tolerance in regard to network problems;
- GUI for job submission and monitoring and system management using WWW.

A last remark is that this system takes advantage of the lack of correlation between event processing, and allows the use of unmodified sequential application programs.

References

[1] F. Rinaldo and S. Wolbers, *Loosely Coupled Parallel Processing at Fermilab* , Computer in Physics, vol. 7, no.2 (1993) 184

PVM, Parallel Virtual Machine

[2] B. Burow, *Funnel: Towards Comfortable Event Processing*, Proceedings of Computing in High Energy Physics, CHEP95, Rio de Janeiro, 18-22 Sept. 1995