Farm Batch System
and
Fermi Inter-Process Communication and Synchronization Toolkit

M. Breitung, J. Fromm, T. Levshina, I. Mandrichenko, M. Schweitzer

Fermi National Accelerator Laboratory,
Batavia, Illinois, U.S.A.

Abstract

Farms Batch System (FBS) was developed as a batch process management system for off-line Run II data processing at Fermilab. FBS will manage PC farms composed of up to 500 nodes and scalable to 1000 nodes with disk capacity of up to several TB. It is designed to manage up to 200 simultaneously running jobs and release 20 to 200 jobs per hour. FBS allows users to start arrays of parallel processes on multiple computers. It uses a simplified "process counting" method for load balancing between computers. FBS has been successfully used for more than a year at Fermilab by fixed target expermints and will be used for collider experiment off-line data processing.

Fermi Inter-Process Communication toolkit (FIPC) was designed as a supplement product for FBS that helps establish synchronization and communication between processes running in a distributed batch environment. However, FIPC is an independent package, and can be used with other batch systems, as well as in a non-batch environment. FIPC provides users with a variety of global distributed objects such as semaphores, queues and string variables. Other types of objects can be easily added to FIPC.

FIPC has been running on several PC farms at Fermilab for half a year and is going to be used by CDF for off-line data processing.

Keywords:

1 Introduction

Large farms of inexpensive Intel-based computers running Linux OS are going to play increasingly important role as main tool for Run II off-line data processing at Fermilab. Computer power needs for Run II experiments are estimated to reach about 350 KMIPS by year 2001. Estimated size of PC farm used by each experiment is about 250 dual-CPU 500 MHz Pentium computers.

Farm Batch System (FBS) was proposed, designed and developed to be a batch process management and resource utilization control system that will be used for farm data processing.

2 Farm Batch System (FBS)

2.1 Requirements

FBS was developed as a batch process management system for off-line Run II data processing in the environment of PC farms. FBS will manage farms of PC computers composed of up to 1000 processors or up to 500 PCs. It will manage up to 1000 simultaneously running batch user processes. FBS will run arrays (sections in FBS terms) of parallel processes on farm nodes (worker nodes). Assuming that typical section will consist of 10 processes, and run for about 10 hours, in average, there will be about 100 simultaneously running sections. FBS will sustain job release rate from 10 to 100 sections per hour.

FBS will provide basic job control services:

- job submission and queueing
- job status monitoring
- job cancellations
- basic accounting
- monitoring of farm resource utilization

Since FBS will be used in the environment of large farms of relatively inexpensive and not necessarily too reliable computers, it has to be robust with respect to unexpected shutdown of some number of farm nodes as well as failures of FBS components.

2.2 Design

FBS is designed based on several assumptions.

2.2.1 Farm Model

FBS uses relatively simple model of a farm. Farm consists of computers (worker nodes) of one or more classes (node types). Computers within one class are identical, and FBS can pick any available of them for next user process to start on.

Computers are grouped into classes based on nature and amount of resources available on them. For example, a farm may consist of several multi-processor nodes (CPU nodes), some nodes with additional disk space attached to them (disk nodes), and some number of nodes with relatively fast network connection (network nodes). In this case, one can describe the farm as having 3 classes of nodes. Node classes can be used to logically partition a farm into non-overlapping parts (farmlets).

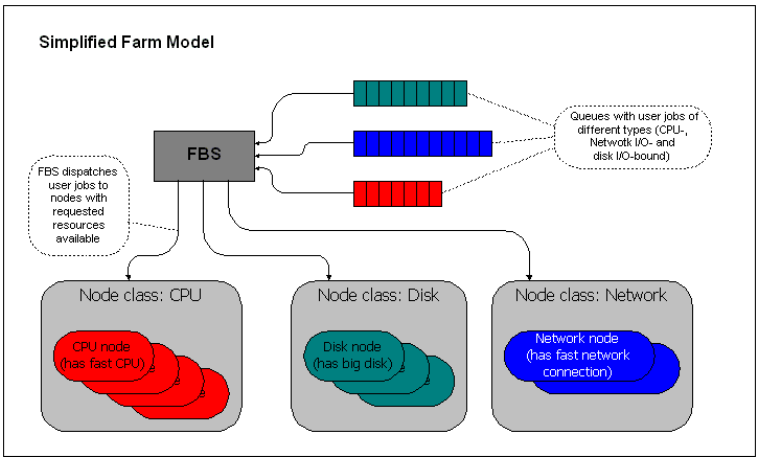

Figure 1: FBS Farm Model

2.2.2 Resources

FBS assumes that every worker node has certain amount of different resources available. In current version, only 2 resources are known to FBS. They are CPU power and local disk space. In future versions of FBS, users and administrators will be able to introduce arbitrary (abstract) resources. FBS user specifies what resources will be consumed by each of section processes, and FBS starts user processes based on availability of requested resources.

2.2.3 Load Balancing and Resource Management

As any batch system, one of FBS' functions is to evenly distribute load among worker nodes and prevent excessive use of resources. Important feature of FBS is that, unlike some other batch systems, it does not actually measure resource utilization. Instead, FBS uses "process counting" method for load balancing. FBS uses the concept of Process Type to allow users to specify what resources will be consumed by the process. Process Type can be viewed as a pre-defined bundle of resource requirements. The only resource which must be defined for every process type is persentage of CPU time the process is going to consume. FBS assumes that the requested resources are allocated by the user process as soon as it starts, and remain in use for the life time of the process.

On the other hand, process type may be associated with single user, group of users or a project. FBS leaves this interpretation open. FBS configuration defines limits on resources allowed for allocation to each process type.

2.2.4 Job and Section

User describes FBS job in Job Description File (JDF). FBS job consists of one or more sections. Section is an array of parallel processes, running on the same or different nodes of the farm. All processes of the same section are of the same process type. FBS starts all processes of the same section at the same time. Each section has its unique name within the job. User refers to a section by combination of numeric job number and section name.

User can specify dependencies between different sections of the same job. For example, user can request that section A does not start until section B finishes. FBS provides 4 types of dependencies:

- started (start section A only after section B starts)
- ended (start section A only when section B finishes)
- done (start section A only when section B finishes successfully)
- exited (start section A only when section B finishes unsuccessfully)

2.2.5 Major Components

In current design, FBS consists of the following components:

- FBS User Interface (UI). UI provides command line as well as graphical interface to FBS. UI allows user to submit, monitor and control batch jobs as well as monitor current status and availability of the farm.
- LSF (Load Sharing Facility) is a commercial software package distributed by Platform Software used as a component of FBS. LSF's responsibilities within FBS are job queueing and scheduling. FBS periodically reports availability of resources to LSF using ELIM.
- FLIMD is an FBS daemon that is responsible for keeping track of running processes, farm nodes staus, and reporting resource availability to LSF through ELIM.
- Job Manager (JM) is an FBS process that controls single section running on the farm. LSF starts JMs as LSF batch processes. JM is responsible for allocating resources on farm nodes with FLIMD, and communicating with FARMD to start user processes, and wait for their completion.
- FARMD is a FBS daemon that runs on each worker node. It creates environment for user processes, starts them as requested by JM, reports their status to JM and UI, notifies JM when user process exits.
- Historian is FBS historical database manager. It receives section start/exit statistics and stores it on disk. UI provides a tool for reading this database and generating reports as requested by user.

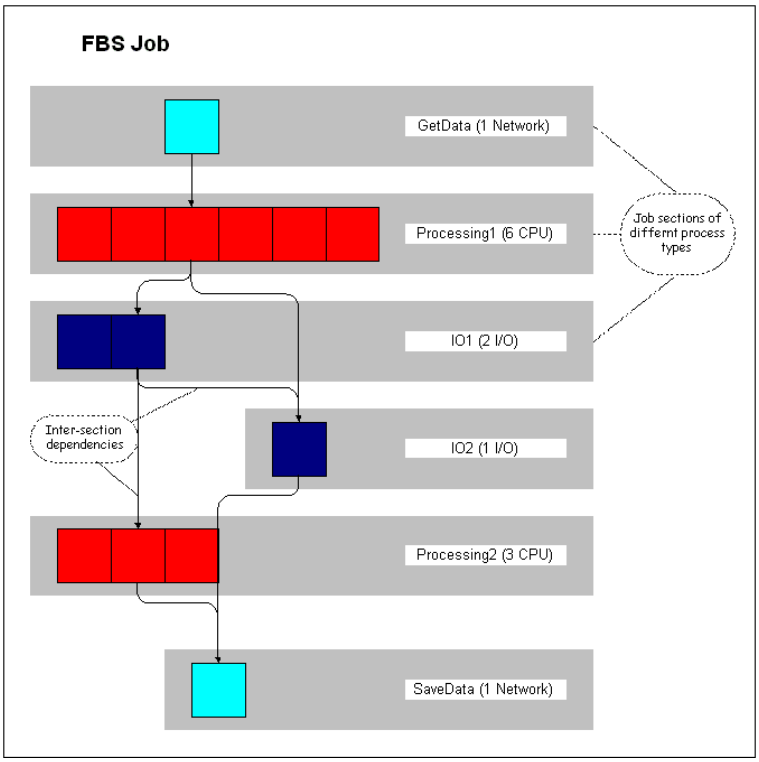

Figure 2: FBS Job Structure

- Logger or log daemon is responsible for receiving and storing error and event log information sent by other FBS components. This information is primarily used for FBS debugging and trouble shooting.

2.2.6 Sample Job Description File

The following is an example of FBS Job Description File. This JDF describes 3 sections, Init, Process and CleanUp.

First section is supposed to dump an input tape to disk. It is submitted to the queue named ”IO_QUEUE”. FBS configuration file specifies that sections submitted to this queue run I/O-bound processes. It consists of one process. This process requires 3 GB of local disk space on the workier node.

Second section is named ”Process”. It performs data processing. It is submitted into the queue designated for CPU-bound processes. It will start 5 processes, each will occupy 10 GB of local disk space. The section depends on Init section. It will not start until Init section finishes successfully.

Last section of this job is supposed to perform clean-up actions in case Process section fails. It will start only if Process section fails.

```
SECTION Init
QUEUE = IO_QUEUE
EXEC = my_bin/dump_tape.sh XYZ1234 /mnt/stage/XYZ1234
NUMPROC = 1
STDERR = /dev/null
```

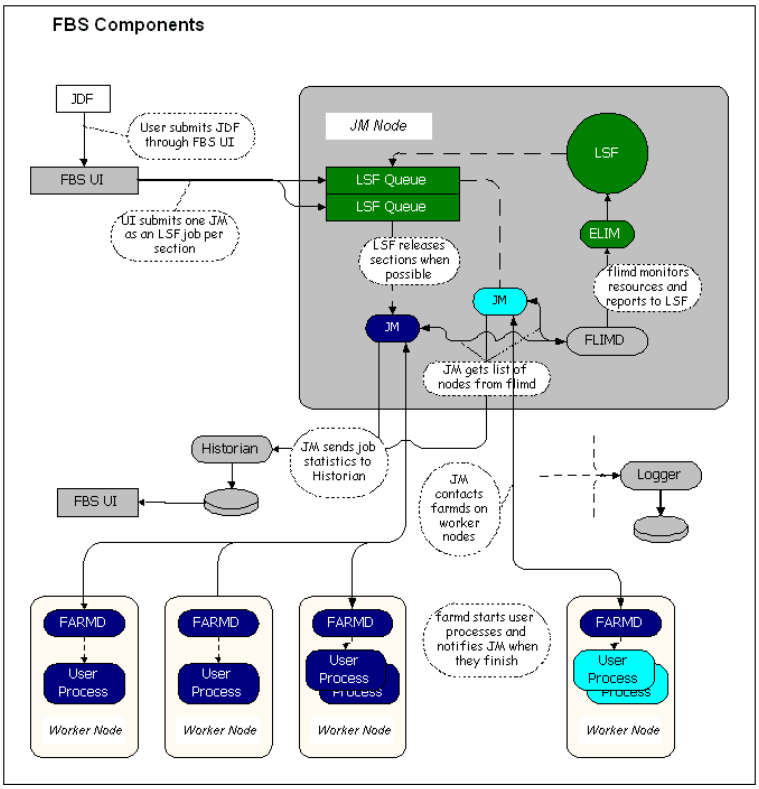

Figure 3: FBS Design

```
STDOUT = logs/%j.%n.out
DISK = 3

SECTION Process
QUEUE = CPU_QUEUE
EXEC = my_bin/do_processing.sh /mnt/stage/XYZ1234
NUMPROC = 5
STDERR = logs/%j.%n.errors
STDOUT = logs/proc_%j.%n.log
DISK = 10
NEED = 1
DEPEND = done(Init)

SECTION CleanUp
QUEUE = FAST_QUEUE
EXEC = my_bin/std_cleanup.sh /mnt/stage/XYZ1234
NUMPROC = 1
DEPEND = exited(Process)
```

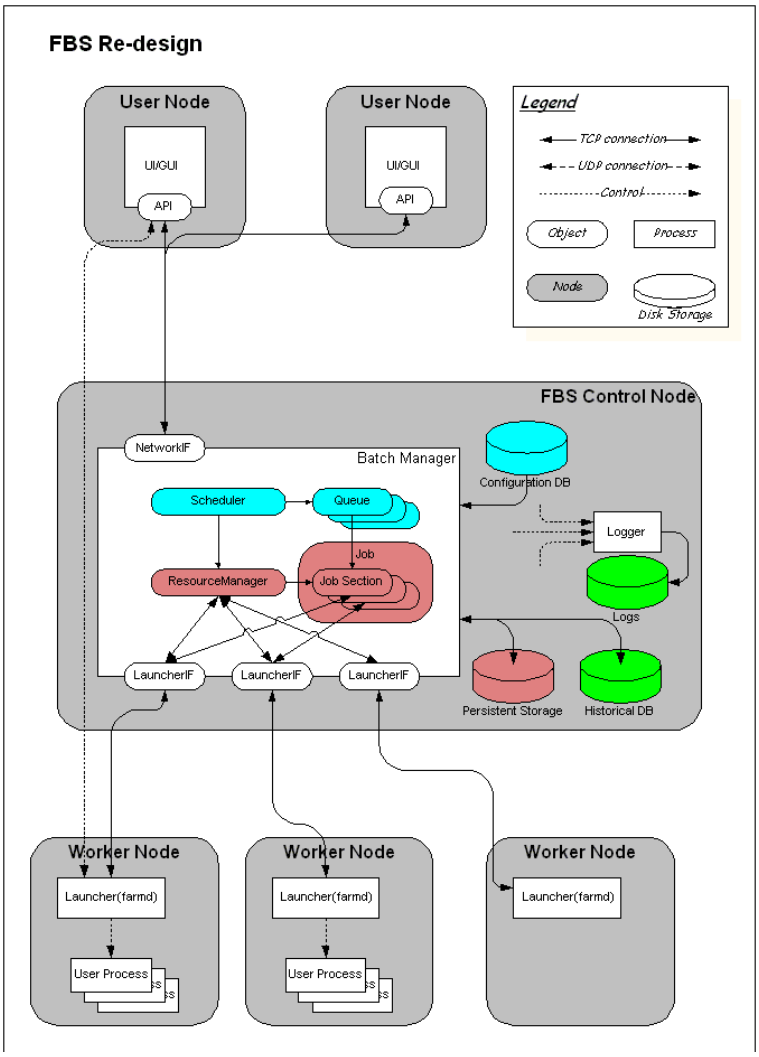

Figure 4: FBS re-design

2.3 Non-LSF Design

Currently our group is working on re-designing FBS. Main goal of this effort is not to use LSF as FBS component. This will allow:

- cleaner and simpler design
- more flexible and controllable scheduling algorithms
- introduction of user-defined ”abstract” resources
- reduce FBS support and maintenance costs
- simpler and more powerful user interface

We plan to have first version of non-LSF FBS some time in Spring 2000.

3 Fermi Inter-Process Communication and Synchronization Toolkit (FIPC)

3.1 Requirements

The idea of FIPC came up after first Fermilab users started using FBS for off-line data processing. First problem they faced was that sometimes too many processes running on worker nodes were trying to transfer data over the network simultaneously, and most of them simply were timing-out due to network bandwidth overload.

Obvious solution for this problem was to provide some sort of global distributed counted semaphore so that user processes would be able to block waiting for their turn to transfer data.

After this, the idea of global IPC package was developed further. The requirements for such system were:

- ability to work in distributed batch environment;
- robustness with respect to shutdown of one or more nodes of the farm;
- ability to handle large number of simultaneous clients;
- an ability to introduce other types of IPC objects besides semaphores;
- an ability for user to recover in a situation when a client shuts down unexpectedly without performing necessary clean-up;
- shell-level user interface as well as API.

3.2 Features

Currently, FIPC provides the following IPC object types:

- Gate is counted semaphore. It resembles an entrance to a room with limited number of seats available.
- Lock is simple binary semaphore object. Lock can be viewed as specific case of a gate.
- Client queue is a slightly improved first-in-first-out queue which clients can enter and wait for an access to certain resources.
- Integer flag is simple integer variable with a functionality of getting/setting value, waiting until the value of the flag becomes greater than, less than or equal to certain shershold, incrementing/decrementing flag's value.
- General purpose list has double-ended queue functionality. User can add and remove arbitrary text string items to head or tail of the list.
- String variable is just a text string. It provides basic string operations in terms of standard UNIX Regular Expressions functionality: getting/setting string value, waiting for value to match some pattern, atomic match-and-set operation.

Object names are organized in a name space similar to UNIX file system name space. User can create "(sub)directories" and place FIPC objects in them. The name space as well as FIPC objects themselves is global: all objects are visible from any node of the FIPC cluster.

FIPC objects have ownership and protection attributes. Users can protect their objects from unauthorized access to them.

All operations over FIPC objects are atomic: any operation (for example match-and-set operation on String) performed by a client on an FIPC object is guaranteed to finish before any other client can modify value of the same object.

Locks, gates, and client queues provide clean-up functionality: for example, if a client entered into a queue, and then unexpectedly shut down without removing itself from the queue, on user's demand, FIPC will clean the queue up removing all non-existing clients from the queue.

In order to provide this functionality, every FIPC client is assigned FIPC ID. There are two types of FIPC clients - single process clients and session clients. Single process client is simply a

UNIX process. Session client is a UNIX session - group of processes with the same UNIX session id. Session client is considered to be alive for the purposes of clean-up as long as there is at least one process of the session running on the client's node. Session clients should be used with FIPC command line interface whereas API users should use single-process client.

3.3 Design

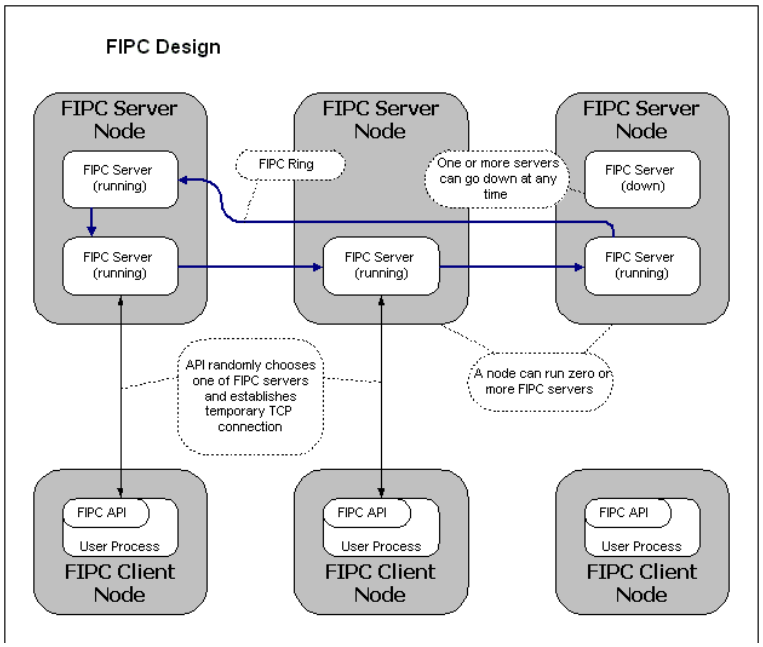

Figure 5: FIPC Design

FIPC consists of the following components:

- FIPC Servers. There are one or more FIPC Servers running on a farm or FIPC cluster. They connect to each other via TCP sockets to form a Ring. Each server connected to the Ring has exactly the same information as any other. This allows any server to disconnect from the Ring at any time without causing any loss of information. Servers can re-join the Ring at any time and recover all the information. IPC objects are represented as string variables in FIPC servers. FIPC servers provide very basic operations over these string variables.
- FIPC API is a library that provides access to FIPC servers and IPC objects. All IPC objects are implemented in terms of string variables provided by FIPC servers. All the functionality of various IPC objects is implemented in FIPC API. This allows to introduce new object types without shutting down servers, by just changing API and User Interface.
- User Interface provides shell-level command line interface to FIPC. It provides the same functionality as API, and implemented using API.
- FIPC Locals are simple daemons that are capable to answer whether particular user process or session is still running on the node. FIPC API contacts Locals when user requests a lock, gate or queue to be cleaned up.

3.4 FIPC and FBS

Although FIPC was meant to be a package used by FBS users, it does not depend on FBS, and can be used separately in any distributed or even non-distributed environment where it is neccessary to

implement simple communication and/or synchronization between processes.

3.5 FIPC example

The following is an example of a solution of the problem of readers and writers using FIPC: there are some number of proceses writing data into a file, and some number of processes that read the data from the file. Only one process, writer or reader should be allowed access to the file.

3.5.1 Writer Process

First writer process creates all necessary FIPC objects: an integer flag which will switch between writing and reading the file, and a queue for writers to wait in.

Then writer starts infinite loop performing the following actions. It appends itself to writers queue and waits there. If waiting takes too long time, writer request to clean the queue up in case some writer shut down without removing himself from the queue. When the writer becomes first in the queue, it waits for the read/write flag to indicate that the file is available for writing, and then writes the data into the file. After the file is ready, writer resets read/write flag to 0 to indicate that it is available for reading and removes itself from the writers queue. Then it starts all over.

```
#
# writer.csh - FIPC solution of readers-writers problem
#
fipc create flag /test/writing_f 1
fipc create queue /test/writer_q

while (1)
fipc append /test/writer_q
while (fipc qwait -t 100 /test/writer_q)
fipc clean queue /test/writer_q
end
fipc fwait /test/writing_f \> 0
write_file
fipc fset /test/writing_f = 0
fipc remove /test/writer_q
end
```

3.5.2 Reader Process

Reader process runs through symmetrical sequence of steps. Notice that both reader and writer have to initialize read/write flag to the same value to make sure that first process to get access to the file is a writer.

```
#
# reader.csh - FIPC solution of readers-writers problem
#
fipc create flag /test/writing_f 1
fipc create queue /test/reader_q

while (1)
fipc append /test/reader_q
while (fipc qwait -t 100 /test/reader_q)
fipc clean queue /test/reader_q
```

```
end
fipc fwait /test/writing_f < 1
read_file
fipc fset /test/writing_f = 1
fipc remove /test/reader_q
end
```