EVB2K Description

I. Assumptions / Requirements:

- Gigabit Ethernet on multiple interfaces (assume 2 input + 1 output to RCF). Additionally there will be another interface for the run control connection which may or may not be Gigabit.
- Goal is 200 MB/sec through the Ethernet interfaces and to disk, in absence of RCF writes.
- Code will run in the existing EVB machines or similar linux machines (2 cpu, 1.8Ghz or greater speed, ~8 striped IDE disks with > ~1 TB storage,)
- Push Architecture: no data source has any knowledge about any other source. The only exception is that the trigger interface "knows" what detectors were triggered.
- GB is the only entity that knows about every event. GB will mark any special handling that is required for the event builders. In particular, GB will indicate which events should be shipped to the event pool.
- Traffic routing will be accomplished without communication between nodes. This implies a simple round-robin scheme based on token numbers alone. The scheme will be to order the event builders. Then, tokens get routed: which_evb = (token % NEVBS*2)/2. Each event builder gets both even and odd tokens.
- Myrinet based detectors will be handled using one or more of the original event builders. The output of these event builders will be modified to ship entire ".daq" files as if they were a single Ethernet based detector.
- The data format will be a simple, low overhead format. The mechanics for reading banks will be independent of STAR and based loosely on regular file system operations. (ie. read("/event1/tpc/sector1/rb1/mz1/dat", &buff, sz);)
- There will be support for "BLOCKING" several events in the same Ethernet message to reduce latency for small events.
- Tag database support will continue.
- Run Control interface will remain unchanged.
- Output file streams will be trigger based. Each trigger will be assigned to a base filename. Each event will be copied into as many files as required based on the triggers it satisfies. Additionally, there may be indicators which modify the base filename, for example the existing "raw" readout files. These indicators may be based on the event contents or on markers sent by GB. Any such additional indicators will modify the base filename, but not affect the number of files the event is copied into.
- Existing DAQ logging and Monitoring will be maintained.

II. External Interface

From the perspective of the Event Builders there is only one type of data source. Every data fragment is prefaced with the following 64 byte header (defined in iccp2k.h).

srcNode		dstNode
idxFragment	nFragments	token
bankName		
bankName		
evtFlags		
trgMask		
detMask		
cmd		

srcNode: This is the source. This field is used to determine the detector/subdetector/instance of the data contribution

dstNode: The node id of the event builder used for debugging

idxFragment: Each detector might send more than one fragment per event. In this case, the event builder needs to know how many to expect. This field indicates which bank.

nFragments: This field indicates the number of fragments expected for this event. The maximum number of fragments from any node is 32.

evtFlags: Control Flags

EVTFLAG_CTL This is the contribution defines trgMask & detMask for the event. Typically, only GB will set this flag, but the EVB protocol does not care which node sets it. It is acceptable for more than one node to set this flag, if and only if the trgMask & detMask are the same from each node.

EVTFLAG_EVP Ship this event to the event pool

EVTFLAG_RAW This event contains TPC raw data

EVTFLAG_PREFORMAT The data section is one or more preformatted JSFS files, representing one or more data banks. EVB will not prepend file system headers

EVTFLAG_BAD The event is known to be bad by the sender.

bankName: The name of the file in event subdirectory. The path is constructed using the srcNode field. This field is ignored if EVTFLAG_PREFORMAT is set.

words: Words in the bank to follow. (not including this header).

trgMask: The mask of triggers satisfied by this event. This field only valid if EVTFLAG_CTL is set.

detMask: The detector mask of triggers satisfied by this event. The bits in this field are defined by the detectors rtsSystemId

cmd: CMD2_DATA : data fragment
CMD2_PING : ping message

When the full event has been built by EVB it returns tokens to GB using the following format:

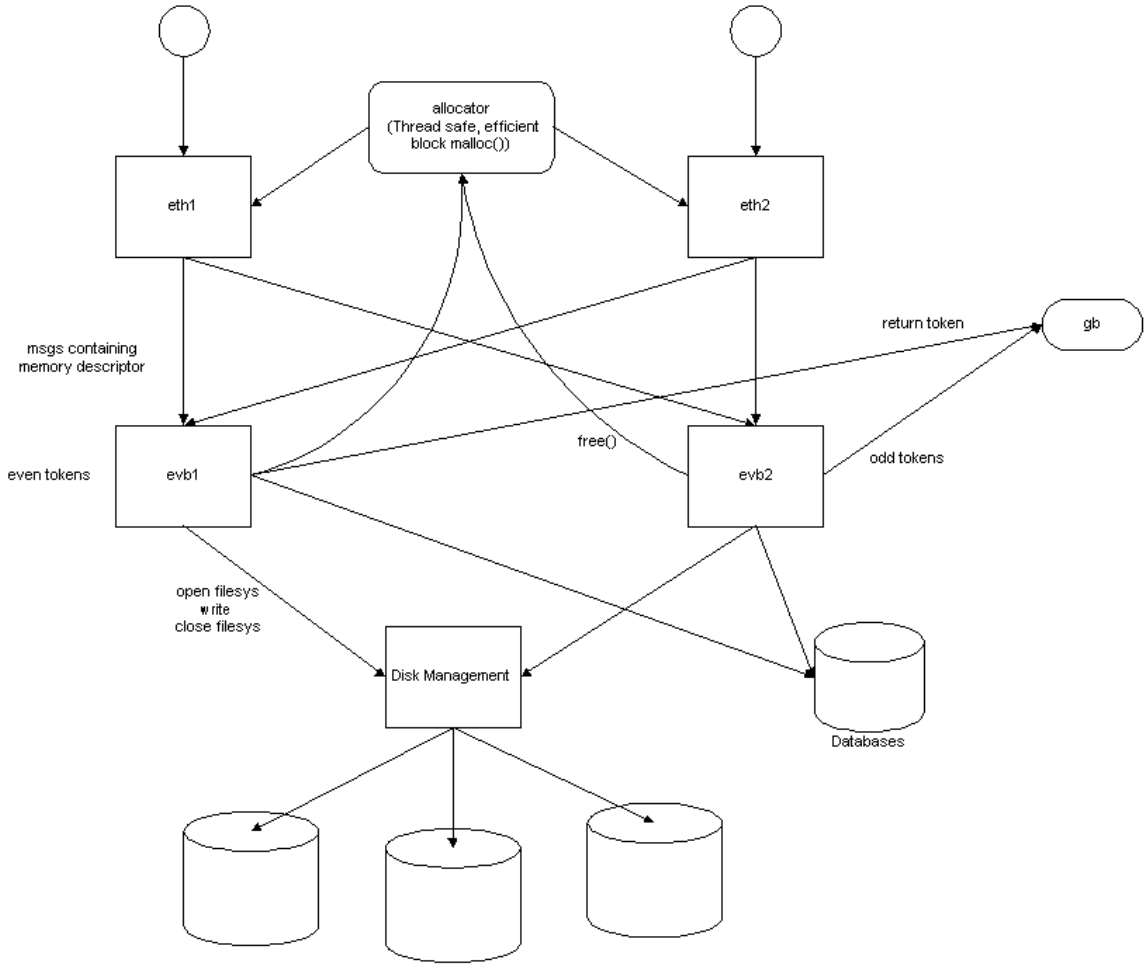
srcNode	dstNode
token	status

One feature worth noting is:

- The event builder attaches no significance to the ip address of the data source. Its uses the srcNode field to determine the source.

III. Design

The event builder will be a multi-threaded process with a block diagram as shown below:



General Features:

- The core EVB will operate as threads in a single process
- Two data paths will be implemented for two reasons:
  - To make use of multiple Gigabit Ethernet interfaces
  - To make use of multiple CPU's
- Thread blocking should be controlled so that error conditions can be detected, logged and monitored. Threads will have one of the two behaviors
  - All blocking calls will be protected by timeouts < ~2 sec.
  - Simple blocking threads for purposes such as disk writing will be monitored by a non-blocking thread.

- Communication between threads will use our standard system V based msgQLib.
- An attempt will be made to limit data copies, if it does not impact simple design. I currently expect 2 copies to be necessary:
    - Socket -> event buffer
    - event buffer -> disk buffer

  This should result in acceptable performance considering that the memcpy rate for these CPUs are > 500MB/sec. However, if it becomes necessary for performance, the event buffer -> disk buffer copy might be eliminated by using a more complex allocation scheme.

Eth

Eth are threads which handle the Ethernet connections.

- Act as Ethernet servers on EVB_DATA_PORT
- One Eth task per Ethernet interface
- Multiplex reads from all incoming sockets
- Allocate event memory
- Read sockets
- Route events to evb by writing memory descriptors to the evb queues

The Eth threads have no knowledge of or connection to run control.

Disk Manager

The Disk Manager is a class writes the data to disk. The data files will be standard LINUX files with a simple file-system like format (JSFS). Each data file is treated as if it were a file system. Each event is placed in a directory on this file system, with data banks treated as files within sub-directories.

The Disk Manager class supports openFileSys(), addFile(), closeFilesys() and fileSysStatus() calls.

The Disk Manager will:

- Use path-less “fileSys” names. The Disk Manager will assign files to physical disks.
- Do traffic shaping to avoid disk contention
- Handle the translation from standard binary blocks to the JSFS file format.
- Handle buffering / directio etc. efficiently
- Use file locks to communicate that files are in process of being built

Most likely, the final version of the Disk Manager class will have one or more internal threads handling the actual data storage, as well as a complex buffering scheme. However, these threads need have no connection with the evb threads.

The Disk Manager has no knowledge of or connection to run control.

EVB Task

The receives event fragments from the eth tasks and writes them using the disk management class. Furthermore, it must:

- Determine the destination filename(s) for each event
- Maintain the completion status for each event
- Handle timeouts and monitor / log errors & spurious events
- Return tokens to GB
- Free event memory after fragments are written to the disk manager
- Write events to the EVP
- Write the runtag, filetag, eventtag, and datasumary database tables.
- Construct an event summary data bank

(details.... how events are assigned to files etc.... how files are managed...)

RCF Writers

The RCF Writers will be identical to the existing writers. They will obtain files to ship by checking directories. They will obey file locks to avoid sending half-formed files as well a directory locks to avoid disk contention with the evb tasks.

IV. Disk Routing

The routing of events out of the event builders is complex because there are many special cases.

Pre-2006 Event Routing

There are 4 event builders named evb01, ... evb04. The evb01, is special in that it has a public Ethernet interface. . We use NFS to share the evb01 buffer disks to the starp network. There is no direct access from any other event builder to the starp domain so they do not have any publicly visible data drives.

Which EVB does the event go to?

The first stage of event routing is that GB decides which event builder to send events to. Events that satisfy any of the following criteria are sent to the event builder with the smallest index, so that they can be analyzed in the control room. The criteria are:

- The Run Control Field **glb_run.destination** is set to **BB_DEST_LOCAL_DISK** or **BB_DEST_COPY**
- The event is token zero.
- The event satisfies a trigger for which the field **trg_setup.triggers[].userdata.dataStream != 0**

If the event satisfies none of these criteria, gb assigns the event to an evb using a round-robin algorithm.

Is the event sent to the Event Pool?

The second stage of event routing is whether the event is sent to the event pool or not. Events are only sent to the event pool if there is a request sent to the evb node from the event pool. The request has a bit mask containing the following bits:

EVP_TYPE_PHYS
EVP_TYPE_SPEC
EVP_TYPE_0

Events always satisfy one of these classes. If the token is zero, it belongs to the EVP_TYPE_0 class. Otherwise, if the trigger command is 4 it belongs to EVP_TYPE_PHYS. If the trigger command is not 4 it belongs to EVP_TYPE_SPEC.

The determination whether ship an event to the event pool is made at the time the event is finished being built and is being sent for taping. For events of class EVP_TYPE_0 or EVP_TYPE_SPEC, if there is a request pending the event is shipped.

For EVP_TYPE_PHYS this simple-minded approach leads to a event-size bias for the events arriving at the event pool, because larger events tend to spend a longer time in the building process. The ideal situation would be to make the evp determination at the time the event is first announced, and this is what we will do for evb2k, however, for historical reasons I simply skip the first event satisfying the EVP_TYPE_PHYS class, and ship the second such event to the event pool. This removes most of the event-size bias, although the algorithm is not necessarily perfect because the event building is pipelined so event-size effects could in principle span the building of several events.

The Run Control Data DESTINATION parameter?

In run control, the user can select the "Data Destination". There are 5 possibilities

None
None Evb
DB only
RCF
RCF and Disk
Disk Only

The first three result in no data being saved. The difference between "None" and "None Evb" is esoteric & only is used for rate studies. When either of these are selected, it is assumed that the run is a test run and no tag database records are written. The "DB only" destination means that tag database records are written, but no data files are produced.

The "RCF", "RCF and Disk", and "Disk only" destinations are all handled identically by the event builders, except for a global prefix added to the name of every file written. For the "RCF" destination, no prefix is written. For "RCF and Disk" the prefix "COPY_" is used. for "Disk only" the prefix "LOCAL_" is used.

The prefixes are used by the rcfWriters which are independent processes which spool data from disk to HPSS. The only communication between the event builder process and the rcfWriter process is in the form of file locks which allow the event builders to prevent the rcfWriters from running. The rcfWriter processes ignore files with "LOCAL_" prefixes. They ship files with no prefix to HPSS and then delete. For files with "COPY_ prefix, they ship the file to HPSS, saving the file at HPSS with the prefix removed and then rename the local file to have the "LOCAL_" prefix.

File Names and use of the term dataStream

The STAR filename convention is the following:

st_NNNNNNN(_acd)_RRRRRRR_raw_XYYZZZZ.daq

NNNNNNNN: The ascii name of the data stream. Currently allowed values (according to our agreement with offline) are:

physics – "normal" physics data
pedestal – pedestal run
laser – laser run
pulser – pulser run
fast – fast detector data
express – express stream data
zerobias – zero bias data

(_adc):	events in this file contain TPC ADC data.
RRRRRRR:	The run number
X:	The event builder index for the event builder that built the event
YY:	An index corresponding to the data stream
ZZZZ:	The file sequence, a number increasing from 1 for each stream.

The data stream names are defined by the run control parameters **trg_setup.dataStreamNames[16].** I will also refer to DataStreamNames[0] as the "default stream."

The correspondence between the data stream (the ascii name) and the data stream index (YY) is dependent on the number of data disks on the event builder machine. In addition, there is a mapping between the data stream index and the disk used for the data stream. The full mapping is given in the table below.

Data Stream Index (idx)	Data Stream Name	adc data	Disk used
1 ... N_DISKS	dataStreamNames[0]	No	idx
N_DISKS+1 ... 2*N_DISKS	dataStreamNames[0]	Yes	idx – N_DISKS
2*N_DISKS + 1	dataStreamNames[1]	No	1
2*N_DISKS + 2	dataStreamNames[2]	No	1
....	....	....	...
2*N_DISKS + 15	dataStreamNames[15]	No	1

Note that the assumption here was that there would be a single "default" stream (dataStreamNames[0]) which would:

1. Contain the vast majority of the data
2. Not require analysis in the control room

At the same time there would be several calibration &/or express stream data which would:

1. Contain only small amounts of data
2. Frequently require analysis from the control room

The strange mapping in the table above was designed to avoid fragmentation of calibration &/or express stream data onto many disks to simplify control room access, while at the same time avoiding disk contention for the "default" streams by spreading the data to many disks.

Which data stream(s) are events sent to

For each trigger, the parameter **trg_setup.triggers.userdata.dataStream** defines the data stream by giving an index into the dataStreamNames[] array. I'll call this the data stream number. Another parameter, **trg_setup.triggers.userdata.calibration**, is defined. If this flag is set, it means that there is no need to make a copy of the event in the default stream for this trigger. Finally, the event builder gets a flag from GB specifying whether adc data is present.

Token zero does not satisfy any trigger. However, there is a additional run control parameter, **trg_setup.triggers.userdata.tokenZero.** For the purposes of the data stream determination, token zero is taken to satisfy every trigger for which this parameter is set.

For each event one examines every trigger satisfied by the event and compile a list of target data streams according to this table:

userdata.dataStream	rawFlag	userdata.calibration	target data stream(s)
0	0	x	default
0	1	x	adc_default
non-zero	0	0	dataStream & default
non-zero	1	0	dataStream & adc_default
non-zero	x	1	dataStream

The event is copied once and only once into every target dataStream arrived at by this fashion. If an event is sent to the default stream or to the adc_default stream, then the actual value of the Data Stream Index is obtained by a load balancing algorithm, consistent with the constraints given in the previous table.

Fast Detector Data

The previous explanations define the full event routing. Fast detector triggers are set up as follows:

1. The data stream is set non-zero, pointing to dataStreamNames[] = "fast"

2. The userdata.calibration flag is set on
3. The trigger conditions are set up such that there is no overlap of this trigger with an slow trigger

The result is that:

- Because dataStream is non-zero, events are treated as “express” events and sent to evb01
- Because calibration is non-zero, events are not copied into the “default” stream
- Because dataStream is non-zero, events are written to disk 1
- Because there is no overlap with a slow trigger, events are written ONLY into the “fast” stream

Express Stream Events

In a similiar way, express stream events are set up as follows:

1. The data stream is set non-zero, pointing to “express”
2. The userdata.calibration flag is set off

The result is that:

- Because dataStream is non-zero, events are sent to evb01
- Because calibration is zero, events ***are*** copied into the “default” stream

2006 Event Routing

For the next-generation event builders we will remove &/or simplify several of the constraints. In particular, we will:

- Remove the policy of public access to event builder disks (and implement a more flexible event pool policy, to alleviate the need for it)
- Stop attempting to avoid having multiple files with small numbers of events
- Remove the assumptions (which proved incomplete) about express/calibration/fast streams having small numbers of events
- Implement a more flexible EVP scheme, while at the same time offloading the decision process from the event builders

Which EVB does the event go to?

In 2006, the DETS do not communicate with GB or one another, so they must make the decision of which EVB to ship data to purely based on the token number.

Is the event sent to the Event Pool?

In 2006, there will be a flag in the data from GB for every event. If this flag is set, the event will be shipped to the event pool. We also intend to implement a "policy" based algorithms for GB to use to set this flag. One important policy that will certainly exist is "Every event", which will eliminate the need for public EVB disk access.

The Run Control Data DESTINATION parameter.

The meaning of these parameters will be unchanged.

Data Streams

Out experience with triggers has shown that the overlap between triggers is generally consistent from run to run for a given trigger configuration. In addition, because of the huge data samples taken by STAR, picking rare triggers out from the default stream is a time consuming process. The express stream concept has worked well for the few groups able to make use of it, but in practice when the trigger is complicated nearly all of the triggers actually qualify, or nearly qualify for the "express stream". For example in the year 2005 ppProduction configuration there were ~25 triggers. The largest rate for any slow trigger was <10hz.

The best generalization for the express stream concept is to simply map the trigger 1 to 1 with the data stream and copy each event into multiple streams as necessary. If we do this for the 2005 ppProduction data set we would obtain a 20% inefficiency measured by data volume due to double copies, but gain a very large improvement for groups wishing to analyze specific triggers. The inefficiency here, could be easily mitigated by

grouping similar triggers (defined as triggers with significant overlap) into the same data stream.

This method works seamlessly with the current configuration structures. We maintain the dataStreamNames[] array, as well as the userdata.dataStream parameter in the trigger configuration. The calibration flag becomes obsolete.

The filename production is also simple under this scheme. All fields stay the same, except for "YY", the data stream index. In 2006, there are likely to be at least two tasks assigned filenames per evb node, and we will maintain the _adc data flag, but the data stream index can be simply calculated according to the following formula:

YY = WhichEvbTask * 2 * NSTREAMS + stream + NSTREAMS * hasADC

V. JSFS Format

The JSFS is a simple file system format optimized for writing and for low overhead.. The advantages of this format are:

- Event navigation is possible using simple content-independent file system like functions.
- Very low overhead. No loss due to block size granularity
- Entire valid file system can be created by appending content

On the other hand, random access directory navigation is rather slow because there is no built-in indexing or directory hierarchy. For a 500MB file system containing files with 5k bytes this represents an initial search overhead of ~1-2 sec (~100,000 seeks).

JSFS Structure

The structure of a JSFS file is as follows

VolumeSpec	
Head	
File1	File1 Binary Data
File2	File2 Binary Data
...	...
Tail	

VolumeSpec: This is simply a 12 byte character string representing filesystem version. For example: "JSFS V00.01"

Head: This is a short header record. The byte order only applies to the time field of this record.

type = "HEAD"
byte_order = 0x04030201
time

File: The File records are a variable length record containing information about a file.

type = "FILE"		
byte_order = 0x04030201		
sz		
head_sz	attr	reserved
name....		
name (continued)....		

"byte_order" corresponds only to this header. The endiness of the data is undefined by JSFS

"sz" corresponds to the datafile size. This may be any number, but the file itself will be padded to take up a multiple of 4 bytes

"head_sz" this must be a multiple of 4

"attr" the only currently defined attribute is "JSFS_ATTR_INVALID" meaning the file has been deleted

"name" the name of the file.

Tail: The tail record has the same format as the "HEAD" record, but type="TAIL". This record is not needed, but can be present to represent that the file was properly closed.

JSFS Filename Convention

Because the JSFS file simply contains a list of file descriptors and binary data, paths may be thought of as adhoc creations of the JSFS reader code. However the following conventions are applied so that one can reconstruct a normal directory structure from a JSFS file:

/xxx/xxx/yyy	/xxx/xxx/ is the "directory part" yyy is the "file" part
/xxx	absolute path
xxx or xxx/xxx	path is relative to the directory part of the previous entry
xxx/	directory
0 length	the name of this file is used to reset the "previous entry", but contains no data

Because there need be no explicit directory entries, a directory is defined to exist if it is part of any existing file's path.

It is perfectly legal to have a "directory" which also contains data, although this is frowned upon.