Status of pbasf2

R.Itoh, KEK

Outline

1. Introduction
   - Change in design policy of pbasf2

2. Actual implementation
   a) Parallelization of ModuleChain processing
   b) Implementation of Input/Ouput module
   c) Histogram module

3. Discussion

4. Plan

1. Introduction

- Basf2(roobasf) has been restructured to have two layers.

- kbasf2 (basf2 kernel) serves as the module driver of the framework which manages the registration, parameter manipulation and execution of modules.

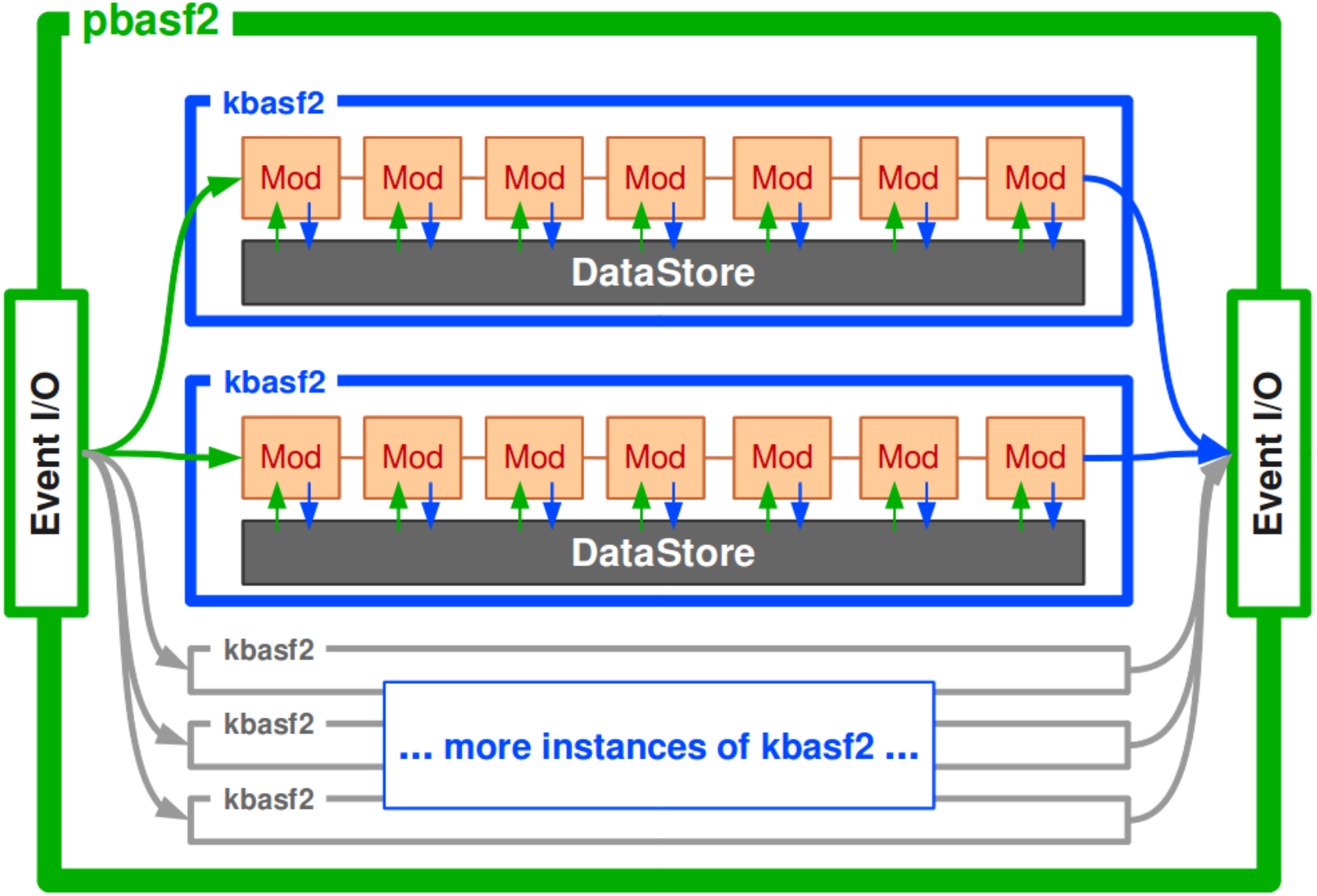
- pbasf2 (basf2 parallel) is implemented as a “super-framework” for kbasf2 and manages parallel processing of kbasf2.

- “basf2” is the generic name of pbasf2+kbasf2 complex.

- The execution of kbasf2/pbasf2 is invoked through Python script.

- Input/output data streams are managed by “modules”.
  <- separated from Framework.

- kbasf is supposed to run as a standalone application
  in a separate process



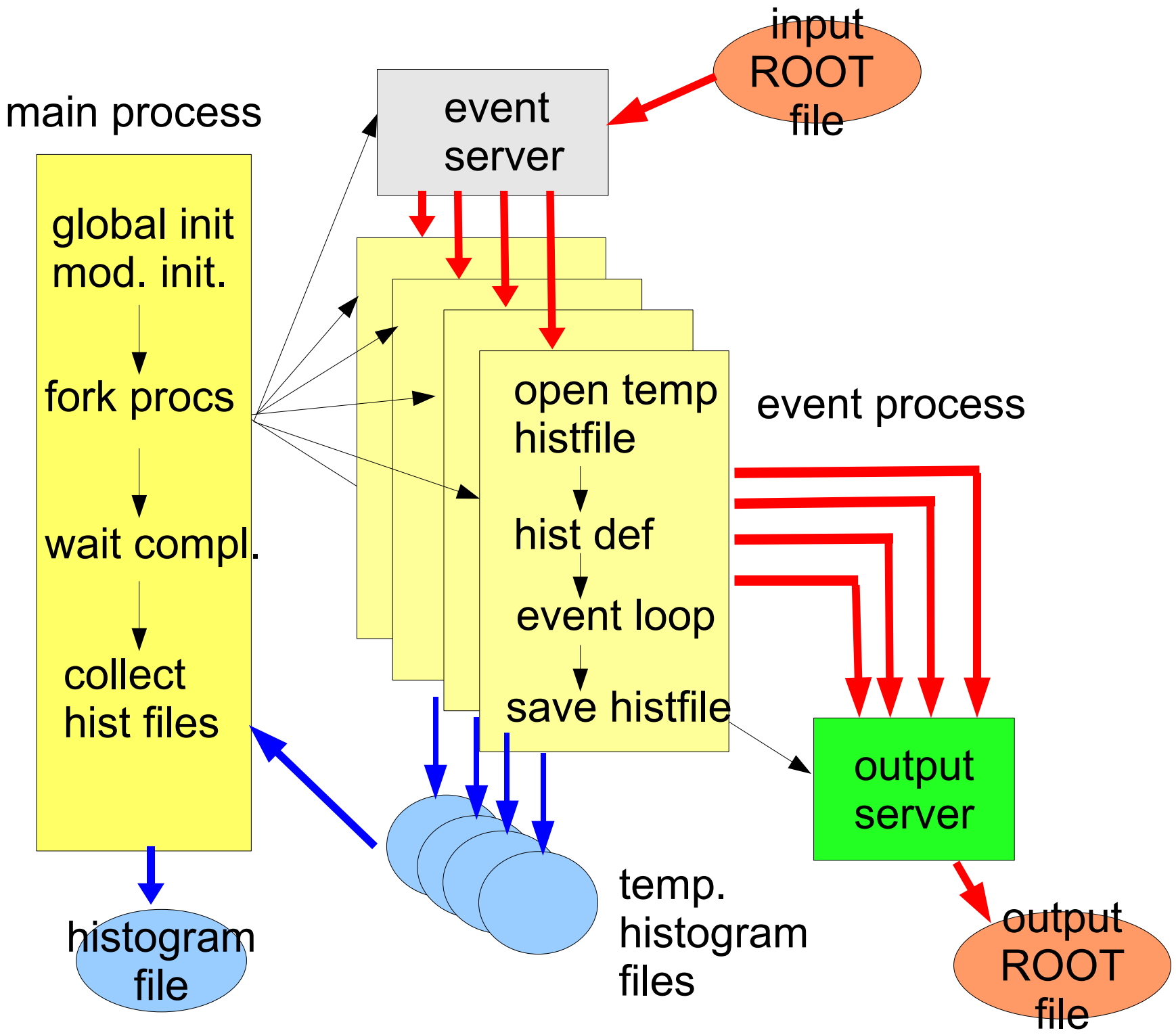
However,

* In this case, the module initializations (Module::initialize()) of kbasf2 is executed separately in multiple processes.
  -> If initialization includes a heavy file reading, data base access, etc., the overhead is multiplied by the number of processes.

  Suppose the use of pbasf2 on your desktop PC with multiple cores. In the current design of pbasf2, the initialization() is called from multiple cores independently which may introduce heavy I/O traffic.

* In the original roobasf design, the initialization is done in "mother process" and then event processes are forked off.
  -> useful to reduce the overhead by the initialization

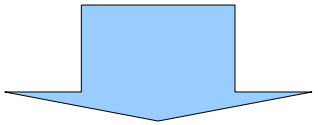
* Redesign pbasf2 to "emulate" original roobasf behaviour.


main process
global init
mod. init.
fork procs
wait compl.
collect
hist files
histogram
file
event
server
input
ROOT
file
open temp
histfile
hist def
event loop
save histfile
event process
output
server
temp.
histogram
files
output
ROOT
file

Parallel processing on multi-core CPU (pbasf2-core)

- Two operation modes:
  a) single process mode (Python global variable nproc=0)
    - pbasf2 internally calls kbasf2 functions as is.
      -> exactly the same behavior as that of kbasf2

  b) multi-process mode (nproc>0)
    - Module::initailize() is called in the main process.
    - Event processes (up to nproc) are forked off after the initialization.
    - Module::event() function is called using kbasf2's ModuleChain in each event process in parallel



Original “roobasf” behavior is kept by the implementation while keeping “kbasf2” as the core module driver.

2. Actual Implementation

a) Parallelization of ModuleChain processing

- New pFramework class is derived from kbasf2's Framework class

```
class pFramework : public Framework
{
 public:
  pFramework ( void );
  ~pFramework ( void );

  void process ( int nprocess, PathPtr startpath, unsigned
                 long maxevt );

}
```

- Python hook to pFramework::process() function is added.

```
void pFramework::process ( int nprocin, PathPtr spath, unsinged long maxev )
{
  if ( spath->getModules().size() == 0 ) return;

  ModuleChain* mchain = getModuleChain();
  ModuleList* modlist = mchain->getModuleList();
  PathList* pathlist = mchain->getPathList();

  //Build the list with all modules which could be executed
  ModulePtrList modulePathList = pathlist->buildModulePathList(spath);

  // 0. Single process case
  if ( nproc == 0 ) { // Single process -> fall back to kbasf2
      mchain->process ( spath, maxev );
      return;
    }
  }

  //  Multiprocess case
  // 1. Event process initialization
  mchain->initModules(modulePathList);

  // 2. Fork event servers
  pEventServerList* evlist = getInputModules();
  for(pEventServerList::iterator ii=evlist->begin();ii!=evlist->end();++ii ) {
    pEventServer& evserv = *evlist;
    procHandler->init_EvtServer();
    if ( procHandler->isEvtServer() ) {
      evserv->event_server();
    }
  }
  // 3. Fork output servers
  pOutputServerList outlist = getOutputModules();
  for(pOutputServerList::iterator ii=outlist->begin();ii!=outlist->end();++ii ) {
    pOutputServer& outserv = *outlist;
    procHandler->init_OutputServer();
    if ( procHandler->isOutputServer() ) {
      evserv->output_server();
    }
  }
```

```
// 3.2 fork event processes
procHandler->init_EvtProc ( nproc );
if ( procHandler->isEvtProc() ) {
  mchain->process_body ( spath, maxev );
  exit ( 0 );
}

// 4. Main process
if ( procHandler->isFramework() ) {
  procHandler->wait_processes();
}
```

b) Implementation of Input and Output modules

“Event server” class derived from Module class

```
class pEventServer : public Module
{
  // Constructor and Destructor
  pEventServer ( const std::string& type, boolselfRegisterType = true );
  virtual ~pEventServer ();

  // Member functions (for event processing)
  virtual void initialize() {};
  virtual void beginRun() {};
  virtual void event() {};                <- pick an event from ring buffer
                                             and place it on DataStore object.
  virtual void endRun() {};
  virtual void terminate() {};

  // Event server functions
  virtual void event_server () {};     <- runs as a separate process.
                                          read event from file and place in on a
                                          ring buffer.
};
```

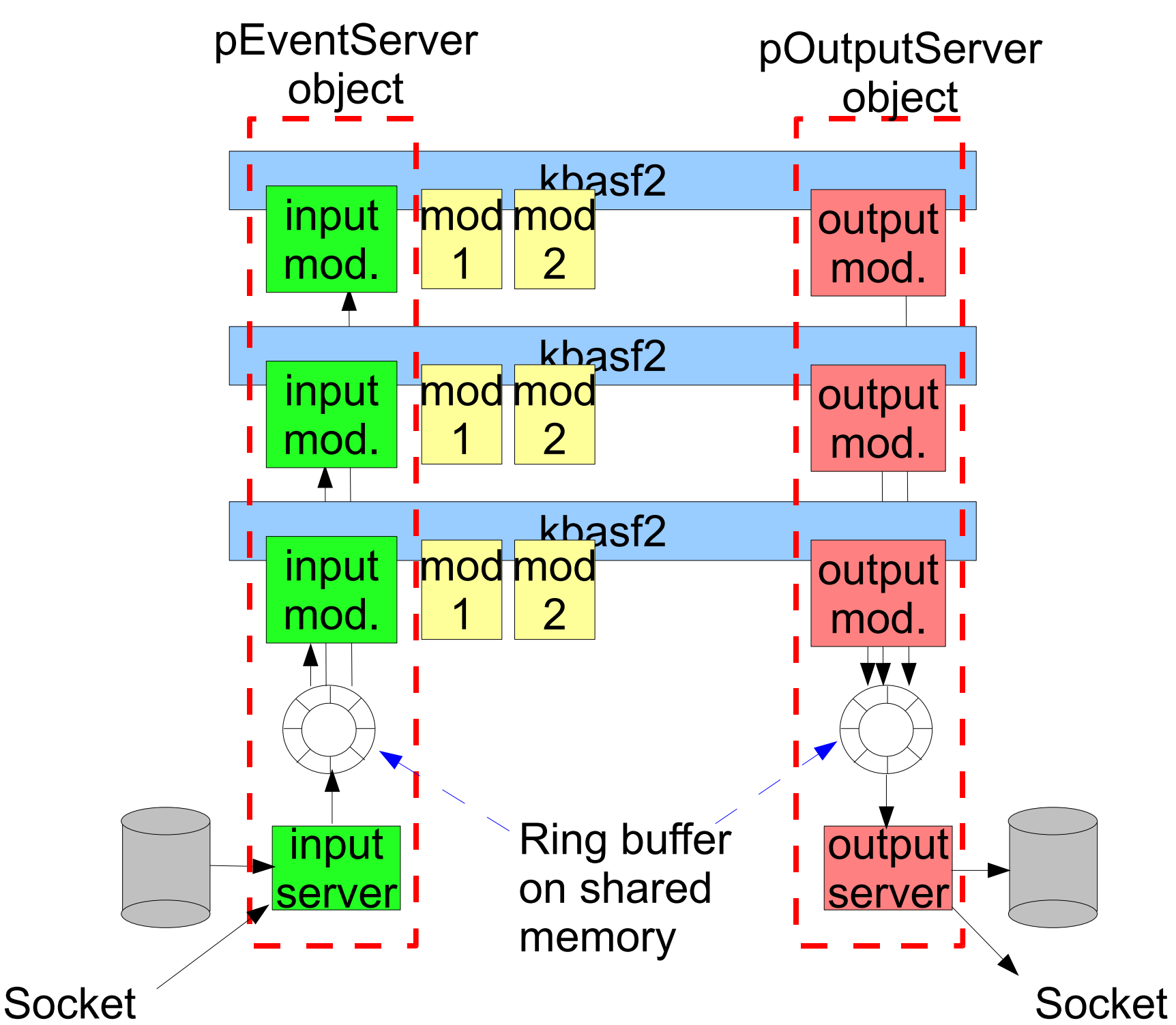
The module should be usable in kbasf2 also (single process mode).

Example: pRootInput <- Extension for “SimpleInput” for pbasf2

```
namespace Belle2 {
  class pRootInput : public pEventServer {

  // Public functions
  public:

    //! Returns a new instance of the module.
    virtual ModulePtr newModule() { ModulePtr nm(new pRootInput(false)); return nm; };

    //! Constructor / Destructor
    pRootInput(bool selfRegisterType = true);
    virtual ~pRootInput();

    //! Module functions to be called from main process
    virtual void initialize();

    //! Module functions to be called from event process
    virtual void beginRun();
    virtual void event();
    virtual void endRun();
    virtual void terminate();

    //! Module function to be called from event server process
    virtual void event_server();

  // Private functions
  private:
    //! gives back Null, if the branch isn't good for read out.
    TBranch* validBranch(int& ibranch, TObjArray* branches);

    ..........................
```

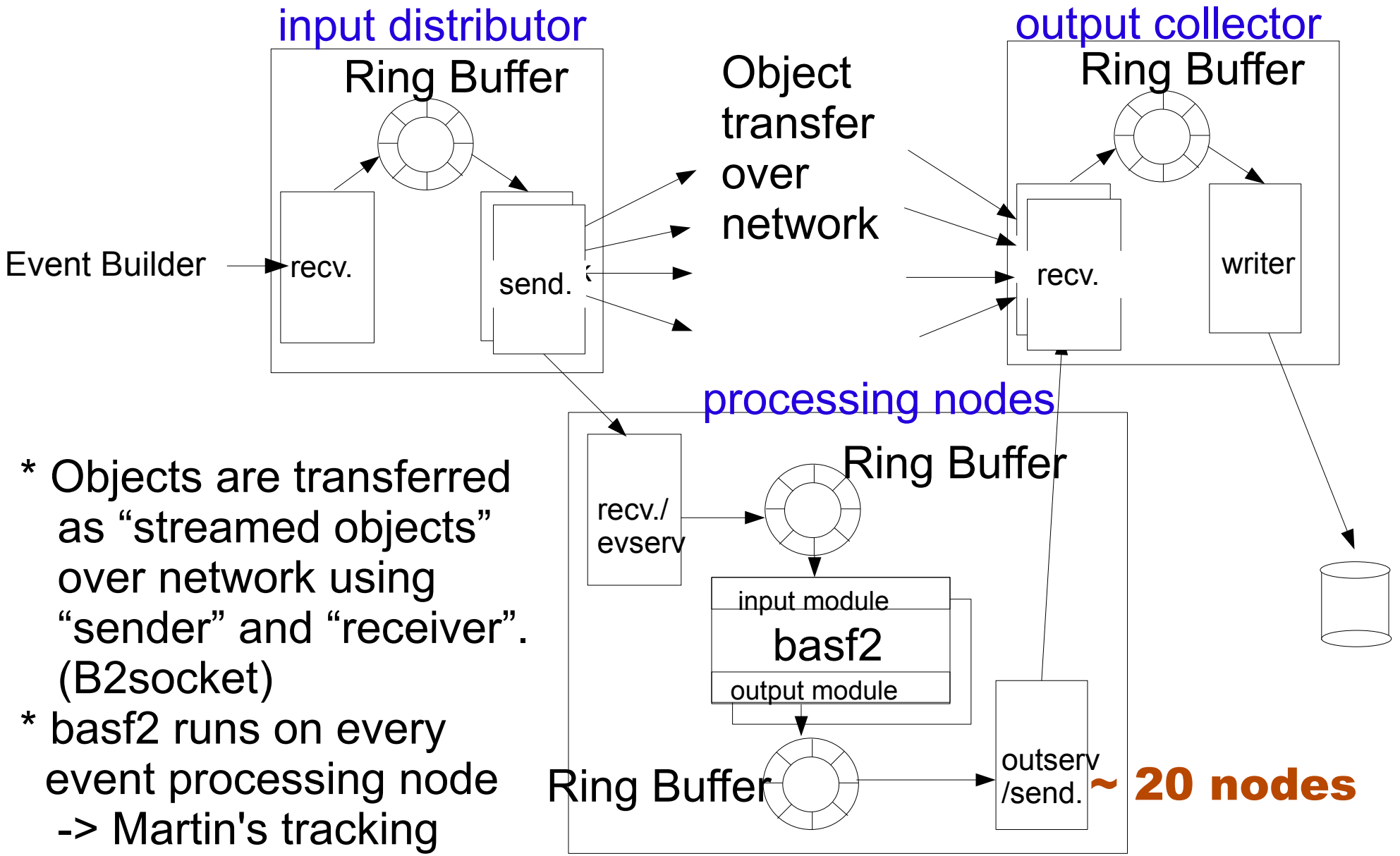
```
void pRootInput::event()
{
  if ( m_nproc == 0 ) {
    m_file->cd();

    readTree();
  }
  else {
    readRingBuf();
  }
  m_eventNumber++;
}
int pRootInput::readRingBuf()
{
  // Get a record from ringbuf
  int size;
  char evtbuf[MAXEVTSIZE];
  while ( ( size = m_rbuf->remq ( (int*)evtbuf ) ) == 0 ) {
    usleep ( 100 );
  }
  if ( size == 2 && *evtbuf == ENDOFFILE )
    return -1;

  // Build EvtMessage and decompose it
  vector<TObject*> objlist;
  EvtMessage* msg = new EvtMessage ( evtbuf ); // Have EvtMessage by ptr cpy
  int status = m_msg_handler->decode_msg( msg, objlist );

  // Store objects in the DataStore
  for (int ii = 0; ii < m_sizeObj; ii++) {
    DataStore::Instance().storeObject(objlist.At(ii), m_objectNames[ii]);
  }
  // Store arrays in the DataStore
  for (int jj = 0; jj < m_size - m_sizeObj; jj++) {
    DataStore::Instance().storeArray(static_cast<TClonesArray*>(
                    objlist.At(jj+m_sizeObj)), m_objectNames[jj]);
  }
}
```

```
void pRootInput::event_server ( void )
{
  while (1) {
    if ( m_eventNumbder > m_nevt ) {
      // Send termination record
      EvtMessage* term = m_msg_handler->encode_msg ( prt_EndOfData );
      for ( int i=0;i<m_nporc;i++ ) {
        while ( ( stat = m_rbuf->insq ( (int*)(term->get_msg()),
                  (term->get_msg_size()-1)/sizeof(int)+1 ) ) < 0 ) {
          usleep ( 200 );
        }
      }
      exit ( 0 );
    }

    // Fill m_objects
    m_tree->GetEntry(m_eventNumber);


    // Store objects in the DataStore
    for (int ii = 0; ii < m_sizeObj; ii++) {
      m_msg_handler->Add ( m_objects[ii] );
    }
    // Store arrays in the DataStore
    for (int jj = 0; jj < m_size - m_sizeObj; jj++) {
      m_msg_handler->Add ( m_objects[jj+m_sizeObj] );
    }

    // Encode event message
    EvtMessage* msg = m_msg_handler->encode_msg( prt_Event );

    // Put the message in ring buffer
    int stat = m_rbuf->insq ( (int*)msg->get_gsg(), get_msg_size()/4+1);

    // Count up event numbder
    m_eventNumber++;

  }
}
```


pEventServer
object
pOutputServer
object
kbasf2
input
mod.
mod
1
mod
2
output
mod.
kbasf2
input
mod.
mod
1
mod
2
output
mod.
kbasf2
input
mod.
mod
1
mod
2
output
mod.
input
server
Ring buffer
on shared
memory
output
server
Socket
Socket

pbasf2 on PC-cluster (=HLT)

S.H.Lee is now working on.



* Objects are transferred as “streamed objects” over network using “sender” and “receiver”. (B2socket)
* basf2 runs on every event processing node
  -> Martin's tracking framework runs “as is”.

c) Histogram Module

* Common for kbasf2 and pbasf2
* Takes care of histogram collection within a basf2 session
* Derived from a module class.

- HisogramMangaer module is placed on the top of the module chain.

- A ROOT histogram file is opened by a call to HistogramManager::initialize()

- Definitions of ROOT histograms/TNtuples/TTrees of each module follows by calling Module::initialize() functions

- The ROOT histogram file is closed by a call to HistogramManager::terminate()
  Histogram collection is invoked for multi-process case.

..... Problem forseen in parallel procesing mode with new pbasf2 design (initialization in main process.) -> under consideration

3. Discussion

- DB access from initialize() and beginRun() function of a module.
  Assumption:
    * Access to Geometry database -> initialize() function
      <- basically exp-by-exp. Not run-by-run.
    * Access to Calibration database -> beginRun() function
      <- run-by-run access.

- In pbasf2, Module::initialize() is called from main process, while Module::beginRun() is called from each event process.

- Assumption is that the access to calibration DB is lighter than that to Geometry DB.

- If the access to calib. DB is not so light, calling initialize() in main process does not gain so much
    -> If no, move back to original pbasf2 design (= call Module::initialize() in each event process)
  * But anyway, I will implement both options in pbasf2.

- Another idea : DB cacheing on every node.

4. Plan

- Some working prototype by soft/comp workshop in Krakow
  -> already behind schedule..... not yet succeeded to make prototype work.

- Interface to S.H. Lee's HLT framework by the time of the special “framework/basf2/roobasf meeting” in KU (Jun.24)
  c.f. framework/basf2/roobasf meeting is held regularly on every Wednesday starting at 10:00AM(JST). You can join at Polycom 30325.

- Prototype release to (limited) users at the time of next B2GM.
  -> I would like to keep this.....

- Input/Output modules for DAQ use (Sequential ROOT IO package) will be implemented during summer to be used in HLT test.