Version

15.1

JEFFERSON LAB

Data Acquisition Group

Event Transfer (ET) User's Guide

JEFFERSON LAB DATA ACQUISITION GROUP

ET User's Guide

Author Carl Timmer
&
Updated by Graham Heyes

25-Jun-2015

12000 Jefferson Ave

Table of Contents

Section 1

1. Introduction to the ET system

1.1 C-based ET system

1.1.1 On local host

The Event Transfer System is an efficient and fast mechanism for transferring events from computer process to computer process. In this context an event is a memory buffer that can be filled with data and tagged with information describing the contents.

Each ET system consists of a master process which memory maps a file into its memory space. This mapped memory contains all the event buffers along with the necessary bookkeeping data. The ET system creator sets the number and size of the events with all events being the same size. Users of the ET system that are on the same node as the ET system will transparently map the same memory which allows for quick communication between processes and forms the foundation of the event transfer system. Users of the ET system on different nodes will transparently communicate with the system over the network using TCP/IP as the system has a server built in.

In order to get an understanding of how things work, think of the ET system as a circular railway line. Events are loaded onto a “train” which starts at Grand Central station. At each station along the line, events can be unloaded - that is, passed to a user. When the user is finished with an event, it is put back onto the next train at that same station. Once an event has made its way through all stations, it is returned to Grand Central. Then the journey starts all over again.

To access events, a user process attaches to a station that it has already created or already exists. It receives a unique identifier for the connection that it can then use to read and write events. Events can be read or written either singly or in blocks (i.e. arrays).

Referring to figure 1.1 below, a station can be thought of as two lists: an input list of events to be read, and an output list of events that are ready to be sent to the next station. Stations, in turn, are themselves arranged into an ordered list. Events pass from station to station until they reach the last station in the list and are then returned to the first station.

Stations are configured to determine which events they will receive based upon the tags added to the events. They are also configured to determine how they will behave when a matching event arrives, for example stations can be grouped to work in parallel, collect sample events in a non-blocking mode or accept all matching events in blocking mode.

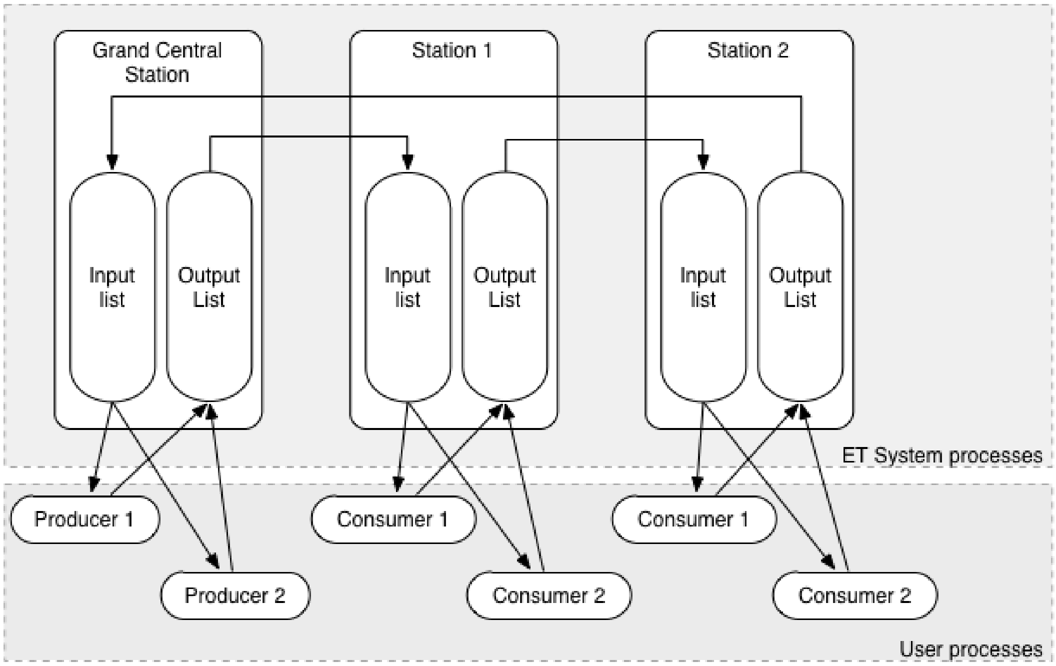


Figure 1.1: ET system layout

The first station is special and, for lack of a better name, is called Grand Central station. This station is created automatically when starting up an ET system and is a repository of all unused events. All other stations are created by users and may be placed in the list in any order after Grand Central.

User processes use functions from an ET system library to connect to or open the ET system. The user can attach to any station. Once attached to a station, a process can read events from and write events to it. The user process can detach from stations, remove stations, and close the ET system. All attached processes must be detached before a station can be deleted from the system. Grand Central station (the first and automatically created station) can never be deleted.

A process can attach to several stations, and it will receive a unique identifier for each station that it is attached to. In this document, processes that write data into event buffers thereby creating data are called producers, while processes that are interested in reading, analyzing, or modifying data produced by others are called consumers. Producers typically attach to Grand Central and request new events. The distinction between consumers and producers is not rigid since any attached processes can request new events and write them into their own stations.

A user process gets an event from a station's input list and, when done, places it into the station's output list. Each output list has enough space to contain all events in the system. Thus, a user can always put events since there will always be room. However, reading events from a station may block if the input list is empty.

Figure 1.2 below shows the flow of events within the ET system process. Each station has its own event transfer thread - or conductor - which is waiting for output events. When an event is written into an output list, it wakes up the conductor, which reads all events in the list, determines which events go where, and writes them to the appropriate station.

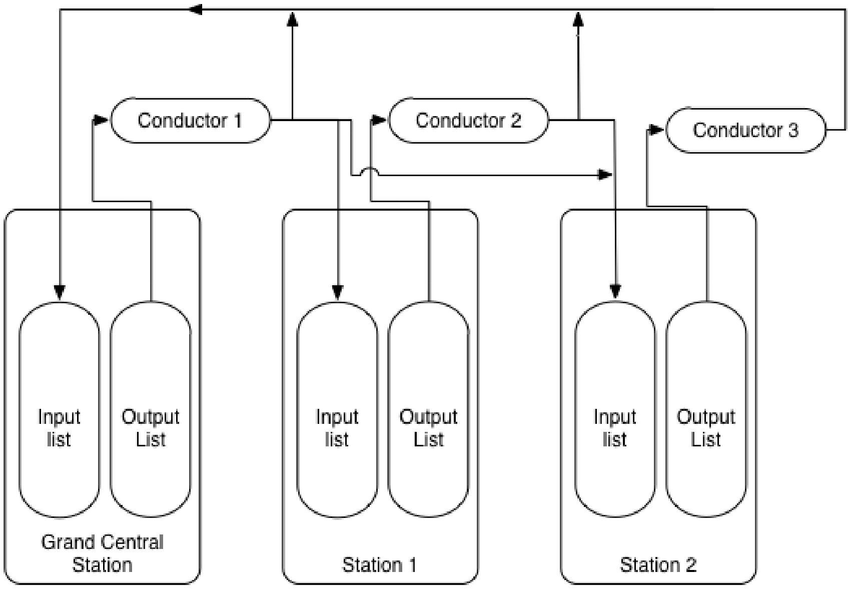


Figure 1.2: ET station details

The conductor also releases specially allocated memory associated with temporary events (more on temp events later). In this architecture, the flow control is totally inside the ET system process, which reduces the chance that a crashed user processes will stop the overall flow of events.

This design has made complete error recovery possible 99.9% of the time. The system and user processes each have a thread that generates a heartbeat by incrementing an integer in the shared memory. Using this heartbeat the ET system monitors each user process and each process monitors the ET system. If the ET system process dies, user processes automatically return from any function calls that are currently pending and can make a function call to find out if the system is still alive or can wait until it resurrects. Likewise, if a user program's heartbeat stops, the system kills the program and erases any trace of it from the system. All events tied up by the dead user process are returned to the system. Users can tell a station to take those recovered events and send them to either: 1) the station's input list, 2) the station's output list, 3) Grand Central station (essentially dumping them), or 4) to the output of the previous station if taken from a parallel station (more on this later).

The ET system tracks an event's owner - the process that currently has control over it. Keeping tabs on who has an event prevents the user from writing the same event twice or writing events into the system that it doesn’t own and thereby avoids serious problems.

Occasionally, a user will need an event to hold an amount of data larger than the maximum event size that was specified when the ET system was started. In such cases a file is memory mapped with all the requested memory. When all users are done with it, this temporary event will be disposed of - freeing up its memory. This is all transparent to the user.

Events can be either high or low priority. When written, high priority events 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.

The ET system consists of one process and does **not** depend on environmental variables affecting its behavior. In addition, there are no global or static variables in the code, making it reentrant. This allows one to use more than one ET system at the same time. Multiple systems peacefully coexist.

1.1.2 On remote host

The ET system can be transparently accessed by remote users by means of a TCP server built in to the ET system. By having a user specify how to connect to the desired ET system, it can operate as if on the same host as the ET system but with all communication occurring through a socket.

1.2 Java-based ET system

Although the ET system is also implemented in the Java programming language, it operates a little differently than the one written in C. Internally, it is **not** based on a memory-mapped file, but on an array of EtEvent objects. All communication with the Java system is done through TCP sockets and for that reason it functions identically to a C-based system with remote users.

1.3 Interoperability

Conveniently, whether ET system users are using the Java or C language, they are interoperable. C users can talk to Java-based ET systems using sockets and vice versa. Furthermore, when compiling an ET C library, the library libet_jni.so is also created. If this library is available, a Java user will access a local C-based ET system by means of its shared memory (using JNI) for the speed-sensitive calls to get new or used events, put events, or dump events. All other communication goes through sockets. This speeds operations up greatly for Java consumers.

Section

2

2. Creating an ET system

There are 2 types of ET systems that one can create. The first is written in C, is fast, and uses a memory mapped file to store all events and metadata. The second is written Java, is slower, and stores all events in an array inside the running Java Virtual Machine or JVM.

2.1 C-based ET system

2.1.1 System identification

The ET system is associated with a memory-mapped file. The file name is a unique identifier for the ET system but is cumbersome to use. To simplify the API we use an ET system id of type et_sys_id. A pointer to a variable is of this type used in calls to create a new system, open a connection to an existing system or close down the ET.

2.1.2 System creation

A new ET system is created by a call to the ET library function:

et_system_start(et_sys_id* id, et_sysconfig sconfig)

with arguments being a pointer to an ET system id and an ET system configuration.

If the ET system file specified in the configuration does not exist, an ET system is created. If the file already exists it is mapped into the process' memory and monitored to see if there is a live system heartbeat. If a heartbeat is detected another ET system is already attached to the file, a new one cannot be created using the same file and an error is returned. If there isn’t a heartbeat, the old file is deleted before creating a new ET system.

All ET systems are completely independent of each other, allowing the creation of as many as are necessary. However, the processes that created each ET system must remain running for as long as the ET system is in use. In other words, the software does NOT spawn or fork off an independent process to manage the ET (although it is possible for any user to implement such behavior).

To close the newly created ET system, use the function:

et_system_close(et_sys_id id)

Only the same process that created the ET system may call this function or an error will be returned.

2.1.3 System configuration

An ET system configuration is stored in a variable of the type et_sysconfig. Once this variable is declared, it must be initialized before further use. Thus users must call the function:

et_system_config_init(et_sysconfig *config).

After initialization, calls can be made to functions that set various properties of the specific configuration. Calls to these setting functions will fail unless the configuration is first initialized.

When the user is finished using a configuration variable, the user must call:

et_system_config_destroy(et_sysconfig config)

in order to properly release all memory used.

The configuration parameters that the user can set include things like the total number of events, the maximum size of each event, and the name of the system. For remote users, one can set how what IP addresses to use and what port numbers to use.

The functions used to set ET system parameters are listed below along with a short explanation for each:

- **et_system_config_setevents(et_sysconfig config, int val)** : sets the total number of events.
- **et_system_config_setsize(et_sysconfig config, int val)** : sets the maximum size in bytes for each events' data.
- **et_system_config_settemps(et_sysconfig config, int val)** : sets the maximum number of temporary events. These events are used when an event is required whose data size exceeds the limit set by the previous function. To accommodate large events, memory is specially allocated as needed. This cannot exceed the total number of events in the system.
- **et_system_config_setstations(et_sysconfig config, int val)** : sets the maximum number of stations.
- **et_system_config_setprocs(et_sysconfig config, int val)** : sets the maximum number of user processes which may open an ET system.
- **et_system_config_setattachments(et_sysconfig config, int val)** : sets the maximum number of attachments to stations.
- **et_system_config_setfile(et_sysconfig config, char *val)** : defines the name of an ET system. Each ET system is defined by a unique file name which is used to implement the memory mapped file basis of the ET system.
- **et_system_config_addmulticast(et_sysconfig config, char *val)** : adds a multicast address to a list, each address of which the ET system is listening on for UDP packets from users trying to find it. The address must be in dotted-decimal form.

- **et_system_config_removemulticast(et_sysconfig config, char *val)** : removes a multicast address from a list of addresses the ET system is listening on for UDP packets from users trying to find it. The address must be in dotted-decimal form.
- **et_system_config_setport(et_sysconfig config, int val)** : for remote users, set the broad/multicast UDP port number.
- **et_system_config_setserverport(et_sysconfig config, int val)** : for remote users making a “direct” connection, set the server’s TCP port number.
- **et_system_config_settcp(et_sysconfig config int rBufSize, int sBufSize, int noDelay)** : for remote users set the parameters of the TCP connection to the client: the sizes of the TCP send & receive buffers and the TCP no delay value.
- **et_system_config_setgroups(et_sysconfig config, int groups[], int size)** : To prevent contention between producers working in parallel, the ET system can create groups of empty events so that producer can request new events from different groups. This function sets the number and size of the groups. Groups are numbered starting at 1. The “groups” argument is an array in which the index is the group number minus 1 and the value is the number of events in the group. All non-zero values must be contiguous and start at index 0. The “size” argument is the size of the array. Events of a certain group number can be retrieved with et_events_new_group.

Similarly, functions used to GET these parameters are available and listed in the chapter describing all the ET library routines.

2.1.4 Starting an ET system

Although originally intended to be an example, the program **et_start** is used by everyone to start their ET systems. Thus the function calls mentioned above are, on a practical level, never used directly by anyone and only ever used in that single program. For the curious the source code is in <et top dir>/src/execsrc. For the rest of you here are all the command line options when running et_start:

```
usage: et_start  [-h] [-v] [-d] [-f <file>] [-n <events>] [-s <eventSize>]
                 [-g <groups>] [-stats <max # of stations>]
                 [-p <TCP server port>] [-u <UDP port>] [-a <multicast address>]
                 [-rb <buf size>] [-sb <buf size>] [-nd]

          -h     for help
          -v     for verbose output
          -d     deletes an existing file first
          -f.....sets memory-mapped file name
          -n     sets number of events
          -s     sets event size in bytes
          -g     sets number of groups to divide events into
          -p.....sets TCP server port #
          -u     sets UDP (broadcast &/or multicast) port #
          -a     sets multicast address
          -rb    TCP receive buffer size (bytes)
          -sb....TCP send    buffer size (bytes)
```

```
            -nd    use TCP_NODELAY option
            -stats max # of stations (default 200)
```

2.2 Java-based ET system

2.2.1 System file

Although a Java-based ET system does not use a memory-mapped file, it does create a file in which a few bits of information are placed. This enables local C users to realize it is a Java-based ET system they are trying to connect to.

2.2.2 System creation

A new ET system is created by instantiating a SystemCreate object:

SystemCreate etSystem = new SystemCreate (file, config);

with arguments being the name of the ET system file (String) and an ET system configuration object (SystemConfig). An exception is thrown if the file exists, it cannot be created, or an argument is bad.

To stop the newly created ET system, call:

etSystem.shutdown();

2.2.3 System configuration

An ET system configuration is given by a SystemConfig object which has methods to set the:

- number of events (**setNumEvents**)
- size of events in bytes (**setEventSize**)
- TCP port (**setServerPort**)
- UDP port (**setUdpPort**)
- multicast port (**setMulticastPort**)
- size and number of groups (**setGroups**)
- TCP send buffer size in bytes (**setTcpSendBufSize**)
- TCP receive buffer size in bytes (**setTcpRecvBufSize**)
- TCP no-delay (**setNoDelay**)
- level of debug output (**setDebug**)
- max number of attachments (**setAttachmentsMax**)
- max number of stations (**setStationsMax**)
- multicast addresses to listen on (**addMulticastAddr**)

- multicast addresses to remove (**removeMulticastAddr**)

.

Also provided are getter methods for all these parameters.

2.2.4 Starting an ET system

The method calls mentioned above are, on a practical level, never used directly by anyone and only ever used in a single, provided program. To run a Java-based ET system execute

java org.jlab.coda.et.apps.StartEt

With the –h option you get:

```
usage: java StartEt  [-h] [-v] [-d] [-f <file>] [-n <events>] [-s <eventSize>]
                 [-g <groups>] [-p <TCP server port>] [-u <UDP port>]
                 [-m <UDP multicast port] [-a <multicast address>]
                 [-rb <buf size>] [-sb <buf size>] [-nd]

          -h  for help
          -v  for verbose output
          -d  deletes an existing file first
          -f  sets memory-mapped file name
          -n  sets number of events
          -s  sets event size in bytes
          -g  sets number of groups to divide events into
          -p  sets TCP server port #
          -u  sets UDP broadcast port #
          -m  sets UDP multicast port #
          -a  sets multicast address
          -rb TCP receive buffer size (bytes)
          -sb TCP send    buffer size (bytes)
          -nd use TCP_NODELAY option
```

Section

3

3. Opening an ET system

The previous chapter discussed how to create an ET system. In this chapter we'll learn to open, close, and kill an existing system which is the first step towards interacting with it. The details of how to do this depends on the programming language used to interact with the ET system. Some of these details involve some knowledge of connecting to ET systems over the network. To learn more about using an ET system remotely, read chapter 9.

3.1 C library user

3.1.1 Opening

To connect to an existing ET system call:

et_open (et_sys_id* id, char *filename, et_openconfig config)

The parameters are a pointer variable of type et_sys_id, the shared memory filename of the existing ET system and a variable of type et_openconfig that determines the behavior of the function. Once a connection to an ET system is opened, a unique identifier is written into the id parameter. Users can open more than one system at a time, referring to each by their respective id.

3.1.2 Closing

When finished using an ET system, it can be removed from a process’s memory by calling:

et_close(et_sys_id id)

This unmaps the ET system memory from the process and makes it inaccessible. It also stops the heartbeat and system-heartbeat-monitor threads. In order to close, all attachments must be detached first. However, there is another function

et_forcedclose(et_sys_id id)

which will automatically do all the detaching first. Of course, the ET system continues to function for other processes as before.

3.1.3 Killing

There are occasions on which the user wants administrative control of ET systems. In addition to system creation, the killing and removal of a system can be very useful. In order to kill an ET system process and remove its file, the user must first open the system, then call:

et_kill(et_sys_id id)

Locally, for the user, this acts as an et_forcedclose.

3.1.4 Configuring the open

The configuration parameter for et_open must be initialized by a call to:

et_open_config_init (et_openconfig *config).

After initialization, calls can be made to functions to set various properties of the configuration. Calls to these setting functions will fail unless the configuration is first initialized:

- **et_open_config_setwait(et_openconfig config, int val)** : setting val to ET_OPEN_WAIT makes et_open block by waiting until the given ET system is fully functioning or a set time period has passed before returning. Setting val to ET_OPEN_NOWAIT makes et_open return immediately after determining whether the ET system is alive or not. If the system is remote, then broadcasting to find its location may take up to several seconds. The default is ET_OPEN_NOWAIT.
- **et_open_config_settimeout(et_openconfig config, struct timespec val)** : in ET_OPEN_WAIT mode, this function sets the maximum amount of time to wait for an ET system to appear. If the time is set to zero (the default), an infinite time is indicated. If broad/multicasting to find a remote ET system, it is possible to take up to several seconds to determine whether the system is alive or not -- possibly exceeding the time limit.
- **et_open_config_sethost(et_openconfig config, char *val)** : sets the name of the host (computer) on which the ET system resides. For opening a local system only, set val to ET_HOST_LOCAL (the default) or "localhost" (including quotes). For opening a system on another, unknown host, set it to ET_HOST_REMOTE. For an unknown host which may be local or remote, set it to ET_HOST_ANYWHERE. Otherwise set val to the name or dotted-decimal IP address of the desired host. (See next routine also).
- **et_open_config_setcast(et_openconfig config, int val)** : setting val to ET_BROADCAST (default) means using UDP broadcast IP packets to determine the location of ET systems so they can be opened. Setting val to ET_MULTICAST uses the newer UDP multicast IP packets to do the same. Setting val to ET_BROADANDMULTICAST does both. However setting val to ET_DIRECT makes a direct connection to the ET system and requires that et_open_config_sethost use the actual host's name, "localhost" or ET_HOST_LOCAL and not ET_HOST_REMOTE or ET_HOST_ANYWHERE. The TCP port number used in the direct connection is set by et_open_config_setserverport and defaults to ET_SERVER_PORT, defined in et.h as 11111.
- **et_open_config_setTTL(et_openconfig config, int val)** : when using multicasting, set the TTL value. This sets the number of routers to hop. The default is 32 which should allow multicast packets to pass through the local network routers.

- **et_open_config_setport(et_openconfig config, unsigned short val)** : sets the port number of the UDP broadcast communications. The default is ET_BROADCAST_PORT, defined in et.h as 11111.
- **et_open_config_setmultiport(et_openconfig config, unsigned short val)** : sets the port number of the UDP multicast communications. The default is ET_MULTICAST_PORT, defined in et.h as 11111. It will be necessary to use this routine when opening a Java-based ET system as broadcast and multicast ports must be different.
- **et_open_config_setserverport(et_openconfig config, unsigned short val)** : sets the port number of the TCP server thread of an ET system. The default is ET_SERVER_PORT, defined in et.h as 11111.
- **et_open_config_addbroadcast(et_openconfig config, char *val)** : adds an IP subnet broadcast address to a list of destinations used in broadcast discovery of the ET system to be opened. The val argument may also be set to ET_SUBNET_ALL which specifies all the local subnet broadcast addresses. Format is dotted-decimal only. Broadcasting is only used if et_open_config_setcast is set to ET_BROADCAST or ET_BROADANDMULTICAST.
- **et_open_config_removebroadcast(et_openconfig config, char *val)** : removes an IP subnet broadcast address from a list of destinations used in broadcast discovery of the ET system to be opened. If there is no such address on the list, it is ignored. The val argument may also be set to ET_SUBNET_ALL which removes all the subnet broadcast addresses (empties the list). Dotted-decimal only.
- **et_open_config_addmulticast(et_openconfig config, char *val)** : adds a multicast address to a list of destinations used in multicast discovery of the ET system to be opened. There can be at most ET_MAXADDRESSES (defined in et_private.h as 10) addresses on the list. Duplicate entries are not added to the list. Format is dotted-decimal. Multicasting is only used if et_open_config_setcast is set to ET_MULTICAST or ET_BROADANDMULTICAST.
- **et_open_config_removemulticast(et_openconfig config, char *val)** : removes a multicast address from a list of destinations used in multicast discovery of the ET system to be opened. If there is no such address on the list, it is ignored. Dotted-decimal.
- **et_open_config_setpolicy(et_openconfig config, int val)** : sets the return policy from an et_open call so that if a broad/multicast generates responses from multiple ET systems, different things can be done. Setting val to ET_POLICY_ERROR returns an error, ET_POLICY_FIRST opens the first responding system, and ET_POLICY_LOCAL opens the first responding local system if there is one, and if not, the first responding system.
- **et_open_config_setmode(et_openconfig config, int val)** : setting val to ET_HOST_AS_LOCAL (default) means users which are on the same machine as the ET system (local) will realize this and take advantage of it. However, setting val to ET_HOST_AS_REMOTE means users will be treated as if they were remote even if

they are local. All transactions will be through the ET system's server and not through shared memory.

- **et_open_config_setdebugdefault(et_openconfig config, int val)** : sets the default level of debugging output. Set val to: ET_DEBUG_NONE for no output, ET_DEBUG_SEVERE for output describing severe errors, ET_DEBUG_ERROR for output describing all errors, ET_DEBUG_WARN for output describing warnings and errors, and ET_DEBUG_INFO for output describing all information, warnings, and errors.
- **et_open_config_setinterface(et_openconfig config, int val)** : sets the network interface to use in order to communicate with the ET system (if acting as a remote consumer). Set val to the dotted decimal form of the IP address of the interface desired.
- **et_open_config_settcp(et_openconfig config, int rBufSize, int sBufSize, int noDelay)** : regarding the TCP connection to the ET system, this sets the TCP receive buffer size in bytes, the TCP send buffer size in bytes, and sets the TCP “no delay” on or off. If either buffer size is zero, then system default values are used. A value of 0 turns off the “no delay” and any other value turns it on.

More information on using remote ET systems can be found in the chapter entitled Remote ET. All of the above "set" functions have their counterpart "get" functions as well.

After calling et_open the user must call:

et_open_config_destroy (et_openconfig config)

to release the memory used by the et_openconfig structure.

3.2 Java jar user

As with all java programming, a look at the javadoc documentation will give details absent in this general presentation.

3.2.1 Opening

To connect to an existing ET system, create an EtSystem object and then call open():

EtSystem sys = new EtSystem(config);

sys.open();

The argument is a configuration of class EtSystemOpenConfig that determines the behavior of the method.

3.2.2 Closing

When finished using an ET system, it can be removed from a JVM by calling:

sys.close();

This closes all streams and sockets to the ET server and does a close on any JNI et_open that may have been done. It is also equivalent to the et_forcedclose() of the C library.

3.2.3 Killing

There are occasions on which the user wants administrative control of ET systems. In addition to system creation, the killing and removal of a system can be very useful. In order to kill an ET system process and remove its file, the user must first open the system, then call:

sys.kill();

Locally, for the user, this acts as a close().

3.2.4 Configuring the open

The configuration for opening is created by instantiating an EtSystemOpenConfig object by a call to:

EtSystemOpenConfig config = new EtSystemOpenConfig();

This class has several different constructors for convenience – some for broadcasting, some for multicasting, one for a direct connection, and one for setting all parameters. Setter methods also exist:

- **seEtName(String name)** : sets ET system (file) name
- **setHost(String val)** : sets the name of the host (computer) on which the ET system resides For opening a local system only, set val to EtConstants.hostLocal (the default) or "localhost" (including quotes). For opening a system on another, unknown host, set it to EtConstants.hostRemote. For an unknown host which may be local or remote, set it to EtConstants.hostAnywhere. Otherwise set val to the name or dotted-decimal IP address of the desired host. (See next method also).
- **setNetworkContactMethod(int val)** : setting val to EtConstants.broadcast (default) means using UDP broadcast IP packets to determine the location of ET systems so they can be opened. Setting it to EtConstants.multicast uses UDP multicast IP packets to do the same. Setting it to EtConstants.broadAndMulticast does both. However setting it to EtConstants.direct makes a direct connection to the ET system and requires that setHost() use the actual host's name, "localhost" or EtConstants.host and not EtConstants.hostRemote or EtConstants.hostAnywhere. The TCP port number used in the direct connection is set by setTcpPort() and defaults to EtConstants.serverPort, defined as 11111.
- **setWait (long val)** : sets max time to wait for ET system to appear in milliseconds.
- **setConnectRemotely(boolean val)** : option to use JNI to access a local C-based ET system if false.
- **addBroadcastAddr(String val)** : adds a single IP subnet broadcast address to a list of destinations used in broadcast discovery of the ET system to be opened. There are 2 constructors of the EtSystemOpenConfig object that specify all local subnets and one

form of the constructor in which a collection of desired addresses can be passed in. Duplicate entries are not added. Format is dotted-decimal only. Broadcasting is only used if setNetworkContactMethod() is set to EtConstants.broadcast or EtConstants.broadAndMulticast.

- **setBroadcastAddrs(Collection<String> addrs)** : sets the collection of IP subnet broadcast address destinations used in broadcast discovery of the ET system to be opened. Duplicates are removed.
- **addMulticastAddr(String addr)** : adds a multicast address to a list of destinations used in multicast discovery of the ET system to be opened. Duplicate entries are not added. Format is dotted-decimal. Multicasting is only used if etNetworkContactMethod() is set to EtConstants.multicast or EtConstants.broadAndMulticast.
- **setMulticastAddrs(Collection<String> addrs)** : sets the collection of multicast address destinations used in multicast discovery of the ET system to be opened. Duplicates are removed.
- **removeMulticastAddr(String addr)** : removes a single multicast address from the Collection of multicast addresses to be used in ET discovery.
- **setTcpPort(int val)** : sets TCP port of ET system. The default is EtConstants.serverPort, 11111.
- **setUcpPort(int val)** : sets port number to broadcast to. The default is EtConstants.broadcastPort, 11111. When opening a Java-based ET system, broadcast and multicast ports must be different.
- **setMulticastPort(int val)** : sets port number to multicast to. The default is EtConstants.multicastPort, 11112. When opening a Java-based ET system, broadcast and multicast ports must be different.
- **setTTL(int val)** : set time-to-live value when multicasting (0 – 253, defaults to 32).
- **setResponsePolicy(int val)** : sets the return policy from a sys.open() call so that if a broad/multicast generates responses from multiple ET systems, different things can be done. Setting val to EtConstants.policyError returns an error, EtConstants.policyFirst opens the first responding system, and EtConstants.policyLocal opens the first responding local system if there is one, and if not, the first responding system.
- **setNetworkInterface(String val)** : sets the network interface to use in order to communicate with the ET system. Set val to the dotted decimal form of the IP address of the interface desired.
- **setTcpSendBufSize(int val)** : sets TCP send buffer size in bytes. If val is zero, then system default value is used.
- **setTcpRecvBufSize (int val)** : sets TCP receive buffer size in bytes. If val is zero, then system default value is used.
- **setNoDelay (boolean val)** : sets TCP no-delay on if true, else off.

Also provided are getter methods for these parameters.

Section
4

4. Using stations

The previous chapter discussed how to open an ET system, and in this chapter we'll learn to use an existing system by showing how to define, create & remove stations and how to attach to & detach from stations. First let’s look at stations in a little more detail.

A station consists of 2 lists: an input list (or queue) of events and an output list of events. The input list is where events come from when a user does a “get”. The output list is where events go when users do a “put” when finished with them. Each list has enough room to store every event in the ET system. In addition to these 2 lists, there are a number of properties each station has which determine which events to process and how it is done.

4.1 Attachments, active and inactive stations

A user must “attach” to a station in order to get and put events from it. An attachment allows the ET system to control access to its events and is used to mark who is currently in possession of each event. The number of attachments to a station can be set to a user-defined limit.

In order to avoid having the ET system block the flow of events, each station is marked by the system as either active or inactive. A station is active if there is at least one attachment to it, else it is inactive. Events are never placed by the system in an inactive station since there would be no user to get, put and send them on their way downstream. Inactive stations are bypassed by the event flow. Once a user is done getting & putting events from a station, he should detach, particularly from a blocking station (see below). Otherwise, the flow of events could be completely stopped.

4.2 Event recovery

If the last attachment to a station is detached without the user putting all of its events back into the ET system, it will find all events owned by that attachment and recover them by placing them back into another station. The exact placement of these recovered events can be set for each station. They can be placed in its input list, output list, in Grand Central’s input list. And in the case of parallel stations (see below), they may be placed in the previous station’s output list for redistribution among a single group of parallel stations.

4.3 Blocking, non-blocking, and queue size

Perhaps the most basic property is whether a station examines all events coming down the track or only a subset of them for possible entry into the input list. A station which looks at every event is called “blocking” and one that does not is called “non-blocking”. A non-blocking station has a user configurable input list or queue size. It examines all events coming down the track and places them in the input queue (if it makes it through the filtering and prescaling process) until its queue is full, after which all other events flowing through the system will bypass the station and go to the next one in line. Once the queue is no longer full, events will once again be placed into it. The events that bypass the station are never examined by it at all. Non-blocking stations will “mess up” the original order of events. A note of caution: setting the queue size large enough to contain all events (its maximum value) will have the effect of making a non-blocking station behave like a blocking station.

4.4 Filtering / distributing events

There are 3 main ways of filtering events to determine which are accepted into a station’s input list. In the first and default filtering method, each event that is examined is always accepted.

In the second method, a built-in function can be activated to act as a filter. To understand what the function does, one must realize that each event has an array of integers associated with it – metadata which can be set by the user. Likewise, each station has a corresponding array of integers of the same size associated with it – again, metadata which can be set by the user. The function compares each element of the 2 arrays together and ORs the results. The first elements are checked for equality, the second elements with a bitwise AND, the thirds for equality, the fourths with a bitwise AND etc., etc. If the result is 1 or true, then the event is accepted into the input list.

In the third method, a user can supply a function name and a library name from which it is loaded in the case of C, or a class name can be supplied in the case of Java. In either case, the filter function/method is dynamically loaded.

There are also 2 additional methods of filtering or distributing events in the case of parallel stations (see below for more on parallel stations). One is to do a round-robin distribution among parallel stations in a single group. The other is to distribute events in such a way that each station’s input list contains roughly the same number of events (load-distribution).

4.5 Prescaling

For all incoming events that make it past the filter, there is the capability to prescale them, that is, to only accept every Nth normally accepted event. The prescale value defaults to 1 which means every event is accepted.

4.6 Serial & parallel

Events normally flow serially through stations, meaning that when a user at one station puts events back, they flow serially downstream to the next station in line, through each and every one in turn, until they eventually reach Grand Central where they are reused. However, often it is useful to distribute events in such a way that several tasks can be performed in parallel. To facilitate this, the ET system allows “parallel” stations to be grouped together so that they act as a unit. Within a single group, one station is declared to be the head station with the others coming after it in a given order. The station previous to the group will distribute events among the group so that each parallel station accepts different events from all the others.

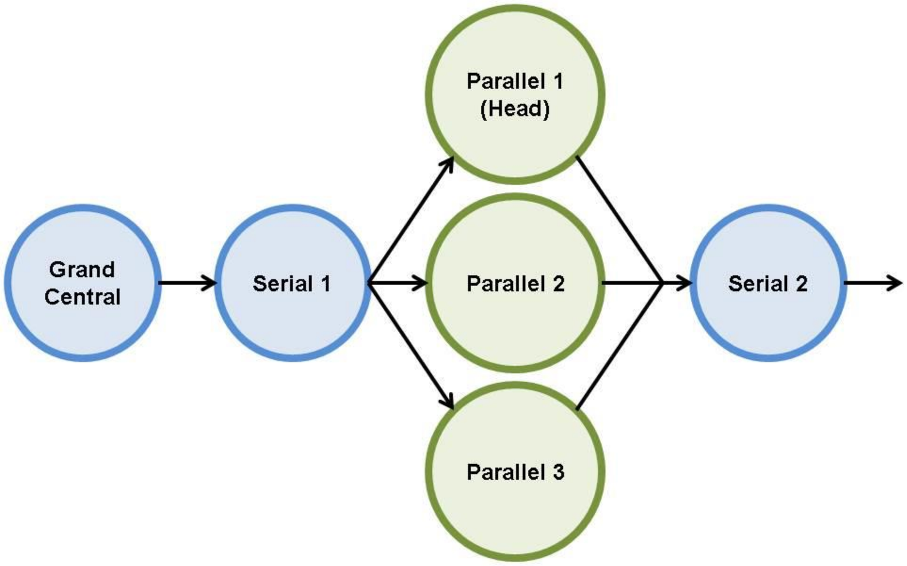


Figure 4.1

In figure 4.1, Parallel 1, Parallel 2, and Parallel 3 are parallel stations in a single group with Parallel 1 being the head. In addition to normal distribution methods, Serial 1’s conductor can distribute its events to the parallel stations with two other algorithms: 1) a round robin algorithm in which each station is given an event in turn and, 2) an equal-queue algorithm in which the conductor tries to keep the input queues of Parallel 1, Parallel 2, & Parallel 3 equally full for load balancing. When events are put back into a parallel station, its conductor will place them into the next station in the main list, in this case Serial 2.

4.7 Placement

There is one station created by the ET system itself – Grand Central. It is always the very first station and cannot be removed. It serves as a repository of all unused events. Hence most producers attach to Grand Central when creating data to be placed into events. All other stations are created by users and may be placed anywhere else in the chain.

4.8 C library users

4.8.1 Creation and removal

A station is created by calling:

> **et_station_create(et_sys_id id, et_stat_id *stat_id, char *stat_name, et_statconfig sconfig)**

with the arguments: 1) ET system id from et_create() or et_open(), 2) pointer to the station ID of the new station, 3) unique name for the station, 4) station configuration (described later). This creates a new station at the end ET's list of stations. Sometimes the user needs to create a new station between a pair of existing stations or create a station in a group of parallel stations. This is achieved using:

> **et_station_create_at(et_sys_id id, et_stat_id *stat_id, const char *stat_name, et_statconfig sconfig, int position, int parallelposition.)**

This function has the same parameters as et_station_create plus two additional parameters:

- position - an integer representing a specific place in ET's main list of stations. The position is defined with Grand Central as 0. The position of an existing station can be found using et_station_get_position (see section **Error! Reference source not found.**). reating a station between two stations, for example stations 1 and 2, increments the position of the higher numbered station by one, so the new station has an position of 2 and the station previously at position 2 is now at 3. The value ET_END places the station at the end of the list.
- parallelposition – an integer representing a specific place in a list of stations acting in parallel. In this case the head of the group is at position 0 (ET_HEAD) and the position parameter is the position of the group.

Since creating stations at a position other than the end alters the indexes of existing stations, it is recommended that et_station_get_position should always be used to determine the position of existing stations and that station creation should be limited to a single thread.

Both functions return ET_ERROR_EXISTS if a station by that name exists already and has a different configuration. If one exists with the same configuration, an id to that station is returned. The functions return ET_ERROR_TOOMANY if the user is the second user to try to attach to a station designated for one user only or ET_ERROR for other unrecoverable errors such as bad positions. If the user is a remote consumer, the error ET_ERROR_REMOTE indicates a bad argument or not being able to allocate memory, and ET_ERROR_READ & ET_ERROR_WRITE indicate problems with the network communication.

A station is removed by calling:

> **et_station_remove(et_sys_id id, et_stat_id stat_id)**

4.8.2 Configuration

The station configuration is defined using a variable of type et_statconfig and must be initialized using the function:

et_station_config_init(et_statconfig* sconfig)

After initialization, calls can be made to functions that set various properties of the specific configuration. Calls to these setting functions will fail unless the configuration has been initialized.

- **et_station_config_setblock(et_statconfig sconfig, int val)** : setting val to ET_STATION_BLOCKING makes the station block by looking at all events in the system, while setting it to ET_STATION_NONBLOCKING allows the station to fill up a queue of events and when that is full, events flow to the next station downstream. The default is blocking.
- **et_station_config_setflow(et_statconfig sconfig, int val)** setting val to ET_STATION_SERIAL makes events flow through the station normally, while setting it to ET_STATION_PARALLEL can make the station part of a group of stations through which events flow in parallel downstream. The default is serial.
- **et_station_config_setcue(et_statconfig sconfig, int val)** : when in non-blocking mode, this sets the maximum number of events that are to be in the station's input list ready for reading. The default is 10.
- **et_station_config_setprescale(et_statconfig sconfig, int val)** : when in blocking mode, every Nth event of interest is sent to the user by setting the val to N. The default is 1.
- **et_station_config_setuser(et_statconfig sconfig, int val)** : setting val to ET_STATION_USER_SINGLE allows only one user process to attach to this station, while setting it to ET_STATION_USER_MULTI allows multiple users to attach. Setting it to a positive integer allows only that number of attachments to the station. The default is multiuser.
- **et_station_config_setrestore(et_statconfig sconfig, int val)** : when a user detaches from a station either deliberately or because it died, the events it read but did not write are recovered and sent to the station's output list if val is set to ET_STATION_RESTORE_OUT. Similarly, they can be sent to the input list with ET_STATION_RESTORE_IN or back to Grand Central station with ET_STATION_RESTORE_GC. Finally, if val is ET_STATION_RESTORE_REDIST and if the station has parallel flow, the events can be sent to the output list of the previous station in which case these events will be redistributed to the group of parallel stations. This final option is useful, for example, in a processing farm of parallel stations. If a farm node (and its accompanying attachments) disappears, all the events that were written to it can be automatically redistributed to other farm nodes. The default is restoration to the output list. There are no guarantees that the recovered events will be in their original order.

- **et_station_config_setselect(et_statconfig sconfig, int val)** : determines which events are placed on the input list. For selection of all events and no filtering set val to ET_STATION_SELECT_ALL. Set it to ET_STATION_SELECT_USER for selection using a user-defined routine loaded through a shared library in a C-based ET system or a user-defined class dynamically loaded with the class loader in a Java-based ET system. The ET_STATION_SELECT_MATCH option takes an event's array of control integers and does a comparison with the station's selection integers or words. The first pair is checked for equality, the next a bitwise AND, then back to equality, then a bitwise AND, etc. The results of all logical comparisons are ORed together. An event is selected if result = 1. This may seem strange but is a holdover from the DD system, which was a precursor to the ET system and can occasionally be useful. For parallel stations there are 2 more possibilities. ET_STATION_SELECT_RROBIN distributes events to a group of parallel stations with a round robin algorithm. ET_STATION_SELECT_EQUALCUE distributes events so that the input queues of each parallel station will be kept the equally full (as much as possible). See below for more details. The default mode is ET_STATION_SELECT_ALL.
- **et_station_config_setselectwords(et_statconfig sconfig, int *val)** : the argument is an array of integers used when the station select mode is set to ET_STATION_SELECT_MATCH or ET_STATION_SELECT_USER depending on what algorithm a user-defined, event selection routine uses. These integers are compared to the incoming events’ control word array. The default is to set all integers to a value of "-1". In ET_STATION_SELECT_MATCH mode each element of the station's selection array is checked to see if the is equal to -1. If it is, then the corresponding element of the event's control array is ignored. Thus, if all elements of a station's selection array are set to -1, the event will NOT be selected. If the first element of the station's selection array is not -1 but is equal to the first element of the event's control array, then the event is selected. Likewise if the second element of the selection array is not -1 and if the bitwise AND (&) of the select and control second elements is true, then the event is selected. This pattern is repeated with the even elements 0,2,4, 6, ... compared for equality and the odd elements 1, 3, 5, ... compared for bitwise AND. If any of the comparisons are true, then the event is selected.
- **et_station_config_setlib(et_statconfig sconfig, char *val)** : for a select mode of ET_STATION_SELECT_USER, val is the name of the shared library containing the function to be used for selecting events. Make sure it’s in the load library path.
- **et_station_config_setfunction(et_statconfig sconfig, char *val)** : for a select mode of ET_STATION_SELECT_USER, val is the name of the function to be used for selecting events.
- **et_station_config_setclass(et_statconfig sconfig, char *val)** : when defining a station on a Java-based ET system with a select mode of ET_STATION_SELECT_USER, val is the name of the class containing the method to be used for selecting events. This class must implement the EtEventSelectable interface, and make sure it’s in the classpath.

Similar functions to those mentioned above are available to GET the values associated with a station configuration.

After use, the memory allocated to the configuration must be freed by calling:

et_station_config_destroy(et_statconfig sconfig)

4.8.3 Configuration examples

Since one of the more difficult tasks facing the first time user is how to properly configure a station, let's look at two examples first:

```
/* declarations */
et_stat_config sconfig;
/* set values */
et_station_config_init(&sconfig);
et_station_config_setselect(sconfig, ET_STATION_SELECT_ALL);
et_station_config_setblock(sconfig, ET_STATION_NONBLOCKING);
et_station_config_setcue(sconfig, 20);
et_station_config_setuser(sconfig, ET_STATION_USER_SINGLE);
et_station_config_setrestore(sconfig, ET_STATION_RESTORE_GC);
```

This station accepts all events ignoring the selection integers. It is non-blocking with a queue size of 20. Once the input list fills to 20 events, all other events will bypass this station. ET_STATION_RESTORE_GC specifies that if the user process should die, the events that it owns will be placed back in Grand Central station.

A more complicated example is shown below:

```
/* declarations */
int selections[] = {17, 22, -1, -1};
et_stat_config sconfig;

et_station_config_init(&sconfig);
et_station_config_setuser(sconfig, ET_STATION_USER_MULTI); // multi user
et_station_config_setblock(sconfig, ET_STATION_BLOCKING);  // process all
et_station_config_setprescale(sconfig, 5);                 // prescale by 5
/* user-specified select function */
et_station_config_setselect(sconfig, ET_STATION_SELECT_USER);
/* Specify selection function */
if (et_station_config_setlib(sconfig, "/stuff/libet_user.so") == ET_ERROR) {
 printf(" cannot set library\n");
}
if (et_station_config_setfunction(sconfig, "et_my_function") == ET_ERROR) {
 printf("cannot set function\n");
}
et_station_config_setselectwords(sconfig, selections);   // ints for selection
et_station_config_setrestore(sconfig, ET_STATION_RESTORE_IN); // error recovery
```

This station allows multiple users to attach. The station is set to accept only every fifth event that passes its selection filter. The user supplies a function in a shared library to determine which events are to be selected. The selection ints used by the selection function are set. If this user process should ever die, the restore mode specifies that any events that it currently owns will be placed in the station's input list so that one of the other processes attached to the station can process them.

4.8.4 Attaching to and detaching from stations

Until a user attaches to a station, the station is placed in an idle or inactive mode, meaning that it is not participating in the flow of events and is bypassed. Once a user attaches to a station, it becomes active and begins to receive events. This logic ensures that events do not get stuck in stations with no attachments. Attach to a station by calling:

et_station_attach(et_sys_id id, et_stat_id stat_id, et_att_id *att).

This routine returns an attachment ID in parameter att, which identifies the attachment. In this manner, a single user can attach to several different stations. For example, a user could have multiple threads with each attached to the same station on a different attachment. The attachment id is also a way of specifying the ownership of an event - which is important in setting limits on how and where events can flow.

To detach from a station call:

et_station_detach(et_sys_id id, et_att_id att)

If this is the last attachment to be detached, any events remaining in the station's input list are automatically passed to the output list. Also a search is made for any events that were read but not written by that attachment with any that are found placed back into the ET system. These are recovered and placed according to the station's configuration property set by the function et_station_config_setrestore().

4.8.5 Changing a station's behavior on the fly

Some of the parameters that define a station's behavior as well as its position in the linked list of stations may be modified while an ET system is operating. The only things that cannot be done are to load new user-defined event selection functions or to change the select mode of the station.

The functions used to SET station parameters are listed below along with an explanation for each:

- **et_station_setposition(et_sys_id id, et_stat_id stat_id, int position, int pposition)** : setting position to a positive integer places the station at that position in the linked list of active and idle stations. The position of 0 is prohibited as the first position is reserved for Grand Central station. Set pposition to the desired place among a single group of parallel stations if it is a parallel station, but it may not be 0 (head spot) if a head parallel station already exists.
- **et_station_setblock(et_sys_id id, et_stat_id stat_id, int val)** : setting val to ET_STATION_BLOCKING makes the station block by looking at all events in the system, while setting it to ET_STATION_NONBLOCKING allows the station to fill up its input queue of events and when that is full, events flow to the next station downstream.

- **et_station_setcue(et_sys_id id, et_stat_id stat_id, int val)** : when in nonblocking mode, this sets the maximum number of events that are to be in the station's input list ready for reading (in so far as it is possible).
- **et_station_setprescale(et_sys_id id, et_stat_id stat_id, int val)** : when in blocking mode, every Nth event of interest is sent to the user by setting the val to N.
- **et_station_setuser(et_sys_id id, et_stat_id stat_id, int val)** : setting val to ET_STATION_USER_SINGLE allows only one user process to attach to this station, while setting it to ET_STATION_USER_MULTI allows multiple users to attach. Setting it to a positive integer allows only that number of attachments to the station.
- **et_station_setrestore(et_sys_id id, et_stat_id stat_id, int val)** : when a process dies or detaches from a station, the events it read but did not write are recovered and sent to a station's output list if val is set to ET_STATION_RESTORE_OUT. Similarly, it can be sent to the input list with ET_STATION_RESTORE_IN or back to Grand Central station with ET_STATION_RESTORE_GC.
- **et_station_setselectwords(et_sys_id id, et_stat_id stat_id, int *val)** : the argument is an array of integers used when the station select mode is set to ET_STATION_SELECT_MATCH or possibly ET_STATION_SELECT_USER (depending on what algorithm a user-defined, event selection routine uses)..

Similar functions to those mentioned above are available to GET the values associated with a station's configuration. Note that none of the above functions are allowed to modify Grand Central station.

4.9 Java jar users

4.9.1 Creation and removal

A station is created by calling:

EtStation station = sys.createStation(config, “myStation”, EtConstants.end);

where sys is an EtSystem object and the arguments are the: 1) a station configuration (EtStationConfig) object, 2) station name, and 3) station position relative to Grand Central. This creates a new station at the end ET’s list of stations. The method createStation() also has an overloaded version with an additional argument allowing a parallel position to be set. The value of EtConstants.newHead will place the station at the head of a group of parallels.

Since creating stations at a position other than the end alters the indexes of existing stations, it is recommended that sys.getStationPosition(station) should always be used to determine the position of existing stations and that station creation should be limited to a single thread.

This method throws an exception for many reasons including if a station by that name exists already but with a different configuration. If a station exists and with the same configuration, it returns with a valid object.

A station is removed by calling:

sys.removeStation (station);

4.9.2 Configuration

The station configuration is created:

EtStationConfig config = new EtStationConfig();

Methods can be called to set various properties of the specific configuration:

- **setBlockMode(int val)** : setting val to EtConstants.stationBlocking makes the station block by looking at all events in the system, while setting it to EtConstants.stationNonBlocking allows the station to fill up a cue of events and when that is full, events flow to the next station downstream. The default is blocking.
- **setFlowMode(int val)** setting val to EtConstants.stationSerial makes events flow through the station normally, while setting it to EtConstants.stationParallel can make the station part of a group of stations through which events flow in parallel downstream. The default is serial.
- **setCue(int val)** : when in non-blocking mode, this sets the maximum number of events that are to be in the station's input list ready for reading. The default is 10.
- **setPrescale(int val)** : when in blocking mode, every Nth event of interest is sent to the user by setting the val to N. The default is 1.
- **setUserMode(int val)** : setting val to EtConstants.stationUserSingle allows only one user process to attach to this station, while setting it to EtConstants.stationUserMulti allows multiple users to attach. Setting it to a positive integer allows only that number of attachments to the station. The default is multi user.
- **setRestoreMode(int val)** : when a user detaches from a station either deliberately or because it died, the events it read but did not write are recovered and sent to the station's output list if val is set to EtConstants.stationRestoreOut. Similarly, they can be sent to the input list with EtConstants.stationRestoreIn or back to Grand Central station with EtConstants.stationRestoreGC. Finally, if val is EtConstants.stationRestoreRedist and if the station has parallel flow, the events can be sent to the output list of the previous station in which case these events will be redistributed to the group of parallel stations. This final option is useful, for example, in a processing farm of parallel stations. If a farm node (and its accompanying attachments) disappears, all the events that were written to it can be automatically redistributed to other farm nodes. The default is restoration to the output list. There are no guarantees that the recovered events will be in their original order.
- **setSelectMode(int val)** : determines which events are placed on the input list. For selection of all events and no filtering set val to EtConstants.stationSelectAll. Set it to EtConstants.stationSelectUser for selection using a user-defined routine loaded through a shared library in a C-based ET system or a user-defined class dynamically loaded with the class loader in a Java-based ET system. The EtConstants.stationSelectMatch option takes an event's array of control integers and does a comparison with the station's selection integers or words. The first pair is checked for equality, the next a bitwise AND, then back to equality, then a bitwise AND, etc. The results of all logical

comparisons are ORed together. An event is selected if result = true. This may seem strange but is a holdover from the DD system, which was a precursor to the ET system and can occasionally be useful. For parallel stations there are 2 more possibilities. EtConstants.stationSelectRRobin distributes events to a group of parallel stations with a round robin algorithm. EtConstants.stationSelectEqualCue distributes events so that the input queues of each parallel station will be kept the equally full (as much as possible). See below for more details. The default mode is EtConstants.stationSelectAll.

- **setSelect(int[] val)** : the argument is an array of integers used when the station select mode is set to EtConstants.stationSelectMatch or The EtConstants.stationSelectUser depending on what algorithm a user-defined, event selection routine uses. These integers are compared to the incoming events' control word array. The default is to set all integers to a value of "-1". In the EtConstants.stationSelectMatch mode, each element of the station's selection array is checked to see if the is equal to -1. If it is, then the corresponding element of the event's control array is ignored. Thus, if all elements of a station's selection array are set to -1, the event will NOT be selected. If the first element of the station's selection array is not -1 but is equal to the first element of the event's control array, then the event is selected. Likewise if the second element of the selection array is not -1 and if the bitwise AND (&) of the select and control second elements is true, then the event is selected. This pattern is repeated with the even elements 0,2,4, 6, ... compared for equality and the odd elements 1, 3, 5, ... compared for bitwise AND. If any of the comparisons are true, then the event is selected.
- **setSelectLibrary(String val)** : for a select mode of EtConstants.stationSelectUser, val is the name of the shared library containing the function to be used for selecting events. Make sure it's in the load library path.
- **setSelectFunction(String val)** : for a select mode of EtConstants.stationSelectUser, val is the name of the function to be used for selecting events.
- **setSelectClass(String val)** : when defining a station on a Java-based ET system with a select mode of EtConstants.stationSelectUser, val is the name of the class containing the method to be used for selecting events. This class must implement the EtEventSelectable interface, and make sure it's in the classpath.

Similar functions to those mentioned above are available to GET the values associated with a station configuration.

4.9.3 Configuration examples

Since one of the more difficult tasks facing the first time user is how to properly configure a station, let's look at two examples first:

```
// Instantiate config object
EtStationConfig config = new EtStationConfig();

// Set values
config.setSelectMode(EtConstants.stationSelectAll);
config.setBlockMode(EtConstants.stationNonBlocking);
config.setCue(20);
config_setUserMode(EtConstants.stationUserSingle);
```

```
config.setRestoreMode(EtConstants.stationRestoreGC);
```

This station accepts all events ignoring the selection integers. It is non-blocking with a queue size of 20. Once the input list fills to 20 events, all other events will bypass this station. EtConstants.stationRestoreGC specifies that if the user process should die, the events that it owns will be placed back in Grand Central station.

A more complicated example is shown below:

```
// Define selection array
int[] selections = {17, 22, -1, -1};

// Instantiate config object
EtStationConfig config = new EtStationConfig();

// Set values
config.setBlockMode(EtConstants.stationBlocking);
config.setPrescale(5);
config_setUserMode(EtConstants.stationUserMulti);
config.setRestoreMode(EtConstants.stationRestoreIn);

// User-specified class
config.setSelectMode(EtConstants.stationSelectUser);
config.setSelectClass(“org.jlab.coda.et.EtStationSelection”);
config.setSelect(selections);
```

This station allows multiple users to attach. The station is set to accept every fifth event that passes its selection filter. The user supplies a class in the classpath to determine which events are to be selected. The selection ints used by the selection method are set. If the user process should ever die, the restore mode specifies that any events that it currently owns will be placed in the station's input list so that one of the other processes attached to the station can process them.

4.9.4 Attaching to and detaching from stations

Until a user attaches to a station, the station is placed in an idle mode, meaning that it is not participating in the flow of events and is bypassed. Once a user attaches to a station, it becomes active and begins to receive events. This logic ensures that events do not get stuck in stations with no attachments. Attach to a station by calling:

EtAttachment att = sys.attach(station);

In this manner, a single user can attach to several different stations. For example, a user could have multiple threads with each attached to the same station on a different attachment. Each attachment marks its ownership of an event - which is important in setting limits on how and where events can flow.

To detach from a station call:

sys.detach(att);

If this is the last attachment to be detached, any events remaining in the station's input list are automatically passed to the output list. Also a search is made for any events that were read but not written by that attachment with any that are found placed back into the ET system. These are recovered and placed according to the station's configuration property set by the function config.setRestoreMode().

4.9.5 Changing a station's behavior on the fly

The only allowable change to a station's behavior while the ET system in operating is to its position in the linked list of stations with:

sys. setStationPosition(station, int position, int pposition)

The position of 0 is prohibited as the first position is reserved for Grand Central station. Set pposition to the desired place among a single group of parallel stations if it is a parallel station, but it may not be EtConstants.newHead if a head parallel station already exists.

One can use:

sys.getStationPosition(station), and

sys.getStationParallelPositin(station)

to get the values associated with a station. Note that none of the above methods are allowed to modify Grand Central station

.

Section
5

5. Using events

After opening an ET system, creating a station, and attaching to it, users are ready to start reading and writing events. Although an event is basically a data buffer, each one has some additional properties that can be set by the user both to help identify its contents and to direct its flow through the stations. Users who initially write data into an empty event are called producers. Those who read and or modify already existing events are called consumers.

Producers get “new” events from a pool of unused events in Grand Central station’s input list (this is all transparent to the user). Once filled with data, the event is “put” back into the ET system where it makes its way from station to station. Most producers attach to Grand Central station since that way all other stations are downstream and therefore all consumers will see the newly produced event. However, producers may attach to any station.

Consumers, on the other hand, attach to the station of choice and do a “get” to access an event recently created by a producer. The consumer is free to do what it wishes with the data. When finished, a “put” is required to send the event on its way to other stations down the track. There is also the option for a consumer to “dump” an event. This will place the event directly into Grand Central’s input list which bypasses all other downstream users and makes it available again for producers.

5.1 Data length

Most obviously, the number of bytes in an event which contain valid data must be set and can only be set by the user.

5.2 Data endianness

The ET system keeps track of the endianness of the host on which the event was created. Any subsequent event reader can then be warned of the need to swap endianness. The endian value can also be set directly by the user at any time.

5.3 Data status

When a consumer detaches from a station without having put or dumped all the events it got, the ET system locates and recovers them, placing them back into a station. Events that have been recovered in this manner are label as having possibly corrupted data as a warning to users.

5.4 Control integer array

Each event has an array of 6 “control” integers whose values can be set by the user. This array can be used by stations when filtering events or for any other conceivable purpose.

5.5 Groups

Each ET system may divide events into groups. Group numbers are integers that start at 1 and sequentially increase. The purpose of this is to solve contention between multiple producers on the same ET system. If multiple producers simultaneously exist, the fastest one will grab most of the available free events, sometimes to the complete exclusion of a slow producer. To avoid this, each producer can call a routine/method that will only acquire new events from a specified group. In such a manner, piggish producers can be forced to play fair with others.

5.6 Event priority

An event can be either high or low priority with low being the default. A high priority event is placed above those of low priority when the user puts it back into the system. In practice this is rarely used since it will change the order of events.

5.7 Use of arrays

When getting new or old events, putting them, or dumping them, it is possible to do so in arrays of events. This can greatly increase the speed and efficiency of an ET system often by a factor of 10 or more. Many calls to read or write small chunks of data are by its nature slower than fewer reads/writes with larger amounts of data.

5.8 C library users

5.8.1 Getting a new (unused) event

To get a single event call:

> **et_event_new(et_sys_id id, et_att_id att, et_event **pe, int wait, struct timespec *time, int size)**

The first two arguments are the ET and station IDs. The third is a pointer to a pointer in which a pointer to the event will be written. The fourth argument, wait, is a flag that is set by using predefined macros:

- **ET_SLEEP**, the call will block until the next free event is available
- **ET_ASYNC**, the call returns immediately with a status.
- **ET_TIMED**, the call waits for the amount of time given by the fifth argument if no events are immediately available. The time specified is a minimum. Obtaining read access to a station's input list could take some additional time.

The fifth argument is the time to wait for a new event to become available before timing out if the ET_TIMED option was chose.

The sixth and last argument is the requested event size in bytes. If the size is larger than the size specified when the ET system was created, for a C-based ET system the new event will be declared a special "temporary" event and ET will allocate the necessary memory as an additional memory mapped file. This memory is automatically freed when the event arrives back at Grand Central. This mechanism is slow and designed to deal with rare oversize events but is completely transparent to the user. If on the other hand a Java-based ET system is used, more memory is allocated for the event. Once increased in size, it stays that way.

To get an array of new events call:

> **et_events_new(et_sys_id id, et_att_id att, et_event *pe[], int wait, struct timespec *time, int size, int num, int *nread)**

Most of the arguments are the same as for et_event_new except that now pe is an array of pointers to events, num is the number of events desired, and nread is the number of events actually placed into the array (which may be less than what was asked for).

There are situations in which several producers work in parallel and simultaneously request new events. There is a danger that some producer will take all of the available events leaving others waiting. This can lead to a deadlock, for example, if a consumer downstream blocks waiting for an event from every producer. To prevent this, Grand Central’s new event pool can be divided into groups (see

et_system_config**_settcp(et_sysconfig config int rBufSize, int sBufSize, int noDelay)** : for remote users set the parameters of the TCP connection to the client: the sizes of the TCP send & receive buffers and the TCP no delay value.

et_system_config_setgroups).

To get an event from a group, the producer calls:

> **et_events_new_group(et_sys_id id, et_att_id att, et_event *pe[], int wait, struct timespec *time, int size, int num, int group, int *nread)**.

This is similar to et_events_new with the exception that only events of the specified event group will be read. An alternative is to call the function:

> **et_system_setgroup(et_sys_id id, int group)**

which sets the default group to retrieve events from. Any subsequent calls to et_event(s)_new will only retrieve events from the default group. This is useful if each producer is a separate process, but beware that in a multi-thread program et_system_setgroup will change the default group for all threads. So, if there are multiple consumers running as different threads in the same program et_events_new_group must be used.

Using et_system_setgroup to set the group to 0 restores the system default.

5.8.2 Getting data-filled events

To read a single event use:

et_event_get(et_sys_id id, et_att_id att, et_event **pe, int wait, struct timespec *time)

The arguments are the same as those for creating a new event but without the size. To read an array of events use:

et_events_get(et_sys_id id, et_att_id att, et_event **pe, int wait, struct timespec *time, int num, int *nread)

The arguments are almost the same as for reading single events except that the user passes in pe an array of pointers to events and additional arguments specify the number of events the user wants to read and a pointer to the number actually read, which may be less than the number requested.

5.8.3 Modifying events

After reading an event, the user has access to a number of its properties for manipulation. Routines to accomplish this are given in the following list:

- **et_event_setpriority(et_event *pe, int pri)** : sets the priority of an event, pri, to be ET_HIGH or ET_LOW (default). A high priority means that such an event gets placed below other high priority but above low priority events when placed in a station's input or output list. Thus, high priority events are always the first to be read. No other guarantees are made.
- **et_event_getpriority(et_event *pe, int *pri)** : gets the priority of an event.
- **et_event_setlength(et_event *pe, int len)** : sets the length or size of the event's valid data in bytes. This may not be larger than the total amount of memory available in the event.
- **et_event_getlength(et_event *pe, int *len)** : gets the length of the event's valid data in bytes.
- **et_event_setcontrol(et_event *pe, int con[], int num)** : sets the control information of an event. The con argument is an array of integers which control the flow of the event through the ET system, and the num argument gives the size of the array. The maximum size of this array is determined at compile time by ET_STATION_SELECT_INTS which defaults to 6.
- **et_event_getcontrol(et_event *pe, int con[])** : gets the event's array of control information.

- **et_event_getdata(et_event *pe, void **data)** : gets a void pointer to the start of an event's data.
- **et_event_getdatastatus(et_event *pe, int *status)** : gets the status of an event's data. It can be either ET_DATA_OK, ET_DATA_CORRUPT (not currently used), or ET_DATA_POSSIBLY_CORRUPT. Data is ET_DATA_OK unless a previous user got the event from the system and then exited or crashed without putting it back. If the ET system recovers that event and puts it back into the system, its status becomes ET_DATA_POSSIBLY_CORRUPT as a warning to others.
- **et_event_setendian(et_event *pe, int endian)** : though normally the ET system automatically keeps track of the endianness of an event's data, this routine can override and directly set the endian value of the data (but does NOT swap). It may be ET_ENDIAN_BIG, ET_ENDIAN_LITTLE, ET_ENDIAN_LOCAL (same endian as local host), ET_ENDIAN_NOTLOCAL (opposite endian as local host), or ET_ENDIAN_SWITCH. See the chapter on Remote ET.
- **et_event_getendian(et_event *pe, int *endian)** : gets the endianness of an event's data - either ET_ENDIAN_BIG or ET_ENDIAN_LITTLE. See the chapter on Remote ET.
- **et_event_needtoswap(et_event *pe, int *swap)** : tells the caller if an event's data needs to be swapped or not by returning either ET_SWAP or ET_NOSWAP. See the chapter on Remote ET.

5.8.4 Putting events back

Once events have been read or new events gotten from the ET system, they MUST be put back into the ET system in order for other users to see them or so they can be recycled and used by producers again. After setting an event's parameters and writing data, the user is finished with the event and wishes to place it into the ET system. Or perhaps the user has only read the data and is done with the event. In any case, the event must be written back into the system by two possible means. Either write a single event with:

et_event_put(et_sys_id id, et_att_id att, et_event *pe)

or write multiple events with:

et_events_put(et_sys_id id, et_att_id att, et_event *pe[], int num).

In the latter case, the user gives the number, num, of events to put back in the array pe. This function never blocks as a station's output list has enough room for all events in the whole ET system.

The ET system checks to see if the attachment (att) that read the event is the same one that is writing it. If it isn't, the call returns an error and nothing is written.

5.8.5 Dumping events

After reading existing events or creating new ones, it's possible that these events may no longer be of interest to the user or any other user on the system. In that case, one may dump or recycle these events by calls to two routines. They are identical to the routines et_event(s)_put in their arguments. The first is:

et_event_dump(et_sys_id id, et_att_id att, et_event *pe)

and dumps a single event. Similarly,

et_events_dump(et_sys_id id, et_att_id att, et_event *pe[], int num)

dumps multiple events. The dump will place the events directly into Grand Central station's input list and so no stations downstream will see them.

5.9 Java jar users

5.9.1 Getting new (unused) events

To get an array of new events call:

EtEvent[] events = sys.newEvents(EtAttachment att, Mode mode, boolean noBuffer, int microSec, int count, int size, int group);

The first argument is the attachment and the second is the mode which can be:

- **Mode.SLEEP**, the call will block until the next free event is available
- **Mode.ASYNC**, the call returns immediately event if no events are available
- **Mode.TIMED**, the call waits for the amount of time given by the fourth argument if no events are immediately available. The time specified is a minimum. Obtaining read access to a station's input list could take some additional time.

If the third arg is true, the events will have no data buffer. They must be provided for each event with a call to:

events[i].setDataBuffer(ByteBuffer buf);

The fifth argument is the number of events desired, and the sixth is the requested event size in bytes. If the size is larger than the size specified when the ET system was created, for a C-based ET system the new event will be declared a special "temporary" event and ET will allocate the necessary memory as an additional memory mapped file. This memory is automatically freed when the event arrives back at Grand Central. This mechanism is slow and designed to deal with rare oversize events but is completely transparent to the user. If on the other hand a Java-based ET system is used, more memory is allocated for the event. Once increased in size, it stays that way.

There are situations in which several producers work in parallel and simultaneously request new events. There is a danger that some producer will take all of the available events leaving others waiting. This can lead to a deadlock, for example, if a consumer downstream blocks waiting for an event from every producer. To prevent this, Grand Central's new event pool can be divided into groups. The last argument is the group from which the events are to be taken. ET systems default to 1 group which is specified by group = 1.

Note that there is a simpler, overloaded form of the newEvents() method available with fewer arguments.

5.9.2 Getting data-filled events

To read an array of events use:

EtEvents[] events = sys.getEvents(EtAttachment att, Mode mode, Modify, modify, int microSec, int count);

The first two arguments and the fourth are the same as those for newEvents(). The third specifies whether the user will modify the data, only the metadata (header), or nothing at all with these values:

- **Modify.ANYTHING**
- **Modify.HEADER**
- **Modify.NOTHING**

And the last argument specifies the number of events requested.

5.9.3 Modifying events

After reading an event, the user has access to a number of its properties for manipulation. Methods to accomplish this are given in the following list:

- **setPriority(Priority pri)** : sets the priority of an event, pri, to be Priority.HIGH or Priority.LOW (default). A high priority means that such an event gets placed below other high priority but above low priority events when placed in a station's input or output list. Thus, high priority events are always the first to be read. No other guarantees are made.
- **getPriority()** : gets the priority of an event.
- **setLength(nt len)** : sets the length or size of the event's valid data in bytes. This may not be larger than the total amount of memory available in the event.
- **getLength()** : gets the length of the event's valid data in bytes.
- **setControl(int con[])** : sets the control information of an event. The con argument is an array of integers which control the flow of the event through the ET system. The maximum size of this array is given by EtConstants.stationSelectInts which defaults to 6.
- **getControl()** : gets the event's array of control information.
- **getData()** : returns event's data as a byte array.
- **getDataBuffer**(): returns event’s data as a ByteBuffer object.
- **getDataStatus()** : gets the status of an event's data. It can be either DataStatus.OK, DataStatus.CORRUPT (not currently used), or DataStatus.POSSIBLYCORRUPT. Data is OK unless a previous user got the event from the system and then exited or crashed

without putting it back. If the ET system recovers that event and puts it back into the system, its status becomes POSSIBLYCORRUPT as a warning to others.

- **setByteOrder(int endian)** : though normally the ET system automatically keeps track of the endianness of an event's data, this method can override and directly set the endian value of the data (but does NOT swap). It may be EtConstants.endianBig, EtConstants.endianLittle, EtConstants.endianLocal (same endian as local host), EtConstants.endianNotLocal (opposite endian as local host), or EtConstants.endianSwitch. See the chapter on Remote ET.
- **setByteOrder(ByteOrder order)** : setByteOrder() is overloaded to accept values of ByteOrder.BIG_ENDIAN and ByteOrder.LITTLE_ENDIAN.
- **getByteOrder()** : gets the endianness of an event's data as either ByteOrder.BIG_ENDIAN or ByteOrder.LITTLE_ENDIAN. See the chapter on Remote ET.
- **setRawByteOrder(int raw)** : set the event’s byte order to big with a raw = 0x04030201 or little with raw = 0x01020304. For internal use only.
- **getRawByteOrder()** : gets the events byte order as 0x04030201 if big or0x01020304 if little. For internal use only.
- **needToSwap()** : tells the caller if an event's data needs to be swapped or not by returning either true or false. See the chapter on Remote ET.
- **getAge()** : returns Age.NEW if event obtained through calling newEvents() or Age.USED if obtained through calling getEvents().
- **getGroup()** : gets the group number event belongs to.
- **getOwner()** : gets the attachment id of the attachment object (an integer) used to obtain the event or EtConstants.system if owned by the ET system.
- **getMemSize()** : gets the max memory in bytes available for this event (data buffer size).
- **getModify()** : gets whether getEvents() was called with Modify.ANYTHING, Modify.HEADER, or Modify.NOTHING.
- **getId()** : get event’s id number (no possible use for the user). For internal use only.

5.9.4 Putting events back

Once events have been read or new events gotten from the ET system, they MUST be put back into the ET system in order for other users to see them or so they can be recycled and used by producers again. After setting events’ parameters and writing data, the user is finished with the events and wishes to place them into the ET system. Or perhaps the user has only read the data and is done with them. In any case, the events must be written back into the system with:

sys.putEvents(EtAttachment att, EtEvent[] events, int offset, int len)

In addition to specifying the attachment and the array of events to put back, the user may also specify the offset into the “events” array and the number of elements to put. Other, simpler overloaded versions of putEvents() exist. This method never blocks as a station's output list has enough room for all events in the whole ET system.

The ET system checks to see if the attachment that read the event is the same one that is writing it. If it isn't, the call throws an exception and nothing is written.

5.9.5 Dumping events

After reading existing events or creating new ones, it's possible that these events may no longer be of interest to the user or any other user on the system. In that case, one may dump or recycle these events by calling:

sys.dumpEvents(EtAttachment att, EtEvent[] events, int offset, int len)

with args are the same as for putEvents(). The dump will place the events directly into Grand Central station’s input list and so no stations downstream will see them.

.

Section

6

6. ET programming in C

This chapter gives some details on programming with an ET system in the C language. It answers questions about program flow, handling signals, useful ET library functions, how to define user functions for selecting events, and various odds & ends.

6.1 Program flow

Being such a complicated, multithreaded, multi-process system, it is probably not at all obvious how a user would put all the calls to the ET library together in a coherent manner. Given below is a bare bones outline of how a local user's program could look.

```
/* declare variables */
int status;
et_statconfig sconfig;
et_openconfig openconfig;
et_event  *pe;
et_sys_id  id;
et_stat_id my_stat
et_att_id  attach;

/* open ET system */
et_open_config_init(&openconfig);
et_open(&id," /tmp/my_et_system_file", openconfig);
et_open_config_destroy(openconfig);

/* define station */
et_station_config_init(&sconfig);
et_station_config_setblock(sconfig, ET_STATION_BLOCKING);
et_station_config_setselect(sconfig, ET_STATION_SELECT_ALL);
et_station_config_setuser(sconfig, ET_STATION_USER_SINGLE);
et_station_config_setrestore(sconfig, ET_STATION_RESTORE_OUT);

/* create and attach to station */
et_station_create(id, &my_stat, "my_station", sconfig);
et_station_attach(id, my_stat, &attach);

/* get and put events */
while(et_alive(id)) {
   status = et_event_get(id, attach, &pe, ET_SLEEP, NULL);
   status = et_event_put(id, attach, pe);
}
```

The very first thing to do is to create access to the ET system with et_open. This maps the given file into the user's memory and also starts a heartbeat and begins to listen for the ET system's heartbeat. Correspondingly, an et_close will close access to the ET system.

At this point one can create any desired station or use one created by another application and then attach to one of them. By attaching, one receives a unique identifier (attach in this case) that will be used in the rest of the transactions.

Once finished attaching, one can read and write events, checking every now and then to see if the ET system is alive. If the ET system dies while the user is waiting to get events, the get call will return with the error ET_ERROR_DEAD. Although not shown in this code, be sure to carefully check the status of each read and write call.

6.2 Handling signals

Because the ET software uses multiple POSIX threads, signal handling must be done carefully. Be sure to use POSIX routines and only those that are thread safe. Refer to the book Programming with POSIX Threads by David Butenhof for a good reference on this subject.

Functions that meet this standard are pthread_sigmask, pthread_kill, sigwait, sigwaitinfo, and sigtimedwait. When masking signals, use the function pthread_sigmask NOT sigprocmask since its behavior in a threaded process is undefined.

The best way to handle things is to initially block or mask all signals with pthread_sigmask. Once the user has called et_open, the new threads that were started as a result of calling it will also have all signals blocked because the new threads inherit the signal mask of its parent thread. Once the ET system is open, handle the signal catching in the main thread or some additional thread spawned from the main thread (see et_client.c). If a separate signal handling thread is used, it can use sigwait to wait for specific signals. It is very convenient to do things this way, but care must be taken as the main thread continues execution even as the signal handler is being run.

6.3 Defining a function for event selection

Should the user wish to provide his own event selection algorithm for a station, this may be accommodated by defining a function especially for that purpose. The function must be part of a shared library and must have the arguments:

my_selection_function (et_sys_id id, et_stat_id stat_id, et_event *pe) .

This function will be called whenever the ET system places events into the station’s input list. The return value must be one for a selected event and zero for a rejected one. Make this function known to the ET system by calling et_station_config_setfunction() and et_station_config_lib() to provide the function and library names so the function can be dynamically loaded.

An example is provided in the source code. Look in the .../et/src/examples directory and at two files. The first, shown below, is et_userfunction.c:

```
#include “et.h”
int et_users_function(et_sys_id id, et_stat_id stat_id, et_event *pe)
{
   int select[ET_STATION_SELECT_INTS],
   control[ET_STATION_SELECT_INTS];


   et_station_getselectwords(id, stat_id, select);
```

```
    et_event_getcontrol(pe, control);

    /* access event control ints thru control[N] */
    /* access station selection ints thru select[N] */
    /* return 0 if it is NOT selected, 1 if it is */


    if (some condition) {
        return 1;
    }
    return 0;
}
```

The first argument is the ET system id which gives the user access to all system information, the second is the station the user is selecting events for, and the last is a pointer to the event being filtered. Return one (1) if the event is selected, and zero (0) if it is not.

Notice that the routines et_station_getselectwords and et_event_getcontrol will prove extremely useful as they allow the user access to all the selection and control integers. The name of this function is completely up to the user. The only obvious restriction is that it should not conflict with names in the ET library (look in et.h and et_private.h). The name of the file is also up to the user.

The second file of interest is SConscript. There is one place where et_userfunction.c is compiled into a shared library but is commented out. A single obvious edit will allow the library to be created. The name of the shared library is again up to the user.

The names of your function and shared library are parameters in the definition of a station and are thus subject to a length limit. The function name is limited to ET_FUNCNAME_LENGTH - 1 chars and the lib name is limited to ET_FILENAME_LENGTH - 1 chars. These limits are enforced in the routines et_station_config_setfunction and et_station_config_setlib.

The function-writer has access to the event's data through functions mentioned in the previous chapter. Similarly, there is access to information about the station's state through the following ET library functions:

- **et_station_getattachments(et_sys_id id, et_stat_id stat_id, int *numatts)** : gets the number of attachments to a station.
- **et_station_getstatus(et_sys_id id, et_stat_id stat_id, int *status)** : gets a station's status.
- **et_station_getblock(et_sys_id id, et_stat_id stat_id, int *block)** : gets a station's blocking mode
- **et_station_getrestore(et_sys_id id, et_stat_id stat_id, int *restore)** : gets a station's restore mode
- **et_station_getuser(et_sys_id id, et_stat_id stat_id, int *user)** : gets a station's user mode
- **et_station_getprescale(et_sys_id id, et_stat_id stat_id, int *prescale)** : gets a station's prescale value
- **et_station_getcue(et_sys_id id, et_stat_id stat_id, int *cue)** : gets a station's cue value

- **et_station_getselect(et_sys_id id, et_stat_id stat_id, int *select)** : gets a station's select mode
- **et_station_getselectwords(et_sys_id id, et_stat_id stat_id, int *select)** : gets a station's selection integer array
- **et_station_getlib(et_sys_id id, et_stat_id stat_id, char *lib)** : gets a station's select function's shared library name
- **et_station_getfunction(et_sys_id id, et_stat_id stat_id, char *function)** : gets a station's select function name
- **et_station_getinputcount(et_sys_id id, et_stat_id stat_id, int *cnt)** : gets the number of events in a station's input list. This function may not be so useful in that this data can change so quickly.
- **et_station_getoutputcount(et_sys_id id, et_stat_id stat_id, int *cnt)** : gets the number of events in a station's output list. This function may not be so useful in that this data can change so quickly.

Using these functions, all relevant information about the ET system necessary to select events for a particular station can be obtained.

6.4 ET utility functions

There are a number of other routines available to the ET system users to get information about stations:

- **et_station_name_to_id(et_sys_id id, et_stat_id *stat_id, char *name)** : returns a station id given a station's name.
- **et_station_isattached(et_sys_id id, et_stat_id stat_id, et_att_id att)** : tells if "att" is attached to a station.
- **et_station_exists(et_sys_id id, et_stat_id *stat_id, char *stat_name)** : tells if a station exists and returns its id.

There are routines available to get information about an ET system:

- **et_system_getnumevents(et_sys_id id, int *numevents)** : tells how many events a system has.
- **et_system_geteventsize(et_sys_id id, int *eventsize)** : tells the size in bytes of a system's events.
- **et_system_getlocality(et_sys_id id, int *locality)** : tells whether the ET system is on a remote node or is local, or is local on a system which cannot share mutexes.
- **et_system_getpid(et_sys_id id, pid_t *pid)** : gives the unix process id or pid of the ET system process.
- **et_system_getheartbeat(et_sys_id id, int *heartbeat)** : tells the heartbeat count.

- **et_system_getattsmax(et_sys_id id, int *attsmax)** : tells the max number of attachments allowed..
- **et_system_getstationsmax(et_sys_id id, int *stationsmax)** : tells the max number of stations allowed.
- **et_system_gettempsmax(et_sys_id id, int *tempsmax)** : tells the max number of temporary events allowed.
- **et_system_getprocsmax(et_sys_id id, int *procsmax)** : tells the max number of processes allowed to open the ET system locally.
- **et_system_getattachments(et_sys_id id, int *atts)** : tells the current number of attachments.
- **et_system_getstations(et_sys_id id, int *stations)** : tells the current number of stations.
- **et_system_gettemps(et_sys_id id, int *temps)** : tells current number of temporary events.
- **et_system_getprocs(et_sys_id id, int *procs)** : tells the current number of processes with the ET system open locally.
- **et_system_gethost(et_sys_id id, char *host)** : tells which host computer the ET system is running on.
- **et_system_getserverport(et_sys_id id, unsigned short *port)** : tells the port number of the ET system's TCP server thread.

Some routines affecting attachments are:

- **et_wakeup_attachment(et_sys_id id, et_att_id att)** : this routine wakes up a particular attachment which is currently blocked on an event read call on a particular station.
- **et_wakeup_all(et_sys_id id, et_stat_id stat_id)** : this routine wakes up all attachments which are currently blocked on an event read call on a particular station.
- **et_attach_geteventsput(et_sys_id id, et_attt_id att_id, uint64_t *count)** : this routine gets the number of events put into a station by an attachment..
- **et_attach_geteventsget(et_sys_id id, et_attt_id att_id, uint64_t *count)** : this routine gets the number of events gotten from a station by an attachment.
- **et_attach_geteventsdump(et_sys_id id, et_attt_id att_id, uint64_t *count)** : this routine gets the number of events dumped by an attachment.
- **et_attach_geteventsmake(et_sys_id id, et_attt_id att_id, uint64_t *count)** : this routine gets the number of new events gotten from a station by an attachment.

Then there are:

- **et_alive(et_sys_id id)** : returns 1 if the ET system is alive and 0 if it is not.

- **et_wait_for_alive(et_sys_id id)** : waits indefinitely until the ET system is alive and then returns.

6.5 Multiple attachments to blocking stations

Having multiple attachments to blocking stations from the same program is a bad idea unless care is taken to thread the program. The problem arises when the read and write statements of a program are done serially in a single logical loop. Since blocking stations have input lists large enough to hold all of the events in the ET system it is possible that all the events are in the input list of one station while the program is waiting to read events from another. Check your logic carefully.

Similar problems can arise when producing events at an attachment, to a station other than Grand Central, that is also being used for reading or consuming events. The difficulty is that if the program blocks when calling et_event_new, all the available events may have previously filled up the station's input list. In this situation the call to et_event_new will never return.

6.6 C includes, flags, and libraries

Using the ET system library from C requires users to include the file et.h. The name of the ET shared library is libet.so, and the name of the static library is libet.a. See the SConstruct file in the ET distribution for other possibly necessary libraries.

On both Solaris and Linux, pthread mutexes have the default behavior such that if a mutex is locked by some thread, any other thread may unlock it. This is non-portable behavior and must not be relied on according to the man pages. However, its use is very convenient when recovering from a crashed process which has locked one or more mutexes. The alternative method to recover from such situations is to re-initialize the locked mutexes. Such behavior can be implemented at compile time by specifying the flag "-DMUTEX_INIT".

6.7 Debug output

To help in finding problems and finding out information about an active ET system, users can adjust the debug output printed by the system. The two routines used for this purpose are:

- **et_system_setdebug(et_sys_id id, int debug)** : sets the level of debug output desired.
- **et_system_getdebug(et_sys_id id, int *debug)** : gets a system's current debug level.

The possible values of the argument debug are:

- ET_DEBUG_NONE - this value results in no output
- ET_DEBUG_SEVERE - this value outputs only the most severe errors
- ET_DEBUG_ERROR - this value outputs all errors
- ET_DEBUG_WARN - this value outputs all errors and all warnings
- ET_DEBUG_INFO - this value outputs everything including informational output

The debug level of an ET system or consumer defaults to ET_DEBUG_ERROR. Notice that the debug level of a system can only be set after the call to et_open or et_system_start.

Normally, by default, debug output is simply printed by means of printf statements. If the user wishes to use the coda routine daLogMsg to output debug messages, simply recompile ET with the flag -DWITH_DALOGMSG. Be sure to link with the library libcmlog.so when doing so.

Section
7

7. ET programming in Java

This chapter gives some details on programming with an ET system in the Java language. It answers questions about program flow, useful ET classes and methods, how to define user functions for selecting events, and various odds & ends.

7.1 Program flow

Being such a complicated, multithreaded, multi-process system, it is probably not at all obvious how a user would put all the calls to the ET library together in a coherent manner. Given below is a bare bones outline of how a user's program could look.

```
// open ET system directly
String etFile = “/tmp/myEtFile”;
int port = EtConstants.serverPort;
String host = “129.57.29.111";

EtSystemOpenConfig config = new EtSystemOpenConfig(etFile, host, port);
EtSystem sys = new EtSystem(config);
sys.open();

// define station
EtStationConfig statConfig = new EtStationConfig();
statConfig.setBlockMode(EtConstants.stationBblocking);
statConfig.setUserMode(EtConstants.stationUserSingle);
statConfig.setRestoreMode(EtConstants.stationRestoreOut);

// create and attach to station
EtStation station = sys.createStation(statConfig, "my_station");
EtAttachment attach = sys.attach(station);

// get and put events
int chunk = 10;
EtEvents[] events;
while(sys.alive()) {
    events = sys.getEvents(attach, Mode.SLEEP, null, 0 , chunk);
    sys.putEvents(attach, events);
}
```

The very first thing to do is to create access to the ET system with sys.open(). Correspondingly, sys.close() it will close access to the ET system.

At this point one can create any desired station or use one created by another application and then attach to one of them. By attaching, one receives a unique identifier (the attach object in this case) that will be used in the rest of the transactions.

Once finished attaching, one can read and write events, checking every now and then to see if the ET system is alive. If the ET system dies while the user is waiting to get events, the get call will throw an EtDeadException. Although not shown in this code, be sure to catch all exceptions for each read and write call.

7.2 Defining a method for event selection

Should the user wish to provide his own event selection algorithm for a station, this may be accommodated by defining a class especially for that purpose. The class must be in the JVM's classpath and must implement the one method in the EtEventSelectable interface:

public boolean select (SystemCreate sys, StationLocal station, EtEvent event)

This method will be called whenever the ET system places events into the station's input list. The return value must be true for a selected event and false for a rejected one. Make this method known to the ET system when creating by calling:

stationConfig.setSelectClass(String className)

to provide the class name so it can be dynamically loaded.

An example is provided in the source code. Look in the .../et/java/org/jlab/coda/et directory and at EtStationSelection.java shown below:

```
public class EtStationSelection implements EtEventSelectable {

   public boolean select(SystemCreate sys, StationLocal stat, EtEvent ev) {
          int[] select  = stat.getConfig().getSelect();
          int[] control = ev.getControl();


          // access event control ints thru control[N]
          // access station selection ints thru select[N]
          // return false if it is NOT selected, true if it is

          if (some condition) {
                 return true;
          }
          return false;
   }
}
```

The first argument is the ET system id which gives the user access to all system information, the second is the station the user is selecting events for, and the last is a pointer to the event being filtered. Return true if the event is selected, and false if it is not.

Notice that stat.getConfig().getSelect() and ev.getControl() will prove extremely useful as they allow the user access to all the selection and control integers. The name of the class is up to the user except that it is limited in length to EtConstants.fileNameLengthMax chars. This limit is enforced in stationConfig.setSelectClass().

The class-writer has access to the event's data through methods mentioned in the previous chapter on events. Similarly, there is access to the station's state through the methods of the StationLocal class. Some of the more useful methods are listed below:

- **getAttachments()** : gets the number of attachments to a station.
- **getStatus()** : gets a station's status as one of EtConstants.stationUnused (not fully created yet), EtConstants.stationCreating (used in C ET systems to indicate station is in the process of being created), EtConstants.stationIdle (station exists but has no attachments), or EtConstants.stationActive (station exists and has at least one attachment).
- **getConfig()** : gets all of a station's configuration info
- **getInputList()** : gets a station's input list (e.g. to get number of events there)
- **getOutputList()** : gets a station's output list (e.g. to get number of events there)

Similarly, the class-writer has access to system information through the SystemCreate object by calling sys.getConfig().

7.3 ET utility functions

There are a number of other methods available to the ET system users to get information about stations using the EtSystem object:

- **stationNameToObject(String name)** : returns a station object given a station's name or null if none.
- **stationAttached(EtStation station, EtAttachment att)** : tells if "att" is attached to “station”.
- **stationExists(String name)** : tells if a station by that name exists.

There are methods available to get information about an ET system from the same object:

- **getNumEvents()** : gets how many events a system has.
- **getEventSize()** : gets the size in bytes of a system's events.
- **getPid()** : gets the unix process id or pid of the ET system process (only for C-based system).
- **getHeartbeat()** : gets the heartbeat count (only for C-based system).
- **getAttachmentsMax()** : gets the max number of attachments allowed..
- **getStationsMax()** : gets the max number of stations allowed.
- **getTempsMax()** : gets the max number of temporary events allowed (only for C-based system).
- **getProcessesMax()** : gets the max number of processes allowed to open the ET system locally (only for C-based system).
- **getNumAttachments()** : gets the current number of attachments.
- **getNumStations()** : gets the current number of stations.

- **getNumTemps()** : gets the current number of temporary events (only for C-based system).
- **getNumProcesses()** : gets the current number of processes open locally (only for C-based system).
- **getHost()** : gets which host computer the ET system is running on.
- **getTcpPort()** : tells the port number of the ET system's TCP server thread.
- **getLanguage()** : language used to implement the ET system. Will be EtConstants.langJava, EtConstants.langC, or EtConstants.langCpp.
- **getGroups()** : gets the number of groups events are divided into.
- **getGroupCount()** : gets the array in which the values are the number of events in each group and the index + 1 is the group number (starts at 1).

Some methods affecting attachments are:

- **wakeUpAttachment(EtAttachment att)** : wakes up an attachment currently blocked on an event read call due to empty station input list.
- **wakeUpAll(EtStation station)** : wakes up all attachments currently blocked on an event read call on a particular station.

Then there is:

- **alive()** : this returns true if the ET system is alive and false if it is not.

7.4 Multiple attachments to blocking stations

Having multiple attachments to blocking stations from the same program is a bad idea unless care is taken to thread the program. The problem arises when the read and write statements of a program are done serially in a single logical loop. Since blocking stations have input lists large enough to hold all of the events in the ET system it is possible that all the events are in the input list of one station while the program is waiting to read events from another. Check your logic carefully.

Similar problems can arise when producing events at an attachment, to a station other than Grand Central, that is also being used for reading or consuming events. The difficulty is that if the program blocks when calling newEvents(), all the available events may have previously filled up the station's input list. In this situation the call to newEvents() will never return.

7.5 Debug output

To help in finding problems and finding out information about an active ET system, users can adjust the debug output printed by the system. The two routines used for this purpose are:

- **sys.setDebug(int debug)** : sets the level of debug output desired.
- **sys.getDebug()** : gets a system's current debug level.

The possible values of the argument debug are:

- EtConstants.debugNone - results in no output
- EtConstants.debugSevere - outputs only the most severe errors
- EtConstants.debugError - outputs all errors
- EtConstants.debugWarn - outputs all errors and all warnings
- EtConstants.debugInfo - outputs everything including informational output

The debug level of an ET system or consumer defaults to EtConstants.debugError. Notice that the debug level of a system can be set in the EtSystem object’s constructor in addition to the above method.

Section

8

8. Fine tuning the ET system

This chapter provides information for the reader who needs to tune the ET system for better or different performance.

8.1 ET version numbering

The C header file et_private.h defines the macro ET_VERSION whose value denotes the major version of the ET software while ET_MINORVERSION denotes the minor version (in Java, EtConstants.version and EtConstants.minorVersion). When a user calls et_open, the routine checks to see if its major version and the major version of the ET system it is opening is the same. If not, an error is returned. Thus, when a user makes fundamental changes to the ET software and recompiles it, the value of ET_VERSION should also be changed to another value, say something over 1,000. Giving the version a large number allows the author and distributors of ET to use the version number for successive releases of the software without conflicting with the versions a user makes with specific modifications. In this way, incompatible versions of ET will always give users a warning.

Modifying the definitions of constants defined in et.h, such as ET_STATION_SELECT_INTS, ET_ATTACHMENTS_MAX, ET_FILENAME_LENGTH, or ET_STATNAME_LENGTH, may cause problems if the user is not careful (in Java, EtConstants.stationSelectInts, EtConstants.attachmentsMax, EtConstants.fileNameLengthMax, and EtConstants.stationNameLengthMax). Difficulties may arise when more than one ET library exist - each with different definitions of one of the above constants. When network communications occur between consumers using one library and ET systems using another library, it is likely that one of the processes involved will crash. Thus, for these modifications, be sure to change ET_VERSION.

8.2 Event Selection

8.2.1 Adding more selection integers

For users who need additional control over the flow of their events, take a look at the file et.h. It is possible to increase the value of the macro ET_STATION_SELECT_INTS (in Java, EtConstants.stationSelectInts), which defaults to six selection integers, and recompile ET. This results in the simultaneously increase of both the number of select words (actually integers) for each station and also the corresponding number of control words (integers) of each event.

Changing the value of ET_STATION_SELECT_INTS and recompiling can cause fatal errors if an ET system and all its users are not using either the same shared-library/jar or one compiled with an identical configuration. Also network communications will fail with unpredictable results. The way to avoid potential problems of this type is to assign another version number to modified ET systems (libraries) by changing the value of ET_VERSION in et_private.h (see above).

8.2.2 Setting heartbeat and heartmonitor periods in C

There are two time periods that are adjustable by modifying their values in et_private.h and recompiling ET. The first of these two periods is the time between heartbeats. As the reader should be aware of by now, each process opening an ET system has a thread start up which provides a heartbeat. By default it is set to a 0.5 seconds:

```
#define ET_BEAT_SEC  0
#define ET_BEAT_NSEC 500000000
```

The second is time period between readings of the system heartbeat if you are a user or consumer heartbeats if you are the system. Remember that upon opening an ET system, another thread starts which monitors the appropriate heartbeats. The default monitor period is 1.6 seconds:

```
#define ET_MON_SEC  1
#define ET_MON_NSEC 600000000
```

The crucial point to remember is that the heartbeat must be faster than the heartmonitor. If the heartmonitor finds that the system heartbeat has not changed in successive monitorings, then it declares that the ET system is dead. The same is true for the system monitoring consumers. If your process declares that the ET system is dead, no further dealings with it are possible.

Notice that the default has a large safety margin built in. The hearts are beating more than three times faster than the monitors are looking at them. This ensures that flakiness in UNIX's handling of timing, sleeping, and the scheduling of processes will not interfere.

The advantage of decreasing the beat and monitor times is that the system and user processes have a much quicker response to the world. The disadvantage is that it slows down the performance of the whole system. The author has run with a beat time of 0.3 seconds and a monitor time of 1 second with no problems.

The reader should be aware that on Unix systems the clock is 100Hz, meaning that when a thread or process encounters a sleep or nanosleep command or is swapped out, it does nothing for a minimum of 0.01 seconds.

8.2.3 Setting the number of attachments and processes

In specifying the configuration of a system, which is passed on to the routine et_system_start, the user can specify the maximum number of attachments and the maximum number of processes which can use the ET system being created. Both of these values are limited however. They cannot exceed the values set by the macros ET_ATTACHMENTS_MAX and ET_PROCESSES_MAX. The reason for doing it that way is that programming is greatly simplified.

By looking in the file et_private.h, the reader can see that the default value of ET_ATTACHMENTS_MAX is 50 and that the macro ET_PROCESSES_MAX is set to ET_ATTACHMENTS_MAX. If more attachments or processes are desired, then these 2 values can be increased and ET must be recompiled. (Be sure to change ET_VERSION as well).

In Java things are simpler since there are no hard upper limits. The max number of attachments can be set by the SystemConfig.setAttachmentsMax() without restriction and Java does not concern itself with processes.

8.2.4 Setting defaults

Although a user can set ET system parameters such as the number of events and their size, it may be nice if some of these parameters could be made the default. This is possible by editing a few lines in the file et.h. The value of a station's cue and prescale along with the value of a system's number of events, max number of temporary events, size of events, and max number of stations can be set to a user's preferred default by changing (respectively): ET_STATION_CUE, ET_STATION_PRESCALE, ET_SYSTEM_EVENTS, ET_STATION_ESIZE, ET_STATION_NSTATS, ET_SYSTEM_NTEMPS. A recompilation is necessary.

In Java these can be set by respectively changing EtConstants.defaultCue, EtConstants.defaultPrescale, EtConstants.defaultNumEvents, EtConstants.defaultEventSize, and EtConstants.defaultStationsMax (temp events don’t exist in Java). Recreating the jar file is necessary after any such changes.

Section

9

9. Remote ET

It is possible to have an ET system on one machine and its consumers on another (remote consumers). Remote consumers can make all the calls that local ones can. Of course, the speed of transferring events over the network is quite a bit slower than the speed of accessing shared memory. With a Java-based ET system, all consumers are remote since there is no shared memory being used.

The way this is done is that each ET system has a built-in server. That is, there is a TCP listening thread which handles all interactions between the ET system and remote consumers. There is also a listening thread for responding to UDP packets from remote consumers trying to find an ET system somewhere on the network by broadcasting or multicasting. A UDP response packet is to send back by the ET system containing the port number of its TCP listening thread, its host's name, all of its IP addresses and other info as well. Using this info, consumers can then make the final TCP socket connection with the server which handles all the receiving and sending of events and other information.

9.1 Direct connection

There are times when using either broadcasting or multicasting is inconvenient or impossible. For example, if an ET system and a consumer are on different subnets, broadcasting from one to the other is stopped by any routers unless such are reprogrammed to allow broadcasting to get through - a hassle in any case. In situations such as these, a direct connection can be made.

Remote consumers need to know the TCP server's port number and the host name that the ET system resides on. Then using et_open_config_setserverport the port can be set, using et_open_config_sethost the host can be set, and using et_open_config_setcast a direct connection can be specified with ET_DIRECT. In Java, openConfig.setTcpPort(), openConfig.setHost(), and openConfig.setNetworkContactMethod(EtConstants.direct) do the same thing.

9.2 Broadcasting

Broadcasting is done to IP addresses which in dotted-decimal form (e.g. 128.7.6.35) can be represented as {netid, subnetid, hostid}. The only type of broadcast address used in ET systems is subnet-directed and is of the form {netid, subnetid,-1} where -1 simply means that that part of the address is composed of all 1's in binary. For example, if 128.7.6 is the subnet with a mask of 255.255.255.0, then 128.6.7.255 is the broadcast address for that subnet. A broadcast will be received by all machines on that particular subnet. You may find the broadcast address(es) of your subnet(s) by using the command "ifconfig -a".

An ET system automatically responds to broadcasts on all its local subnets and no configuration is necessary or possible.

An ET consumer, by default, broadcasts on all its local subnets to find ET systems. Otherwise, one can use the et_open_config_setcast routine to set the configuration to a setting of ET_BROADCAST or ET_BROADANDMULTICAST to do so. Call et_open_config_addbroadcast to add a specific broadcast address to the list of active broadcast addresses. Use it with a value of ET_SUBNET_ALL to add all the local broadcast addresses to the list (the default remember). Likewise, call et_open_config_removebroadcast to remove addresses from the active list with ET_SUBNET_ALL removing all broadcast addresses.

To do the same thing In Java, use config.setNetworkContactMethod() with either EtConstanst.broadcast or EtConstanst.broadAndMulticast as the argument. Call config.addBroadcastAddr() to add an address and config.setBroadcastAddrs() to set the entire list of addresses to use. Since ET broadcasts on all subnets by default, normally adding or setting these addresses is not necessary. Setting the addresses to all local subnets can be done by hand:

```
HashSet<String> allSubnetAddrs =
          new HashSet<String> (EtUtils.getAllBroadcastAddresses());
config.setBroadcastAddrs(allSubnetAddrs);
```

9.3 Multicasting

In multicasting, a consumer sends out a packet to a special multicast IP address. The listeners (ET systems) sign up to receive any packets send to that address and only computers hosting such listeners will receive the packets - not all machines on the subnet as is the case in broadcasting. Multicasting has the ability to go beyond the local subnet and thus is more flexible than broadcasting. The following table 9.1 lists all available multicast addresses as well as "TTL" values (reproduced from Unix Network Programming, Volume 1 by Richard Stevens):

Scope	IPv6 Scope	IPv4	
		TTL Scope	Administrative Scope
node-local	1	0	
link-local	2	1	224.0.0.0 to 224.0.0.225
site-local	5	<32	239.255.0.0 to 239.255.255.255

organization-local	8		239.192.0.0 to 239.195.255.255
global	14	<255	224.0.1.0 to 238.255.255.255

Table 9.1 Multicast Addresses

The use of TTL values and ranges of addresses is meant to set the range or the scope of the multicasts. Setting the TTL value is recommended practice with a default value of 32 meaning the local site only. However, administrative scoping is preferred when possible. The range 239.0.0.0 to 239.255.255.255 is the administratively scoped IPv4 multicast space. "Addresses in this range are assigned locally by an organization but are not guaranteed to be unique across organizational boundaries. An organization must configure its boundary routers (multicast routers at the boundary of the organization) not to forward multicast packets destined to any of these addresses".

In short, pick an address between 239.0.0.0 and 239.255.255.255 for use at one particular site. If this is confusing, talk to your system administrator and ask for a safe multicast address for your use. The default TTL value used in ET is 32 while the default multicast address is ET_MULTICAST_ADDR (in Java, EtConstants.multicastAddr) which is defined as 239.200.0.0.

A C-based ET system can respond to multicasts on up to ET_MAXADDRESSES (defined in et_private.h as 10) multicast addresses. A C ET consumer can multicast by using the et_open_config_setcast routine to set the configuration to a setting of ET_MULTICAST or ET_BROADANDMULTICAST. Call et_open_config_addmulticast to add a specific multicast address to the list of active addresses. Likewise, call et_open_config_removemulticast to remove addresses from the list. Use et_open_config_setTTL to set the TTL value of the multicast. Both broadcasting and multicasting may be done simultaneously by specifying ET_BROADANDMULTICAST as an argument for et_open_config_setcast.

A Java-based ET system can respond to multicasts to any number of multicast addresses. A Java ET consumer can multicast by using the openConfig.setNetworkContactMethod() method to set the configuration to a setting of EtConstants.multicast or EtConstants.broadAndMulticast. Call openConfig_addMulticastAddr() or openConfig.setMulticastAddrs() to add specify multicast addresses. Likewise, call openConfig.removeMulticastAddr() to remove addresses from the list. Use openConfig.setTTL() to set the TTL value. Both broadcasting and multicasting may be done simultaneously by specifying EtConstants.broadAndMulticast as the contact method.

9.4 Port selection for broad/multicasting

In addition to choosing broadcasting and/or multicasting and corresponding addresses, the user must also choose the port number for these communications. The Internet Assigned Numbers Authority (IANA) states that the range of port numbers from 0 to 1023 are controlled and assigned by the IANA. Thus, these are unavailable. The ports

1024 to 49151 are *not* controlled by the IANA and are available for use, but the IANA registers and lists the uses of these ports as a convenience to the internet community. For example, ports 6000 to 6063 are assigned for an X window server for both TCP and UDP. Generally, the higher numbered ports are less likely to be used. Finally, ports 49152 to 65535 are called dynamic or private or ephemeral ports. The IANA says nothing about these.

Use the routine et_system_config_setport (sysConfig.setUdpPort() and sysConfig.setMulticastPort()) to configure an ET system to listen for broad/multicasts on a particular port. Use et_open_config_setport (openConfig.setUdpPort()) to configure a consumer to send broadcasts and et_open_config_setmultiport (openConfig.setMulticastPort()) to send multicasts on a particular port. The port numbers used by the consumer must be the same as those used by the ET system for things to work. By default, if not set explicitly, they are set to ET_BROADCAST_PORT and ET_MULTICAST_PORT (EtConstants.broadcastPort and EtConstants.multicastPort) respectively. In C both are defined as 11111 in et.h, but in Java the multicast port is set to 11112.

9.5 Defaults

When defining a configuration in C to use in opening an ET system, the defaults are to use broadcasting only to port ET_BROADCAST_PORT (defined as 11111 in et.h) on all local subnet addresses. The macro ET_MULTICAST_PORT is also similarly defined to be 11111, while the macro ET_MULTICAST_ADDR is defined to be "239.200.0.0". The value of ET_MULTICAST_TTL is 32. All of these macros are only defined for the users' convenience.

In Java, there is no single default, but there are several different constructors for the EtSystemOpenConfig object so chose carefully. There are a couple of constructors designed for broadcasting, a couple for multicasting and one for a direct connection. Giving port arguments values of 0 or address lists as null result in the above default values being used. See the Javadoc for more details.

9.6 Examples creating an ET system

When setting up an ET system, very little needs to be done to allow it to be discovered by broadcasting consumers:

```
/* In C */
et_sys_id id;
et_sysconfig config;
/* Initialize configuration */
et_system_config_init(&config);
/* Set ET file name */
et_system_config_setfile(config, "/<dir>/<myEtFile>");
/* Start ET system, listens to broadcasts by default */
et_system_start(&id, config);
/* Release configuration's allocated memory */
et_system_config_destroy(config);
```

```
// In Java it’s even simpler
SystemConfig config = new SystemConfig();
SystemCreate sys = new SystemCreate("/<dir>/<myEtFile>", config);
```

When setting up an ET system for both broadcasting and multicasting, try the following:

```
et_sys_id id;
et_sysconfig config;
et_system_config_init(&config);
et_system_config_setfile(config, "/<dir>/<myEtFile>");
/* Already listening for broadcasts */
/* Listen for multicasts to these 2 addresses: */
et_system_config_addmulticast(config, ET_MULTICAST_ADDR);
et_system_config_addmulticast(config, "239.111.222.0");
et_system_start(&id, config);
et_system_config_destroy(config);

// In Java
SystemConfig config = new SystemConfig();
config.addMulticastAddr(EtConstants.multicastAddr);
config.addMulticastAddr("239.111.222.0");
SystemCreate sys = new SystemCreate("/<dir>/<myEtFile>", config);
```

When setting up an ET system with specified ports, try the following:

```
et_sys_id id;
et_sysconfig config;
et_system_config_init(&config);
et_system_config_setfile(config, "/<dir>/<myEtFile>");
/* Remote users broad/multicast to this port */
et_system_config_setport(config, ET_BROADCAST_PORT);
/* Set port of tcp server thread */
et_system_config_setserverport(config, 11222);
et_system_start(&id, config);
et_system_config_destroy(config);


// In Java
SystemConfig config = new SystemConfig();
config.setUdpPort(EtConstants.broadcastPort);
config.setTcpPort(11222);
SystemCreate sys = new SystemCreate("/<dir>/<myEtFile>", config);
```

9.7 Examples creating an ET consumer

When setting up a consumer to open an ET system on an unknown host which may be anywhere (local or remote), and it's trying to find that system using broadcasting on all local subnets, then include the following code:

```
et_sys_id id;
et_openconfig config;
et_open_config_init(&config);
/* Broadcasting by default */
/* ET is on an unknown host */
et_open_config_sethost(config, ET_HOST_ANYWHERE);
```

```
et_open(&id, "et_name", config);
et_open_config_destroy(config);


// In Java
EtSystemOpenConfig config = new EtSystemOpenConfig("et_name",
                                     EtConstanst.hostAnywhere);
EtSystem sys = new EtSystem(config);
sys.open();
```

When setting up a consumer that knows the ET system is on a different host, and is trying to find it using multicasting on port ET_MULTICAST_PORT at address ET_MULTICAST_ADDR, then include the following code:

```
et_sys_id id;
et_openconfig config;
et_open_config_init(&config);
/* ET is remote */
et_open_config_sethost(config, ET_HOST_REMOTE);
/* Use multicast to find ET system */
et_open_config_setcast(config, ET_MULTICAST);
/* Remote users multicast to this port */
et_open_config_setmultiport(config, ET_MULTICAST_PORT);
/* Remote users multicast to this address */
et_open_config_addmulticast(config, ET_MULTICAST_ADDR);
et_open(&id, "et_name", config);
et_open_config_destroy(config);


// In Java (last 0 denotes default TTL value)
ArrayList<String> addrs = new ArrayList<String>();
addrs.add(EtConstants.multicastAddr);
EtSystemOpenConfig config = new EtSystemOpenConfig("et_name",
          EtConstanst.hostRemote, addrs,
          EtConstants.multicastPort, 0);
EtSystem sys = new EtSystem(config);
sys.open();
```

When setting up a consumer that knows the name of the host running the ET system (ethost.mylab.org) but nothing else, and is trying to find that system using both broadcasting and multicasting at address 239.235.89.12, then include the following code:

```
et_sys_id id;
et_openconfig config;
et_open_config_init(&config);
/* ET is running on ethost.mylab.org */
et_open_config_sethost(config, "ethost.mylab.org");
/* Use broad and multicasting to find ET system */
et_open_config_setcast(config, ET_BROADANDMULTICAST);
/* Remote users multicast to this address */
et_open_config_addmulticast(config, "239.235.89.12");
et_open(&id, "et_name", config);
et_open_config_destroy(config);


// In Java
ArrayList<String> addrs = new ArrayList<String>();
addrs.add("239.235.89.12");
```

```
EtSystemOpenConfig config = new EtSystemOpenConfig("et_name",
          "ethost.mylab.org", null, addrs, false,
          EtConstants.broadAndMulticast, 0, 0, 0, 0,
          EtConstants.policyFirst);
EtSystem sys = new EtSystem(config);
sys.open();
```

When setting up a consumer to open an ET system on a known host (129.182.54.67), and trying to directly connect to it on server port 12345 (bypassing all UDP communications), then include the following code:

```
et_sys_id id;
et_openconfig config;
et_open_config_init(&config);
/* ET is on 129.182.54.67 */
et_open_config_sethost(config, "129.182.54.67");
/* Use a direct connection to the ET system */
et_open_config_setcast(config, ET_DIRECT);
/* ET system's server is on this port */
et_open_config_setserverport(config, 12345);
et_open(&id, "et_name", config);
et_open_config_destroy(config);


// In Java
EtSystemOpenConfig config = new EtSystemOpenConfig("et_name",
                                   "129.182.54.67", 12345);
EtSystem sys = new EtSystem(config);
sys.open();
```

9.8 Network interface selection

There are occasions when the ET consumer wants to select which network interfaces it wants to use when communicating with the ET system. Often hosts have multiple interfaces – perhaps on different subnets and with different speeds. It is not unusual that a slower interface is used for control information while a faster one is used for data transfer. The two parts to this problem that must be considered are the general interface configuration and the consumer's specification of IP addresses and subnets.

9.8.1 Network interface configuration

Linux and MacOS, and to a lesser extent Solaris, have what is called a "weak end" model of network communication. This means all of a host's IP addresses are considered to belong to the host in general and not to a particular network interface. This can create a problem with ARP tables – tables which associate a specific IP address with a specific MAC hardware address. When an ARP request gets sent out, by default on Linux, a particular interface may respond with all the IP addresses on its host and the ARP table may end up with the interface's MAC address associated with an incorrect IP address. Thus a TCP packet may arrive at the correct host but on an incorrect network interface – one associated with a different IP address. What happens at this point is that Linux merely forwards the packet to the socket even though, strictly speaking, it came in the "wrong" interface.

How could this affect an ET system and its consumers? Say an ET system exists on a host with 2 interfaces, one fast and the other slow. It is possible that in an open-configuration a consumer would select the host it wants to connect to by specifying the IP address of the fast interface. Because of the ARP table's incorrect mapping, the ET consumer's TCP packets would end up being delivered to the slow interface on that host. They would still reach their intended destination but over the slow network connection.

The correction for this problem is fairly simple. It's possible to correct the ARP table (even across reboots) by making the following changes in Linux to the /etc/sysctl.conf file. Simply add the following 2 lines and reboot:

```
# Allow ARP reply only if the target IP address is local address
# configured on the incoming interface
net.ipv4.conf.default.arp_ignore = 1
```

To make the change without rebooting, in a console, type similar lines for each network interface:

```
% sysctl net.ipv4.conf.eth0.arg_ignore=1
% ifconfig eth0 down
% ifconfig eth0 up
```

Once the ARP table is correct, TCP packets will be delivered to the correct interface.

9.8.2 *Specification of network interfaces and subnets*

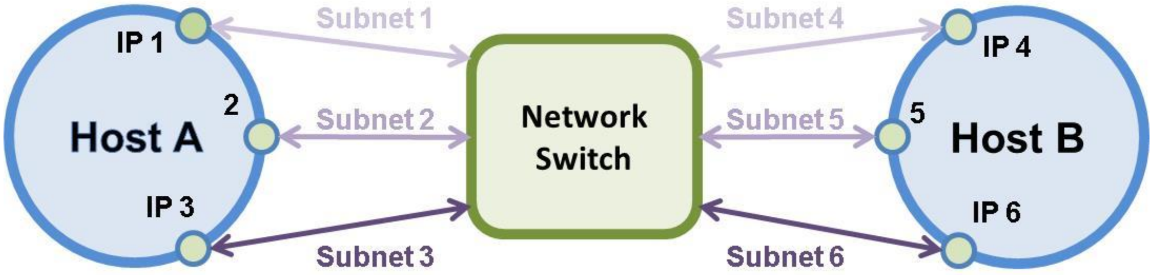


Figure 9.1 Multiple Network Interface Handling

Using the figure above as a reference, consider what is necessary to communicate over a specific interface and subnet. In figure 9.1 there are two hosts, each with 3 different IP addresses. Each address is associated with its own network interface card and subnet. Both hosts are connected to all the subnets through a switch. The question of interest is, "How can the user specify which subnets/interfaces to use when connecting to the ET system?"

Say, for example that the ET system is on host B, the consumer is on host A, subnets1 & 4 are fast (10G ethernet), and all other subnets are slow (1G ethernet). Furthermore, we want data to only flow over the fast network. If subnets 1 & 4 are the same, then having the consumer specifying IP 4 as the receiving end of the TCP socket guarantees all communication happens only over subnet 1/4. But if subnets 1 & 4 are ***not*** the same, then only specifying the receiving IP address does ***not*** mean that packets leaving host A will go through fast subnet 1. In order to ensure that packets leaving host A do go over subnet 1, the socket must bind the sending address to IP 1. Specifying both ends of the socket is the only way to ensure communication over the desired subnets.

Let’s translate that to the ET system. It is possible to select a preferred, common subnet between system and client by calling openConfig.setNetworkInterface(ip) in Java (et_open_config_setinterface in C) prior to opening. This method will take either a specific, local IP address or a local broadcast/subnet address in dot-decimal format as the argument. If given an IP address, it will convert that into its corresponding local broadcast address. If doing a broad/multicast to find ET, sys.open() will examine all the IP addresses of the ET system host and pick the first that exists on the preferred, common subnet. If one does not exist or was not specified by setNetworkInterface(), it will pick the first on a common subnet. If there is no common subnet, it will simply pick the first IP address in its list. Similarly, it will bind the local end of the socket to the IP address on the preferred subnet if specified. This guarantees network traffic over the preferred subnet if both system and client share it (subnet1 = subnet4 in the figure). However, if the preferred subnet is not local to the client, then all bets are off so to speak. In that case, the operating system chooses which client interface and which system IP address gets used. There is a way around this if the user really needs to specify both ends of the socket. Do this by connecting directly with a fixed IP address and specifying a preferred subnet at the same time. The direct connection will use the given system IP address and will bind the local end of the socket to the given subnet. For direct connections in general, be sure to specify the ET system host by dot-decimal IP address in order to set things up properly.

9.9 Remote Programming Details

9.9.1 Errors in C

Some remote user errors are given by ET_ERROR_REMOTE - those errors which are unique to a remote user and do not occur locally. In practice, this error is returned when memory cannot be allocated by the remote end. If there are errors in reading or writing over the network, the errors generated will be ET_ERROR_READ or ET_ERROR_WRITE.

9.9.2 Local C consumer and Java ET system

If a local C consumer tries to open the file of a Java-based ET system, it will recognize that and stop. It will then try to connect to the ET system as a remote consumer.

9.9.3 Local Java consumer and C ET system

If a local Java consumer tries to open a C-based ET system, it will recognize that and try to use the ET JNI library to access the shared memory directly for calls to getEvents(), putEvents, newEvent(), and dumpEvents(). All other communication is done through sockets. If the JNI library is not accessible, it will make a use the socket connection for those calls.

9.9.4 Remote behavior on a local host

It is possible to tell consumers to run the code that a remote consumer runs even if it is running on the same computer as the ET system. In this case, all communication with the ET system is done through sockets with no usage of the shared memory. In C this is done by calling et_open_config_setmode with the ET_HOST_AS_REMOTE option. The default mode is ET_HOST_AS_LOCAL. With a Java consumer and a Java ET system, everything is “remote” in the sense that sockets are always used With a Java consumer and a C ET system, by default the consumer will use JNI access to the ET system so it acts as a local C consumer. This can be changed by calling openConfig.setConnectRemotely(true) in which case sockets are used.

9.9.5 Getting new events

When a remote user is obtaining new events through calling et_event(s)_new, it is occasionally convenient to have a user-created buffer provide the data to be written into an ET event. In order to avoid an extra copy, the ET_SLEEP, ET_TIMED, or ET_ASYNC flag may be ORed with the flag:

ET_NOALLOC .

Normally, when a remote user calls et_event(s)_new, memory is automatically allocated for the buffer that will be holding the data being written. When this flag is set, however, this memory is NOT allocated. Instead, the user must call:

et_event_setdatabuffer(et_sys_id id, et_event *pe, void *data)

and provide his own data-filled buffer in the last argument. This avoids a copy of the data from some user buffer into the event’s buffer. When the user puts this event back into the ET system, the buffer is NOT freed as it would have been without the ET_NOALLOC flag.

In Java the same thing is accomplished when calling sys.newEvents() and its third arg is false, meaning that the user will supply the data buffer by calling,

event.setDataBuffer(ByteBuffer buf).

9.9.6 *Modifying events*

After opening an ET system, creating a station, and attaching to it, users are ready to start reading events. There are a few details to keep in mind when doing so remotely.

When a remote user calls et_event(s)_get (sys.getEvents in Java), the ET system sends a copy of the events over the network to the user and then immediately puts the originals back into the ET system with a call to et_event(s)_put (sys.putEvents). There may be times, however, when a user first wishes to modify the events and then send them back over the network to the ET system. To aid in this effort an extra flag is introduced in C:

ET_MODIFY .

By ORing this flag to ET_SLEEP, ET_TIMED, or ET_ASYNC, the user announces an intention to modify the requested event. In Java, the same thing is accomplished by specifying the third arg to sys.getEvents() as Modify.ANYTHING. Thus, when the ET server initially gets the event for the remote user, it does NOT put it back into the ET system immediately afterwards. It waits until the user has called et_event(s)_put (sys.putEvents) before doing that. Without this flag, the server puts the events back into the ET system immediately.

There may be occasions when the remote user doesn't want to modify the data but only the header information such as the priority, control words and such. In this case it makes no sense to send all the data back to the ET system when putting the event back. By using the flag:

ET_MODIFY_HEADER

instead of ET_MODIFY, only the header information will be sent back (in Java, Modify.HEADER) - speeding up communication greatly.

9.9.7 Getting data-filled events

When a remote user is obtaining events through calling et_event(s)_get, and is NOT modifying it (see above), the ET system which sent the event normally puts the events back into its local system immediately afterwards. The user has the option of having the ET system dump the events instead (send them directly back to Grand Central station). Do this by ORing the flag:

ET_DUMP

with the flag, ET_SLEEP, ET_TIMED, or ET_ASYNC.

In Java, this is not implemented.

9.9.8 Multithreading

If a remote consumer is a multi-threaded program, no special precautions are necessary as the ET library is thread-safe. However, if more than one thread uses the same ET system id obtained from a single call to et_open (in Java, a single EtSystem object), there will be a bottle neck as only one remote ET library function call at a time can be made. To avoid this problem, each thread that wants access to the ET system may to do its own et_open (in Java, EtSystem object) and thus communicate on its own socket to its own server thread. This should speed things up.

9.9.9 Swapping data in C

Transferring data between machines where one is big endian (the most significant byte is placed in the lowest memory address) and the other is little endian (the least significant byte is placed in the lowest memory address), requires the data to be "swapped". Since in general a user may not be knowledgeable about the machine on which a particular event was originally produced, a simple call to the function:

et_event_needtoswap(et_event *pe, int *swap)

will reveal whether the data needs to be swapped or not. If the return value placed in swap is ET_NOSWAP, no swapping is necessary; however, if the return value is ET_SWAP, then the opposite is true.

The ET system automatically keeps track of the endianness of an event's data. However, the user may want to forcibly set the data's endianness for some reason. In that case, a call to:

et_event_setendian(et_event *pe, int endian)

can be made. The endianness can be set to ET_ENDIAN_BIG, ET_ENDIAN_LITTLE, ET_ENDIAN_LOCAL (same endian as local host), ET_ENDIAN_NOTLOCAL (opposite endian as local host), or ET_ENDIAN_SWITCH (switch the endian from whatever it is). This routine does NOT swap the data but simply keeps track of the data's endianness in the event's header. A user may also read the endianness of an event's data by a call to:

et_event_getendian(et_event *pe, int *endian).

It returns either ET_ENDIAN_BIG or ET_ENDIAN_LITTLE.

To do a swap of evio format data, use the routine provided in the evio library, evioswap(). To swap it in place one can do the following:

```
int toLocal = 1; // 0 if local host is same endian as data, else 1
uint32_t *dataPtr;
et_event_getdata(pe, (void **)&dataPrt);
evioswap(dataPtr, toLocal, NULL);
```

Users of data formats other than CODA format must write their own swapping routines.

Another routine of interest is:

et_system_getlocality(et_sys_id id, int *locality).

This returns the value ET_REMOTE in the variable locality if the ET system is remote, ET_LOCAL if it is local, and ET_LOCAL_NOSHARE is it is local but is using an operating system which does not allow sharing of pthread mutexes across processes (e.g. MacOS).

9.9.10 Swapping data in Java

Similar to the C library, Java-based ET code keeps track of an event’s endianness which can be accessed by the user. It can also indicate whether an event’s data needs to be swapped:

```
EtEvent ev;
ByteOrder order = ev.getByteOrder();
ev.setByteOrder(ByteOrder.LITTLE_ENDIAN);
boolean needToSwap = ev.needToSwap();
```

Endianness can be set with ev.setByteOrder() as seen above, but be aware that it does **not** do any actual data swapping.

If an event has evio format data, it can be swapped by using the ByteDataTransformer class and its swapEvent methods in the evio library (jar file):

```
EtEvent ev;
ByteDataTransformer.swapEvent(ev.getDataBuffer(), null, 0, 0);
```

9.9.11 Transferring events between two ET systems in C

While it is certainly possible for a user to copy events from one ET system and place them in another with "normal" ET function calls, the ET system provides a more efficient way to do this. By using ET's bridging software, unnecessary coping of the data may be eliminated from the procedure. Regardless of whether the ET systems are on the same or different computers or if the process running the bridging routine is on one or the other or on yet a third machine, the transfer should take place smoothly. It will save time except perhaps when both ET systems and the bridging process are on the same machine in which case only a single copy of the data is made - no different than when using the "normal" ET function calls. A call to the following function will take care of all the details:

et_events_bridge(et_sys_id id_from, et_sys_id id_to, et_att_id att_from, et_att_id att_to, int num, int *ntransferred, et_bridgeconfig bconfig).

The arguments are respectively: the ID of the ET system from which the events are copied, the ID of the ET system to which the events are going, the attachment to a station on the "from" ET system, the attachment to a station on the "to" ET system (usually an attachment to Grand Central), the total number of events desired to be transferred, the total number of events that were actually transferred at the routine's return, and a configuration argument that will be described shortly. The configuration argument may be NULL in which case defaults are used.

The configuration for bridging events is very similar to the configuration for opening a system or creating a system. There are a number of functions used to create and define the config argument. It is initialized by a call to:

et_bridge_config_init (et_bridgeconfig *config).

When the user is finished using the configuration,

et_bridge_config_destroy (et_bridgeconfig config)

must be called in order to properly release all memory used.

After initialization, calls can be made to functions which set various properties of the specific configuration. Calls to these setting functions will fail unless the configuration is first initialized. The functions used to SET these properties are listed below along with an explanation for each:

- **et_bridge_config_setmodefrom(et_bridgeconfig config, int val)** : setting val to ET_SLEEP, ET_TIMED, or ET_ASYNC determines the mode of getting events from the "from" ET system. The default is ET_SLEEP.
- **et_bridge_config_setmodeto(et_bridgeconfig config, int val)** : setting val to ET_SLEEP, ET_TIMED, or ET_ASYNC determines the mode of getting new events from the "to" ET system. The default is ET_SLEEP.
- **et_bridge_config_setchunkfrom(et_bridgeconfig config, int val)** : setting val sets the maximum number of events to get from the "from" ET system in a single call to et_events_get - the chunk size if you will. The default is 100.
- **et_bridge_config_setchunkto(et_bridgeconfig config, int val)** : setting val sets the maximum number of new events to get from the "to" ET system in a single call to et_events_new - the chunk size if you will. The default is 100.
- **et_bridge_config_settimeoutfrom(et_bridgeconfig config, struct timespec val)** : setting val sets the time to wait for the "from" ET system when the mode is set to ET_TIMED. The default is 0 sec.
- **et_bridge_config_settimeoutto(et_bridgeconfig config, struct timespec val)** : setting val sets the time to wait for the "to" ET system when the mode is set to ET_TIMED. The default is 0 sec.
- **et_bridge_config_setfunc(et_bridgeconfig config, ET_SWAP_FUNCPTR func)** : setting func to a function pointer (function name) means that the function will be called to swap data whenever it's determined to be necessary. Using this feature is a convenient way of swapping data while it's being moved from one ET system to another with no intervention from the user needed. The function must be of the form: int func(et_event *src, et_event *dest, int bytes, int same_endian). It returns ET_OK if successful, otherwise ET_ERROR. The arguments consists of: *src* which is a pointer to the event whose data is to be swapped, *dest* which is a pointer to the event where the swapped data goes, *bytes* which tells the length of the data in bytes, and *same_endian* which is a flag equaling one if the machine and the data are of the same endian and zero otherwise. This function must be able to work with src and dest being the same event. With this as a prototype, the user can write a routine which swaps data in the appropriate manner. Notice that the first two arguments are pointers to events and not data buffers. This allows the writer of such a routine to have access to any of the event's header information. In general, such functions should NOT call et_event_setendian in order to change the registered endian

value of the data. This is already taken care of in et_events_bridge. The default is NULL which means no swapping is done.

For swapping CODA format data, wrap evioswap() from the evio library into a function of the required signature, for example:

int mySwapFunction(et_event *src, et_event *dest, int bytes,

```
                   int same_endian) {

    int toLocal, swap;
    et_event_needtoswap(src, &swap);

    toLocal = (swap == ET_SWAP) ? 1 : 0;

    evioswap((uint32_t *)src->pdata, toLocal,
             (uint32_t *)dest->pdata);
}
```

There are corresponding et_bridge_config_get... functions to get the configuration values of everything except the swapping function.

Section

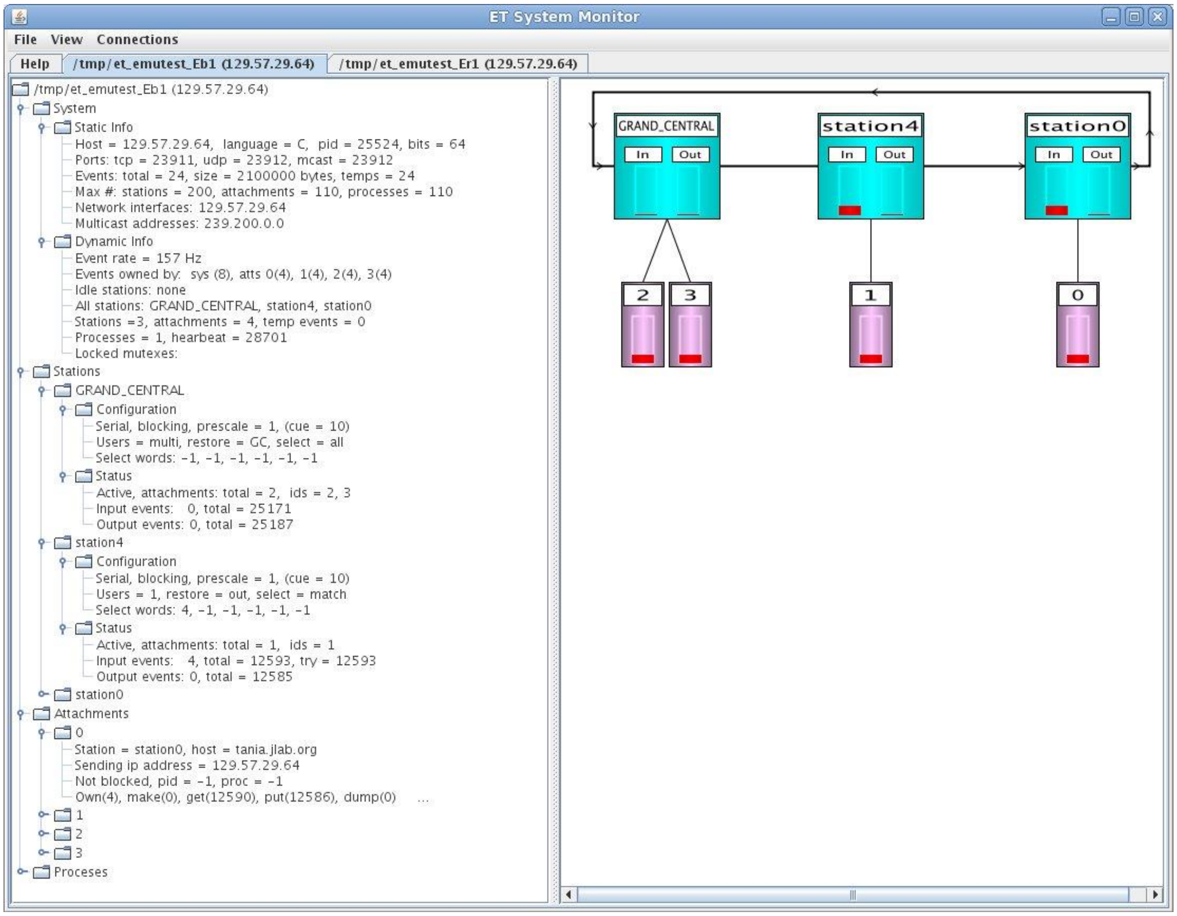
10

10. Monitoring

There are 2 different means to monitor ET systems. One is text-based and the other gui-based. The gui-based monitor requires Java in order to run.

10.1 Gui

There is a graphical ET monitoring application, the main window of which can be seen in figure 10.1 below. Written in Java, it is more conducive to seeing where all events are located in the system at a particular moment in time. This makes it a useful tool to find bottlenecks.



```
/tmp/et_emutest_Eb1 (129.57.29.64)
  System
    Static Info
      Host = 129.57.29.64,  language = C,  pid = 25524, bits = 64
      Ports: tcp = 23911, udp = 23912, mcast = 23912
      Events: total = 24, size = 2100000 bytes, temps = 24
      Max #: stations = 200, attachments = 110, processes = 110
      Network interfaces: 129.57.29.64
      Multicast addresses: 239.200.0.0
    Dynamic Info
      Event rate = 157 Hz
      Events owned by:  sys (8), atts 0(4), 1(4), 2(4), 3(4)
      Idle stations: none
      All stations: GRAND_CENTRAL, station4, station0
      Stations =3, attachments = 4, temp events = 0
      Processes = 1, hearbeat = 28701
      Locked mutexes:
  Stations
    GRAND_CENTRAL
      Configuration
        Serial, blocking, prescale = 1, (cue = 10)
        Users = multi, restore = GC, select = all
        Select words: -1, -1, -1, -1, -1, -1
      Status
        Active, attachments: total = 2,  ids = 2, 3
        Input events:   0, total = 25171
        Output events: 0, total = 25187
    station4
      Configuration
        Serial, blocking, prescale = 1, (cue = 10)
        Users = 1, restore = out, select = match
        Select words: 4, -1, -1, -1, -1, -1
      Status
        Active, attachments: total = 1,  ids = 1
        Input events:   4, total = 12593, try = 12593
        Output events: 0, total = 12585
    station0
  Attachments
    0
      Station = station0, host = tania.jlab.org
      Sending ip address = 129.57.29.64
      Not blocked, pid = -1, proc = -1
      Own(4), make(0), get(12590), put(12586), dump(0)    ...
    1
    2
    3
  Proceses
```

Figure 10.1 Main ET Monitor Gui

Its usage is:

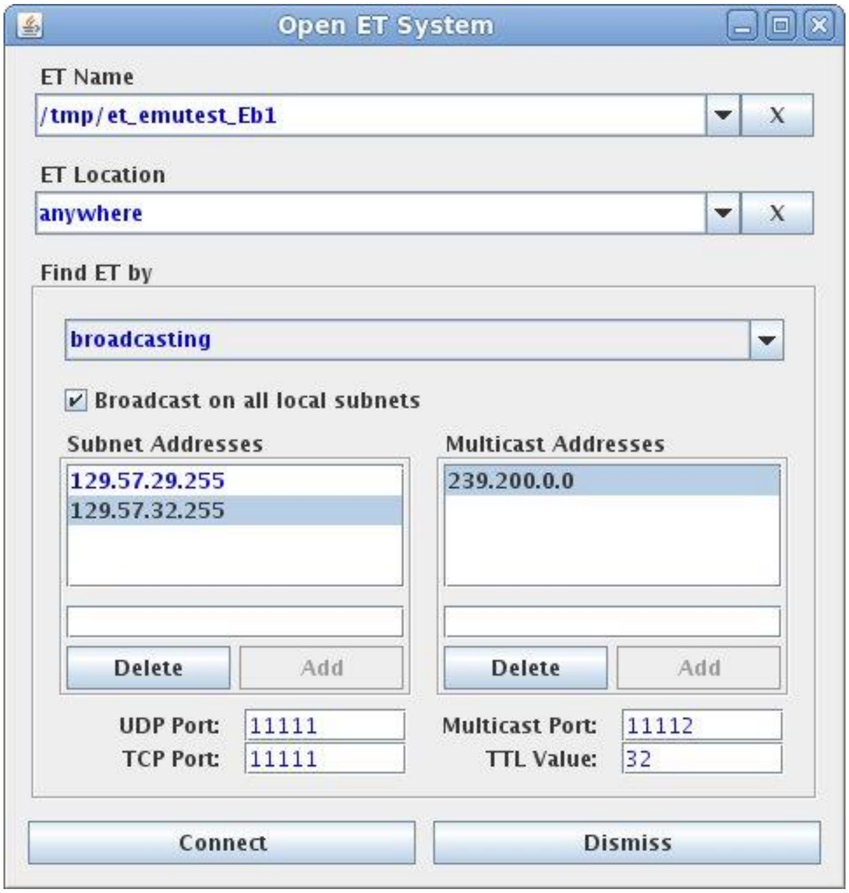
```
java org.jlab.coda.et.monitorGui.Monitor [-f, -file < configFile>]
```

Since most users don’t use config files specific to this program, one can just run

```
java org.jlab.coda.et.monitorGui.Monitor
```

Each connected ET system gets its own tab (along with the help page). Each tab is split into 2 parts: the text data on the left or top and the graphical forms on the right or bottom. In the text, both static and dynamic information is shown about the ET system in general, the stations, the attachments and the proceses. The graphical part shows the position of each station in the chain and the attachments to each station. The red bars show the percentage of the total number of events belonging to the attachments or the input/output lists of the stations. Anywhere a tall, red bar shows up, a bottleneck exists there.

In the “Connections” menu item of the above window, one can select the option to “Connect to ET System”. The window shown in figure 10.2 below pops up allowing the input of parameters necessary to connect to the ET system of interest. Once all the parameters are filled in, hit the “Connect” button to make the connection and display the system in the main gui.



ET Name: /tmp/et_emutest_Eb1

ET Location: anywhere

Find ET by: broadcasting

☑ Broadcast on all local subnets

Subnet Addresses:
129.57.29.255
129.57.32.255

Multicast Addresses:
239.200.0.0

UDP Port: 11111
TCP Port: 11111
Multicast Port: 11112
TTL Value: 32

Figure 10.2 Connection ET Monitor Window

Starting at the top of the connection window, enter the name of the ET system to open. The next input down is its location. There are 3 items preloaded into the comboBox (anywhere, remote, and local) which specify where to look for the ET system when broadcasting or multicasting. If making a direct connection, type the specific name of the host as the location. The next input down is the method to be used in finding the ET system. Its comboBox is preloaded with: broadcasting, multicasting, broad & multicasting, and direct connection. Choose one.

By default, the “Broadcast on all local subnets” button will be selected and all local subnet addresses will be loaded into the “Subnet Addresses” panel. The button can be unselected and various addresses added and/or removed from that address list to specify where broadcasts go. Reselecting the button will place all local subnet addresses back into the panel. Broadcasts go out over the port listed as the “UDP Port”. These parameters are, of course, only used when either broadcasting or broad&multicasting.

Similarly, the default multicast address used by ET systems is automatically loaded into the “Multicast Addresses” panel. Additional addresses can be entered for multicasting to different and/or multiple addresses at the port listed as “Multicast Port”. The TTL value may also be set. Again, these parameters are only used when either multicasting or broad&multicasting.

Finally, one can make a TCP connection directly to the ET system by choosing “direct connection”. In this case the “ET Location” must be the specific host name and the port listed as “TCP Port” is the one that is used.

Under main gui’s “View” menu and “Load Connection Parameters” option, all the settings of the connection window can be reset to those used in one of the existing connections.

10.2 Text

There is also a text-based program provided to monitor an ET system. It simply opens an ET system, reads its data, and then prints out the values that it reads there. If an ET user runs into difficulty, this program can help isolate the problems. The usage is:

```
usage: et_monitor  -f <ET name> [-h] [-r] [-host <ET host>]
                   [-t <time period (sec)>] [-p <ET server port>]
                   [-u <mcast port>]

          -host ET system's host (direct connection)
          -f ET system's (memory-mapped file) name
          -h help
          -r connect with local host as if remote
          -t time period in seconds between updates
          -p ET server port (direct connection)
          -u ET multicast port
```

It defaults to the local host with a period of 5 seconds between updates. If the user wants the monitor to communicate with the ET system as if remote even if it's local, use the -r option. The value of <ET host> can be provided in various formats. It can be an IP address in dotted-decimal form, the name of the host with or without the domain, ".local" or "localhost" which means look locally only, ".remote" which means look remotely only, or ".anywhere" which means any local or remote node which responds. If no specific host is given and the multicast port is given, multicasting is used to open the ET system; otherwise, a direct connection is attempted.

Here is an example output:

```
ET SYSTEM - (/tmp/et_emutest_Eb1) (host tania.jlab.org) (bits 64)
            (tcp port 23911) (udp port 23912) (multicast port 23912)
            (pid 11314) (lang C) (local) (period = 5 sec)

STATIC INFO - maximum of:
  events(24), event size(2100000), temps(24)
  stations(200), attaches(110), procs(110)
  network interfaces(1): 129.57.29.64,
  multicast addresses(1): 239.200.0.0,

DYNAMIC INFO - currently there are:
  processes(1), attachments(4), temps(0)
  stations(3), hearbeat(155)

STATIONS:
  "GRAND_CENTRAL" (id = 0)
    static info
      status(ACTIVE), flow(SERIAL), blocking(YES), user(MULTI), select(ALL)
      restore(GC), prescale(1), cue(10), select words(-1,-1,-1,-1,-1,-1,)
    dynamic info
      attachments: total#(2),  ids(2,3,)
      input  list: cnt =     6, events in  = 2
      output list: cnt =     0, events out = 2

  "station0" (id = 1)
    static info
      status(ACTIVE), flow(SERIAL), blocking(YES), user(1), select(MATCH)
      restore(OUT), prescale(1), cue(10), select words(0,-1,-1,-1,-1,-1,)
    dynamic info
      attachments: total#(1),  ids(0,)
      input  list: cnt =     0, events in  = 1, events try = 1
      output list: cnt =     0, events out = 1

  "station4" (id = 2)
    static info
      status(ACTIVE), flow(SERIAL), blocking(YES), user(1), select(MATCH)
      restore(OUT), prescale(1), cue(10), select words(4,-1,-1,-1,-1,-1,)
    dynamic info
      attachments: total#(1),  ids(1,)
      input  list: cnt =     0, events in  = 1, events try = 1
      output list: cnt =     0, events out = 1

LOCAL USERS:
  process #0, # attachments(0), pid(10516), hbeat(26)

ATTACHMENTS:
  att #0, is at station(station0) on host(tania.jlab.org)
             at pid(-1) from address(129.57.29.64)
```

```
              proc(-1), blocked(YES)
              events:  make(0), get(1), put(1), dump(0)
     att #1, is at station(station4) on host(tania.jlab.org)
                at pid(-1) from address(129.57.29.64)
              proc(-1), blocked(YES)
              events:  make(0), get(1), put(1), dump(0)
     att #2, is at station(GRAND_CENTRAL) on host(tania.jlab.org)
                at pid(-1) from address(129.57.29.64)
              proc(-1), blocked(NO)
              events:  make(10), get(0), put(1), dump(0)
     att #3, is at station(GRAND_CENTRAL) on host(tania.jlab.org)
                at pid(-1) from address(129.57.29.64)
              proc(-1), blocked(NO)
              events:  make(10), get(0), put(1), dump(0)

   EVENTS OWNED BY:
     system (6),  att0 (0),  att1 (0),  att2 (9),  att3 (9),

   EVENT RATE of GC = 0 events/sec

   CREATING STATIONS:
   IDLE STATIONS:
   STATION CHAIN:      GRAND_CENTRAL, station4, station0,
   LOCKED MUTEXES:

 *****************************************
```

The output lists a number of properties of the ET system like: name, host, port numbers, whether implemented in C or Java, whether local or not, host’s IP addresses, multicast addresses being listened to, event size, number of events, etc.. There are also a number of parameters listed which are subject to change such as: existing stations and their properties, all attachments and their properties, how many events are currently owned by which attachments, and the sequence of stations.