Event Filter Dataflow Software

Ch. Meessen[1], C. Bee[1], M. Bosman[2], E. Fede[1], K. Karr[2,]
A. Pacheco[2], Z. Qian[1], F. Touchard[1]

[1]CPPM Marseille, [2] IFAE Barcelona

Version : 1.1
Date : February 14, 2001

Abstract

This document describes the software handling the flow of events through the Event Filter. This is the fourth iteration of the implementation of the code. When compared to previous versions, enhancements come from the feedback by users and can be seen as simplifications while the interfaces have been kept identical.

The high level design of the dataflow is briefly reminded. Then, the details of the implementation of the components are given.

Table of Contents

1. Introduction

This document describes the software handling the dataflow in the Event Filter. The monitoring and control software is described elsewhere [1].

The version of the software and design described in this document is the fourth iteration on the design and implementation of the software. This version 4 is mainly a simplification of the version 3 implementation to take in account past experience and comments and criticism from users and reviewers.

The simplification is also the result of the experience gained in translating the dataflow software into Java. The implementation described in this document is done in C++, but we have adopted some Java ideas for the helper class: Socket, Thread, String?

Version 4 enhancements are:

- simplify building, combining and starting up of the different elements of the component,
- simplify access to the component parameters and statistics,
- simplify Socket and Thread management by implementing Java equivalent classes,
- simplify the processing task core element.

There are a few additions:

- provide IOElement to pass events via TCP between components using one thread per connection,
- provide IOElement to pass events by reference between components avoiding the need of threads,
- provide IOElement to randomly distribute event references between clients,

Protocols have not changed, so version 4 components are backwards compatible with version 3 components and they may inter−operate. From the event builder or the supervisor point of view, there is no difference between version 3 and version 4.

Since the global architecture and behaviour of the different component elements has not changed from version3, very limited effort will be required to port specific IOElement classes or processing tasks running with version 3 to the version 4.

Reading this document should be sufficient fort a good understanding of the Event Filter dataflow software and the architecture of the version 4 code.

This document also gives guide lines for users who want to implement their own processing task or implement their own event transfer protocol.

2. High level design of the event filter

2.1 general architecture

The Event Filter has been logically divided in three parts [2]: the distributor receives events from the SFI, processing tasks which perform selection and any other transformation on the event, and the collector which re−injects selected events into the SFO. In addition, the distributor has some sorting capability according to tags contained in the event header.
Each part of the Event Filter has been itself divided in one or several components (Figure 1):

- D1 implements the API with the SFI and performs the sorting operation. It will also provide a backup facility while the event is processed in the farm
- a D2 component is present for every event type
- the D3 component buffers events for the processing tasks. It can be omitted if the processing task directly requests events from D2
- PT implements the processing task
- C1 collects selected events. It has a function symmetric of the one of D3 and can be omitted too.
- C2 re−injects events into the SFO

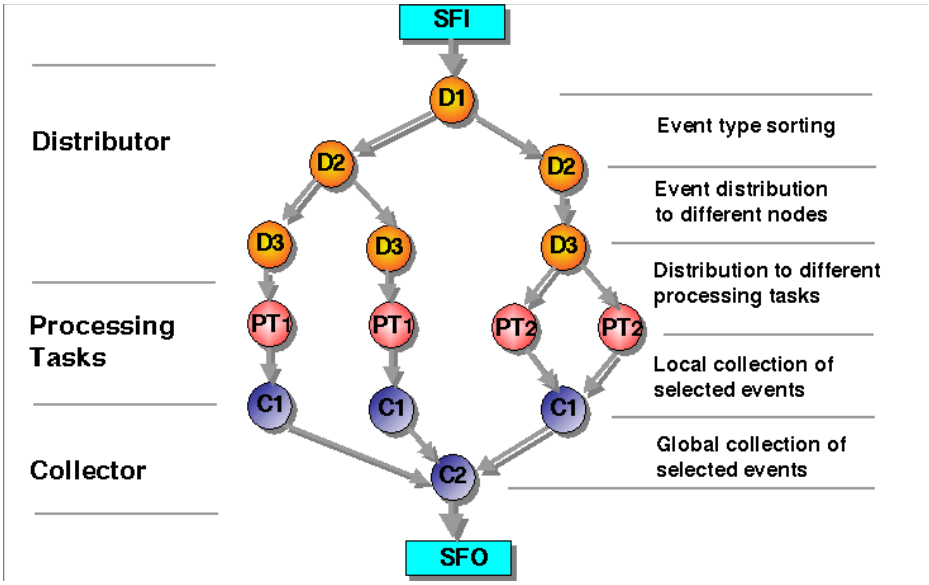

Figure 1: High Level Design of the Event filter

2.2 component architecture

All components can be seen as simple FIFO queues of a given depth. Events traverse them in a specific direction and may be processed and eventually deleted. Components are interconnected according to the client server paradigm. Servers accept connections and requests of multiple clients. Clients connect and interact with one server. A component can be client or server for the input or output of events. The Event Filter can thus be described by a directed graph without cycles where the summit is the component, and the directed links are the connections. Components have a unique name, the component identifier. If a source or a

destination are specified, the component will behave as client for that destination. If no source or no destination is specified, the component will behave as server for the specified side. Defining the graph is thus very simple and makes total abstraction of the protocol used to communicate between components. This is totally hidden to the user.

We currently have two types of components: a FIFO queue and a processing task. A FIFO component will simply queue events and behave as a buffer. The depth of the buffer is specified at component creation. Processing tasks may contain at most one event but will do some processing on it. According to a return value of the processing operation, the processing task component will delete the event or send it further on.

For user defined processing, one simply has to inherit the processing component class and override the event processing method. There is also a method to load a user specified file.

When building up a component, the user must specify the protocol she wants to use. Different protocols may well be used for components in the same address space than for components running in different processes or different computers. TCP is currently used for components distributed on different computers, and simple reference passing is used for event transfer between components in the same address space.

The component has an interface with the supervisor allowing to control and monitor the component's activity.

To make it easy to use different transfer and supervision protocols with each type of components we have divided the component into four elements.

The central element is the core element (`CoreElement`) that is also the component facade object. One can attach an input and output element (`IOElement`) which encapsulates the transfer protocol for events, and a control element (`CtlElement`) which provides the interface for the supervisor.

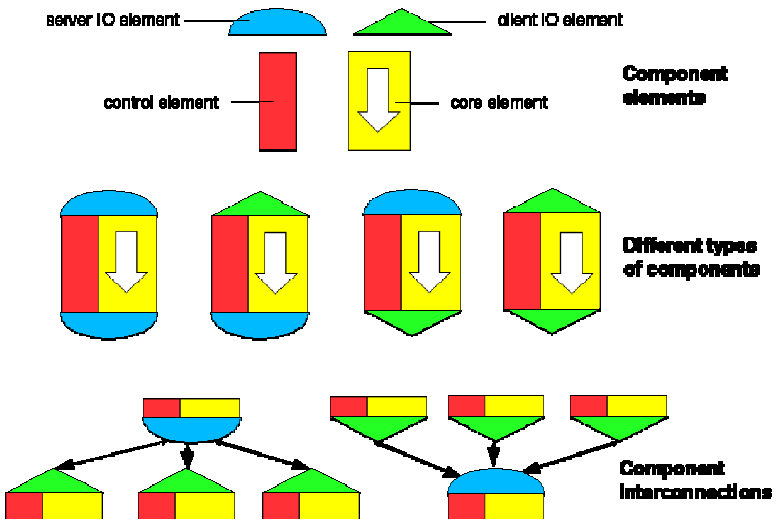

Figure 2: Different component elements and typical components

A component is thus an aggregation of elements and is constructed at startup time. We may combine these different elements in various combination so we can build many different components. It is also possible to modify at run time the component composition, although it is not used for the moment.

2.3 Events

From the dataflow point of view, events are blocks of bytes of different lengths. In the Event Filter, these byte blocks are wrapped into an object with associated information that is needed to simplify manipulating and transferring it. To simplify memory management, events are referred to by smart pointers which also take care of instance deletion when it is no longer referred to.

2.4 Event backup

When entering the Event Filter, events may have to be transferred across the network to different computers where they can be processed. In such a configuration, there are many possible problems that may arise that could result in event corruption or loss. To protect events from all these possible errors we propose the simple strategy of backing up the event on disk storage (for now at least) as soon as event enters the Event Filter.

This backup is stored in the so−called Distributer Global Buffer (DGB). The name comes from an old implementation design (historical reasons) and does not reflect anymore its actual role. This name will be kept for now although it is simply a backup repository for event currently traversing the Event Filter. Copies of events are removed from the DGB when they successfully leave the Event Filter (event accepted), or when they are deleted by a processing task (event rejection).

A more detailed policy for the handling of events in case of error during transfers or processing will be addressed later.

3. Component implementation

The component comprises a `CoreElement`, two `IOElement` and a `CtlElement` which we will described in this chapter. The last section describes how all these elements are bound together and how elemental components are turned into operational ones.

Depending on the required functionality, each of the four basic elements may or may not be run in their own thread: a processing task `CoreElement` is run in its own thread while a Fifo `CoreElement` is not; `RefIOElement` client and server are not run in a thread, but `TCPIOElement` servers are; `TCPIOElement` client may possibly be run in their own thread. More details will be given below.

3.1 the `CoreElement`

The `CoreElement` is the base class and corner stone of the component. All other elements are connected to it. To define a modified `CoreElement` behavior, the class must be derived from this `CoreElement` class.

3.1.1 General design

The `CoreElement` has a set of parameters that are accessible to the `IOElement` and the `CtlElement`. Table 1 gives a detailed description of the class.

Figure 3: Inheritance tree for the CoreElement

The variables of an instance of the class are public. In order to permit access of other elements to these variables, they should not be modified by the user.

CoreElement
parameters
+Id:String
+src:String
+dst:String
+ns:NSClient
+path:String
+host:String
+type:String
+reject:int
+depth:int
+runNbr:int
statistics
+nbrEvents:int
+inEvents:int
+outEvents:int
+nbrReject:int
+occupancy:double
+updateOccupancy()
+setRand(int)
+rand()
-rndVal:int
input output interface
+input:IOElement
+output:IOElement
+setInputElement(IOElement)
+setOutputElement(IOElement)
+reserveSpace()
+releaseSpace()
+putReserved(Event)
+reserveEvent():Event
+releaseEvent(Event)
+getReserved()
+delReserved()
control interface
+ctl:CtlElement*
+setControlElement(CtlElement)

Table 1: CoreElement class description

Events may be inserted and extracted from the `CoreElement` by using of a set of methods. Event insertion or extraction is split into a two step operation to support slow event transfer that can be cancelled.

The decoupling between the `IOElement` and the `CoreElement` is required so as to leave as much degree of liberty as possible in the functionality of these elements. A `CoreElement` may be a passive object like a `Fifo` in that does not call the `pullEventReq` or the `pushEventReq` methods of the `IOElement`. On the other hand, the need of a thread in each `IOElement` should be avoided whenever possible. Therefore, these `pushEventReq` and `pullEventReq` methods allow to execute all the methods required to fetch and insert the event. It is used for instance by the `RefIOElement` and by the `PTElement`.

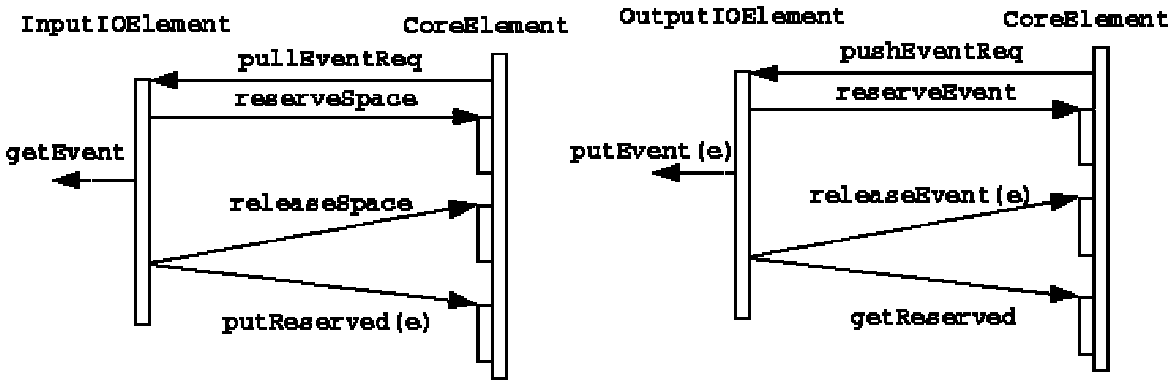

Figure 4: Sequence diagram for CoreElement interaction with IOElement clients. The left diagram shows how a CoreElement triggers event insertion and the right diagram shows how a CoreElement triggers an event extraction. releaseSpace or releaseEvent are called if the getEvent or putEvent operation failed.

`IOElements` that are run by their own thread or whose operation is executed by the client thread may ignore these `pullEventReq` or `pushEventReq` methods. But a `CoreElement` must call them so they can operate with any type of `IOElement`

Figure 4 shows a typical sequence diagram. Figure 6 and 7 show the same diagram for event transfer using a reference passing protocol to transfer events between components in the same address space. Figure 8 and 9 show the same sequence diagram but when TCP is used to transfer events.

To insert an event, one first reserves space in the component by calling the `reserveSpace` method. The call is blocking: the method returns as soon as the request has been satisfied. If the connection is broken, re−establishing it is performed internally. To complete the event insertion one must call the `putReserved(Event)` method. Once space has been reserved, it can be released to cancel the reservation by calling the `releaseSpace` method.

To extract events one first has to reserve an event by calling the `reserveEvent` method. This method returns a reference to the reserved event. As for the `reserveSpace` method, the call is blocking: the method returns as soon as the request has been satisfied. If the connection is broken, re−establishing it is performed internally. One may cancel the reservation by calling the `releaseEvent` method and passing back the event reference.

There are two ways to complete the operation. One may delete the event by calling the `delReserved` method or one may complete the extraction by calling the `getReserved` method. The method differs in the way that they update the component's statistics. A deleted event will be counted as a rejected event.

Event instance deletion is managed by the smart pointer so one does not need to care about it.

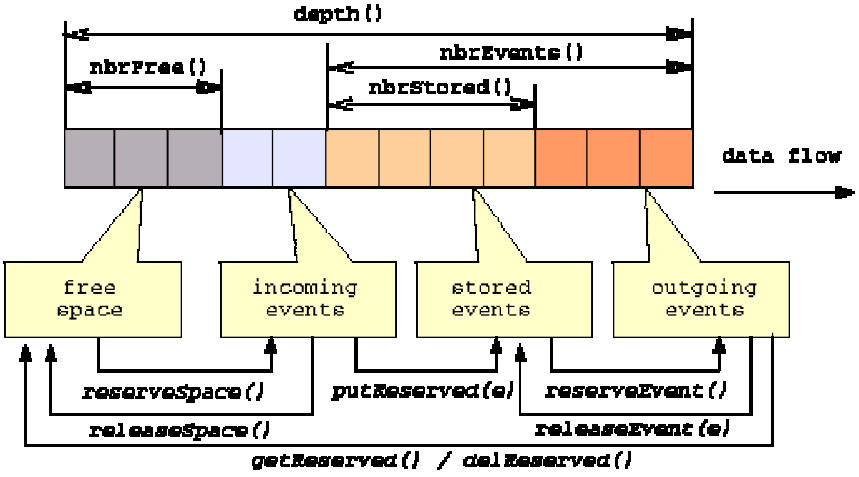

Figure 5: CoreElement internal structure and operations

3.1.2 The `FifoCoreElement`

The `FifoCoreElement` class is a simple buffer for events with a specified depth. It maintains a queue of events protected by a mutex for asynchronous access by the input and output elements and semaphores to control the `reserveSpace` and `reserveEvent` operations.

Since the component is passive and performs no special operations on the stored event, there is no thread associated to the `FifoCoreElement`. Thus if the input or output elements are clients, a thread that calls their `pullEventReq` and `pushEventReq` methods must be launched repetitively.

3.1.3 The `PTCoreElement`

The `PTCoreElement` is the base class for the processing task. Any user defined processing task must inherit form this class and override the virtual `processEvent(event)` method and possibly the `setGeometry(filePath)` method if access to a sort of configuration or parameter file is required. This method is called at startup time or on supervisor request, where the file path may also be changed. If requested by the supervisor, the command will only be called when the currently `processEvent` method is completed.

The `PTCoreElement` can only contain and process one event at a time. In order to allow prototype testing, additional parameters have been used: `PTLo`, `PTHi` and `Reject`. The `PTLo` and `PTHi` parameters specify the number of times the event must be processed (in order to crudely simulate different processing times). If the values are different and `PTHi` is bigger than `PTLo`, the `PTCoreElement` picks a random number of processing iteration between `PTLo` and `PTHi`. The current random number distribution is flat. The `Reject`

parameters, if different from zero, specifies the percentage of events that the component must reject. The rejected events are picked randomly.

The `PTCoreElement` requests an event for processing by calling the `pullEventReq` method of the `IOElement`. Then `pushEventReq` is called after the processing if the event has not been rejected. Otherwise, `delReserved` is called.

After having completed the insertion of an event into the `PTCoreElement`, the `IOElement` calls `putReserved`. This will set a semaphore so that the PT can start processing the event. When processing is completed, the PT will delete the event or push it out after having set a semaphore indicating that the event is ready. This strategy ensures that it is possible to use server or client for both input and output.

3.2 input−output element

3.2.1 General design

These elements are the links between components. They are responsible of the transfer of events using a specific protocol.

The `IOElement` class is the base class for all types of input and output element. A distinction is made between input and output `IOElements` and client and server `IOElements`.

Clients are connected to one server `IOElements`. An input client will *pull* events and an output client will *push* events. Server `IOElement` may have multiple simultaneous connected clients and waits and executes client requests.

The `IOElement` base class has only three methods: the `setCoreElement` method to bind it with the `CoreElement` and initialising and starting up the element; the `pullEventReq` method for the input element; the `pushEventReq` for the output element. Protocol specific derived classes may have additional methods.

The `CoreElement` must call the `pullEventReq` or `pushEventReq` of the `IOElement` whenever it wants to receive an event. On a server `IOElement`, this method has no effect but on a client `IOElement` this method will effectively pull or push the event.

The event insertion into, or extraction from, a server is triggered by the clients. With reference passing `IOElement`, it is the client that calls a method of the server object to insert the given reference into the `CoreElement` or extract a reference from the `CoreElement`. With `TCPIOElement`, each connection on the server `IOElement` side is run by a thread. When a event is to be inserted or extracted, the thread calls the appropriate methods.

Since the `FifoCoreElement` is just a buffer without a running thread, it is necessary that the `IOElement` runs a thread to repeatedly call the `pullEventReq` and `pushEventReq`. Therefore, one must start the `IOElement` thread if the `FifoElement` is a client.

Different derived classes of `IOElement` are currently provided to support different event transfer protocols:

- There is a set of classes to transfer events between components in the same address space. We call these the `RefIOElement` because they share the same memory space and simply need to pass a reference to the event data.
- There is also some variant of the output server to support random distribution of events between clients instead of on demand event distribution.
- Finally, there is a set of classes for TCP event transfer between components in different address spaces on different computers, called the `TCPIOElement` classes.

Components are interconnected by the client server connections. To keep the specification of this interconnection simple, each component is given a unique identifier by the user.

When a component is built, a source or a destination component identifier for events may be defined. When a source or a destination is defined, it means that the component should behave as a client for the input or the output. Otherwise it will behave as a server.

So when a server `IOElement` is bound to a `CoreElement`, it builds a unique name, `<id>_in` for input and `<id>_out` for output and registers it into a place where the clients may get the protocol specific address. The input client will search for the address associated to the `<srcid>_out` name and the output client will look for an address associated to the `<dstid>_in` name.

The association between names and addresses is called the *naming service*. For `RefIOElement` it is a simple map(STL) associating the name string and the pointer to the corresponding `IOElement` server element. For `TCPIOElement` the naming service is a process running on a specific computer to which components may send requests to register an association or get the associated address of a name.

3.2.2 `RefIOElement`

`RefIOElement` instances are used to interconnect components in the same address space.

When a server element is created, it registers its name and address (`this`) association in a map (STL) which is a static variable of the base class.

The client can easily retrieve this address or wait a predefined time (presently 2 seconds) until the other threads have had time to start up and register the server name and address.

The client then calls a `connect` method giving a reference to itself. The server will use it to inform clients of the obsolescence of its own reference if it is destroyed.

`RefIOElement` server objects also have a `getEvent`, `peekEvent` and `putEvent` method.

Once connected the client may get events from the server by calling the server's `getEvent` method. The server will return an event extracted from the remote `CoreElement`. The client then insert the received event into the local `CoreElement`.

This operation is done by the `pullEventReq` method. It can be called explicitly by the `CoreElement` like for instance the `PTCoreElement` or repeatedly called by a thread as it is done for the `FifoCoreElement`. To launch such a thread, one simply has to call the `start` method of the input client element.

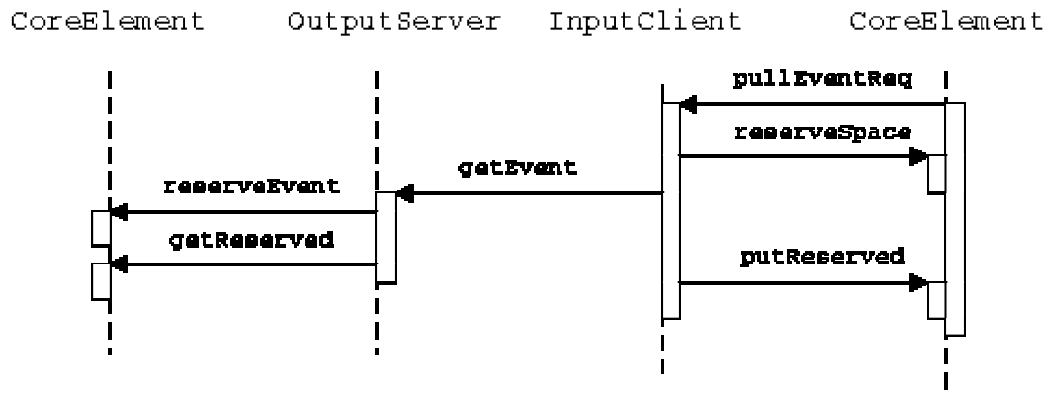

Figure 6: Pull event operation using RefIOElement

The output client has a `pushEventReq` method. Similarly to the input element, it may be called explicitly by the `CoreElement` or called repeatedly by a thread launched by calling the `start` method of the output client element.

The input server's `pullEventReq` method is a blocking call that will return if one of the clients has called the `putEvent` method. The same goes for the output element with the

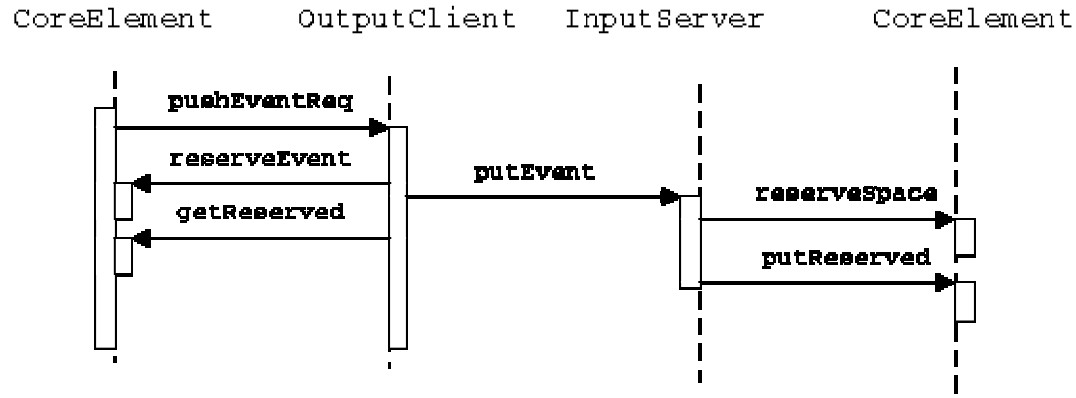

Figure 7: push event operation using RefIOElement

`pushEventReq` method. It will return when one of the clients has called the `getEvent` method.

3.2.3 `TCPIOElement`

To transfer events between components in different address spaces, four `IOElement` classes using the TCP/IP protocol have been implemented.

There are four `TCPIOElement` classes: `InSrvTCPIOElement`, `OutSrvTCPIOElement`, `InCliTCPIOElement`, `OutCliTCPIOElement`.

They behave as the `RefIOElements` as far as the `pullEventReq` and `pushEventReq` methods are concerned. The difference is that events are sent across the network using the TCP/IP protocol.

Each element has its own socket. To connect to a remote server a client needs to obtain its TCP/IP address : host name or IP address and port number.

They all use a naming service accessible through the network. The naming service is described in the next section.

When servers are created, they create a listening socket. With the first call to the `setCoreElement` method, they create a thread that will run the server and process any incoming connection request. They also register their listening socket address to the naming service. The address has the following format: "`tcp:<hostname>:<port>`". For every incoming connection they create a small object run by its own thread to handle the information exchange on the connection. When the connection is closed, the object is destroyed.

Clients will get the address of the remote server using a naming service request and will connect to the server. If the connection is broken, the client asks again the naming service for the server address and tries to reconnect. If it fails, it will retry after having waited for 2 seconds and requested the address again to the naming service.

If the server was stopped and restarted on a different computer, the client will thus automatically reconnect to the server as soon as it registers its new address in the naming service.

If a component crashes one simply needs to restart it. Connections will be restored automatically. To move a component on to a different computer, it is sufficient to kill it and restart it on the new computer. This simplifies a lot the Event Filter management.

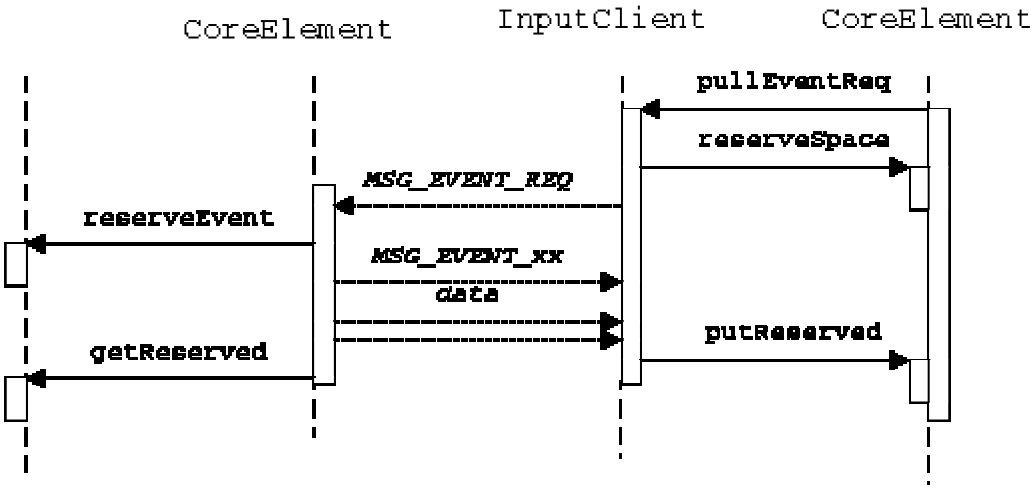

Figure 8: pull event operation using TCPIOElement. If a problem occurs, the connection is closed and the event or space released

The protocol is very simple. Requests and operations are coded in a single byte. The input client will pull events from an output server by sending an event request (MSG_EVENT_REQ = 0x0F) byte. The server will respond by sending back the event when there is one. It must return the size, the identifier of the event used for the DGB and the event byte sequence. There are two different control bytes to signal the arrival of the event data to distinguish between little endian and big endian format (MSG_EVENT_BIG = 0x29, MSG_EVENT_LTL = 0x4E). The interest of this is that computers with the same endianness will not have to swap data. This is not true if they would have used network byte order. This does not make a big performance difference, but this information could be useful if the server had to do some smarter event marshalling.

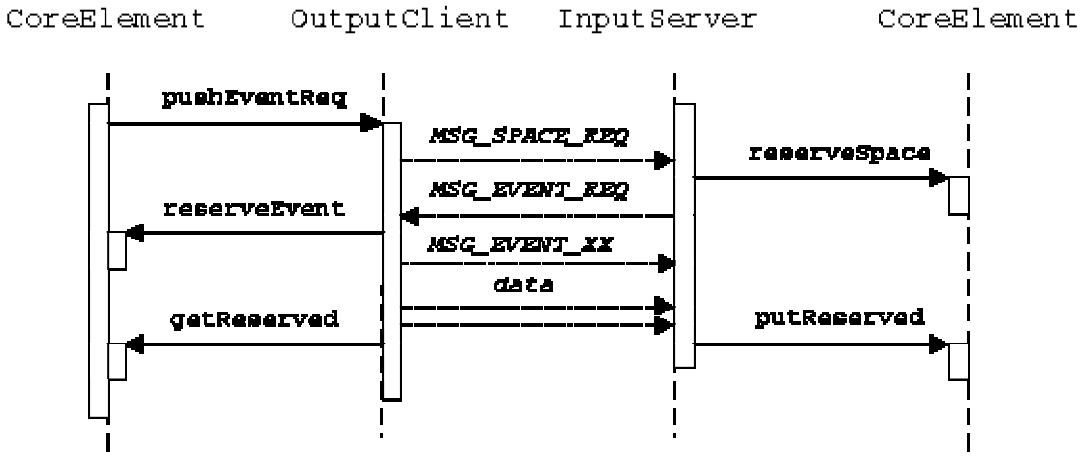

Figure 9: push event operation using TCPIOElement. If a problem occurs, the connection is closed and the event or space is released

When an output client needs to push an event into an input server, it must first ask for space by sending the (MSG_SPACE_REQ = 0x17) control byte. The server then calls the `reserveSpace` method on the `CoreElement`. When it completes, it returns the event request control byte. The client responds by sending back the event.

The event pulling operation of the client is executed in the `pullEventReq` method. The event pushing operation of the client is executed in the `pushEventReq` method.

The server `IOElement` whether it is input or output, is always run by its private thread. On these objects, the `pullEventReq` or `pushEventReq` have no effect. The server TCPIOELement will register their address in the naming service. The address will have the following format : `tcp:<hostname>:<port>` . The Input server will register the name `<id>_in` and the output server will register the name `<id>_out`. Of course, clients do not register any name−address reference.

3.2.4 Naming service

The naming service is a repository for name−address association accessible through the network. Server element may register their name−address association and clients may retrieve the address associated to a given name. The name and the address must be ASCII strings to simplify debugging.

Currently the address and name may not be longer than 1024 bytes.

The naming service does not impose any particular rules for the names. However the adopted convention is that input server has a name obtained by appending "`_in`" to the component identifier. The output server name is obtained by appending "`_out`" to the component identifier. If the component has also a supervisor interface accessible through the network, its name is obtained by appending "`_ctl`" to the component identifier.

In order to simplify protocol and address validity checking, the address must be prefixed with a protocol specific keyword ended by a semi−colon. Ex: "`tcp:?`","`udp:?`","`ior:?`".

The naming service class operations are defined in the `NServiceAbs` class. It is an abstract class. The provided implementation of this class is the `NService` class that uses a map (STL) to associate the name and the address.

NseerviceAbs
+add(String name, String addr):bool +del(String name, String addr):bool +set(String name, String addr):bool +get(String name, String &addr):bool +getNext(String &name, String &addr):bool

Table 2: Naming Service class definition

The `add` method allows to insert a new name−address association in the naming service. If the name already exists, the operation fails.

The `set` method will add or replace an association in the naming service. It is dangerous because it will silently overwrite a valid address if the same name has been given to two different components.

The `get` method will return the address associated to the given name.

The `del` method will delete the name−address association if and only if the name and address perfectly match.

The `getNext` method is used to traverse the list of components. When called with an empty name string, it returns the first name−address association of the Naming Service. It fails if called with the last name of the naming service content. It will return the next name−address association for a given name.

A `NSUDPClient` class and a `NSUDPServer` class are also provided. The `NSUDPClient` class is derived from the `NServiceAbs` class and send requests using UDP datagrams to the `NSUDPServer`. The `NSUDPServer` inherits the `NService` and process the UDP datagram requests.

The protocol used for the UDP interaction with the naming service exchange ASCII strings. The ASCII string is a sequence of values separated by a comma. The first value is always 0. The second value is a request id number. This is used by the client to make sure the answer is a response to it's request. The third value is a command word: ADD, DEL,SET,GET or GETNEXT. The fourth value is the name and the optional fifth value is the address. Only the `add`, `del` and `set` operation require an address value.

The response is the same format as the request, ASCII string of values separated by comas. The first value is always 0 and the second value is the request id number. The third value is a status value. It may be `Ok` or `Ko`. If it is `Ko`, there might be an explanation message as fourth value. Otherwise the next values depend on the operation. The `add`, `del` and `set` operation has no additional values. The `get` operation returns the address and the `getNext` operation returns the name and the address.

3.3 Control element

The purpose of control element is to provide an interface with the EF Supervisor. A wrapper template class may encapsulate the communication protocol.

3.3.1 General design

The `CtlElement` is used to interact with the `CoreElement` from the outside world. It may be used to get the component parameters, change them, or get statistics about the component behavior.
The `CtlElement` uses the pointer to the `CoreElement` to call the appropriate methods or access the appropriate variables of the `CoreElement`.

If the `CoreElement` derived class has additional parameters, like the `PTElement` for instance, one must implement also a corresponding `CtlElement` derived class. The `PTCtlElement` for instance. This derived class must override the `setOneParam` and `getParams` methods and calls the corresponding base class method. These methods return a `String` object.

A `UDPCtlServer` class allows to manage multiple `CtlElement` in the same process so that there is a unique listening port and thread to process supervisor requests.

3.3.2 CtlElement implementation

The `CtlElement` class has four public methods: a `getParams` and `getDeltas` returning a String, and a `setParams(String)`and `setOneParam( String var, String val)` method returning a `bool`.

CtlElement
+getParams() : String +setParams(String params) : bool +setOneParam(String param, String val): bool +getDeltas() : String

Table 3: CtlElement class

The `getParams` method returns a string composed of a sequence of variable−value pairs separated by a comma. Each pair is made of a variable name followed by the = character and a value. . I.e. `id=fifo,src=,dst=,?` Of course, the value cannot contain a comma character. The `setParams` method uses the same input string format, and it should only contain the variable, value pair we want to modify. The order is not important.

The `getDeltas` method returns a set of statistic values: the rejection percentage between two method calls (float); the CPU usage percentage (float); the throughput (number of events per second: float); the total number of events that entered in the component (int); the component occupancy percentage between two requests (float); the number of milliseconds elapsed since the last call of this method.
Because a portable way to compute the CPU usage percentage has not yet been found, this value is presently set to 0.

3.3.3TCP/IP implementation

To support control from a remote host, there is a template class `UDPCtl<CtlElement>` wrapping all the communication aspects. It uses the UDP protocol. To use a different protocol one simply needs to define a different template wrapper class.

The message format is the same as for the UDP Naming Service interaction except that the first value has a non zero long integer value. This value is used as a key to check that the request was effectively destined to that component because a confusion is possible if the component is destroyed and the UDP port is re−used by a new component.

So `UDPCtl` messages are ASCII strings made of values separated by comas. The first value is the key, the second value is the request id and the third value is the command. The commands implemented now are PING, GETDELTAS, KILL, SETPARAMS and GETPARAMS.

If the key or the request value does not match, the `UDPCtl` ignores the request. Otherwise, the `CtlElement` responds with the same key and request value. The third value is the status, `Ok` or `Ko`. If `Ko` is returned, an explanation message may follow as fourth value.

The PING command will return `Ok` if the key matches and the element is running. Otherwise there is no response to that request.

The GETDELTAS, GETPARAMS, SETPARAMS commands will return the result of the corresponding call to the `CtlElement`.

The `UDPCtl` will start a thread to process supervisor requests asynchronously. The name is registered in the naming service in the form `<id>_ctl` and the address `udp:<hostname>:<port>:<key>`. The port is a `short` value and the key is a `long integer` value.

A `UDPCtlServer` class allows to manage multiple `CtlElement` in the same process so that there is a unique listening port and thread to process supervisor requests. For a `CtlElement` to be managed by a `UDPCtlServer` one simply has to register the `CtlElement` in the `UDPCtlServer`. The `UDPCtlElement` is run by a thread to process supervisor requests asynchronously. The user must start this thread by calling the `start` method when all the components of the process are ready to be managed by the supervisor.

3.4 Binding elements together

This section explains how the different elements which have been described above are bound together to form an operational component.

The binding must be bi−directional so that the `CoreElement` has access to the `IOElement` and `CtlElement` and the converse.

The `IOElement` and `CtlElement` may not be operational until they are connected to a `CoreElement` because they need to know the component name for instance.

To keep this operation simple for the user, the following strategy is used. The user construct the `CoreElement`. She then creates the appropriate `CtlElement` and the `IOElement`. At this moment they are not yet connected and they are not operational.

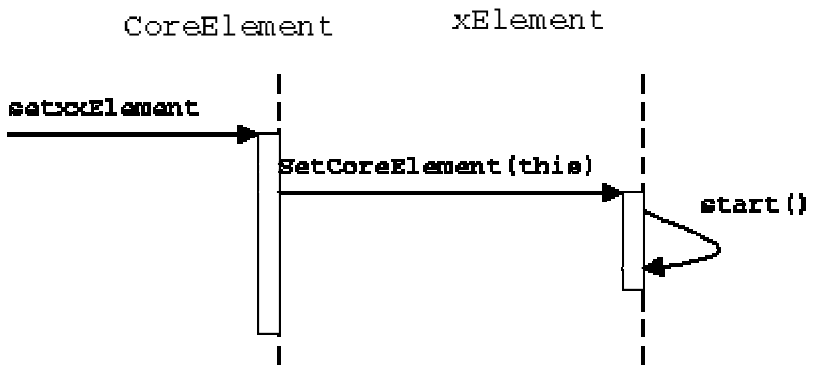

Figure 10: To attach an element to the CoreElement one call the setxxElement method with a point on the element to attach as argument. This method will call the setCoreElement method of the object with a point to it self as argument. The element may then eventually call the start method resulting in starting a thread executing the run virtual method.

Then the user simply calls the `setCtlElement` method of the `CoreElement` which passes the reference to the `CtlElement`. The effect of this method is to store the reference in the `CoreElement` instance and to call the `setCoreElement` method of the `CtlElement` by passing a reference to itself. The `CtlElement` stores the reference in its instance but takes all the necessary actions to make itself operational. It then may get the component name, start a thread, register to a naming server or whatever.

Binding and activating `IOElement` is done in the same way. One simply needs to call the `setInputElement` or `setOutputElement` method to bind elements together and trigger activation of the objects.

The `setCoreElement` method of the `IOElement` and `CtlElement` must take care of multiple call to this function. For instance the first time they may start a thread and the next time they simply want to change the reference.

4. Event implementation

The test event currently used in the dataflow is very simple. It is an array of bytes. It is sufficient to know is its size and the sub event filter event identifier. This number is used internally in the event filter to identify events when interacting with the distributer global buffer (DGB).

This event is implemented in a simple class described below. There is also an internal `Ptr` class using reference counting to manage event object deletion.

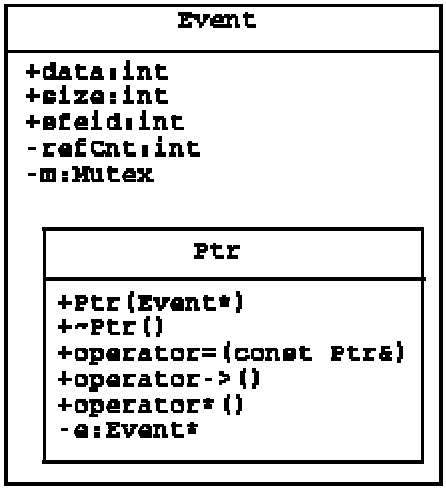

Table 4: the Event class definition with its reference counting smart pointer

Thus a reference on an event must be of type `Event::Ptr` so the reference counting is properly updated. The reference counting is thread safe by use of a mutex.

5. Helper classes

To simplify the implementation of the previously described classes, a set of helper classes have been defined. They are described in the following paragraphs.

5.1 Args

The `Args` class simplifies the access to the program arguments by using the STL routines. A conversion from ASCII C strings into `String` object is performed.

Arguments comprise an argument key word followed by an optional value. One may test for the presence of a given argument key word, i.e. "−dgb" or get the value associated to a given argument. I.e. "−src <String>".A default value can also be provided to return when the argument is not present.

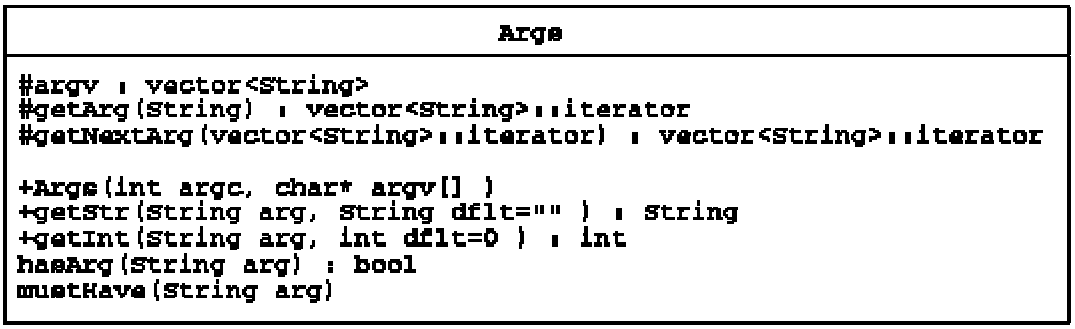

Table 5: the Args class definition to access program arguments

5.2 Socket

The Socket classes are used to simplify access the TCP/IP services. This class is totally different form the class used in Version 3. It now uses a strategy equivalent to the one found in Java.

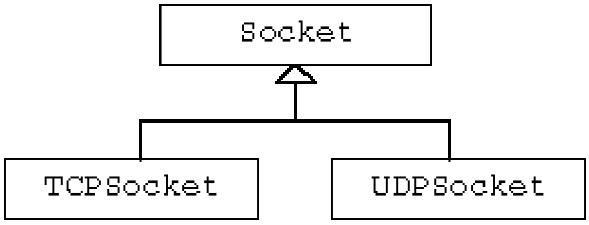

Figure 11: Socket class inheritance tree

Socket
#e:int #Socket() +getAddr(String) : int +getHost(int) : String +isReadReady(int ms=-11) : bool +isWriteReady (int ms=-1) : bool +get Local(String& host, short& port) +isValid() : bool

Table 6: the Socket base class. Since the constructor is protected, only a derived class can be implemented

The `getHost` and `getAddr` methods are static methods. The `isReadReady` method will block until there is some data to read on the socket or the waiting time expressed in milliseconds expired. −1 means infinite wait. The `isWriteReady` works the same way but for writing. The `getLocal` method returns the local host name and port number of the socket.

TCPSocket
+TCPSocket(short=0) +connect(String host, short port) : TCPSocket* +accept() : TCPSocket* +getRemote(String& host, short& port) +send(void* buf, int len) : int +recv(void* buf, int len) : int

Table 7: the TCPSocket class for connection oriented data transfer

To start a socket accepting connections, it is sufficient to create a `TCPSocket` object providing the port number to be used. If it is zero, the system will pick a free port number. The `getLocal` method of the `Socket` base class will return the port number. This creates the listening socket.

The `accept` method will block until another socket tries to connect to the listening socket. It will then return a pointer on the `TCPSocket` object created to process that connection.

To create a `TCPSocket` that connects to a listening socket, it is sufficient to call the static `connect` method providing the host name and port number where to find the listening socket. The static `connect` method returns a pointer on a `TCPSocket` object or NULL if it fails. The `send` and `recv` methods can then be used to exchange data with the remote socket. These methods return a value <= 0 if the connection is closed by the remote socket or if an error occurred. Otherwise it returns the number of bytes effectively sent or received.

The `getRemote` method returns the host name and port number of the remote socket.

UDPSocket
+UDPSocket(short=0) +send(int addr, short port, void* buf, int len) : int +recv(int& addr, short& port, void* buf, int len) : int

Table 8: the UDPSocket class for datagram data transfers

The `UDPSocket` class is simpler to use than the `TCPSocket` since there is no need to establish a connection before sending or receiving a message. A `UDPSocket` may be used for receiving or sending UDP datagrams. The socket is set up to support broadcasts.

If a specific port number is provided when creating the UDPSocket, the system will try to use that socket. If it is already use the object creation fails. Use the `isValid` method to check if it succeeded. In future versions, the Socket classes may throw exceptions. If no port number or a value of zero is given, the system picks a port number that is free. One can determine the port number by using the `getLocal` method of the Socket base class.

5.3 Thread

The `Thread` class uses the same strategy as Java to make it simple to create threads and define their operations.

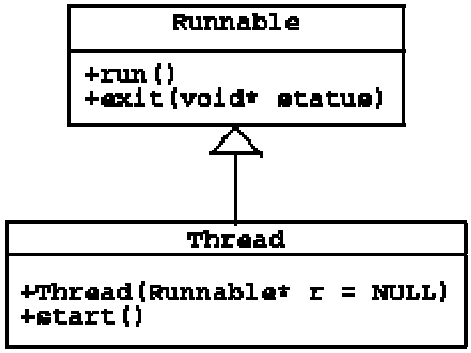

Figure 12: Thread class inheritance tree

The `Thread` class inherits from a `Runnable` class. The `Runnable` class has a virtual `run` method that will be executed by the newly created thread. The `Thread` class may receive a pointer on a `Runnable` object. A call to the `start` method will result in a thread creation executing the `run` method of the `Runnable` object. If no `Runnable` object is passed to the `Thread` object, the `start` method will execute the `Thread`'s run method.

Thus to create an object run by its own thread, it is sufficient to inherit from the Thread class and call `start` when the thread must be started. The `run` method is executed in the object instance scope, so all instance variables are accessible.

5.4 Mutex, Monitor and Semaphore

In order to synchronise the threads, critical sections using mutexes must be used. This functionality is supported by the `Mutex` and `Monitor` classes.
A mutex is created by simply creating an object of type `Mutex`. One may call the `in` or `out` methods to enter or leave the critical section. But it is strongly recommended to use rather an object of type `Monitor` to manage the critical section.

On creation the Monitor object will call the `in` method of the `Mutex` object given as argument, and on destruction it will call its `out` method. This ensures that whatever the way a block is exited (return, break, exception, ?), the mutex will be released.

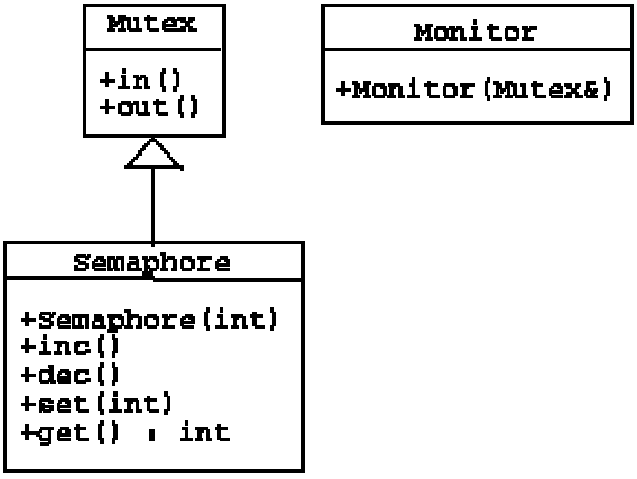

Figure 13: Mutex, Monitor and Semaphore inheritance tree

`A Semaphore` class has also been provided since this is a specific type of thread synchronization needed for the components. It has an internal integer value. The `dec` method will block if the value is zero otherwise it will decrement it. The `inc` method will increment the value and possibly release any blocked thread. One may get the value or change it.

5.5 String

The String class inherits the STL string class and adds a few methods.

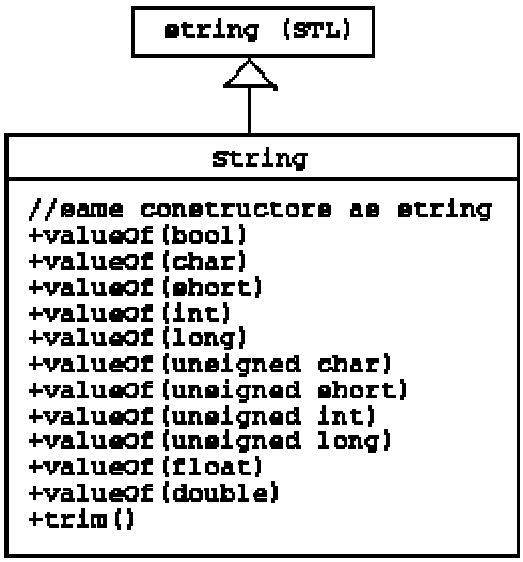

Figure 14: String class extending the STL string class

This `String` class methods are inspired from the Java language. They provide a convenient way to convert values into `String` objects. The `trim` method will remove all the spaces characters at the beginning and the end of the String. A space is the blank char, a new line or a tab.

In a future version, functions to convert a `String` into different base types will be provided.

5.6 StringTokenizer

The `StringTokenizer` class is also inspired of the equivalent Java class. The field concept has been added. A string may be a sequence of fields separated by a character or a string. Concerning tokens, one may specify a default separator in the constructor which can be superseded at call time.

The `nextField` method will return all characters until the separator or the end of file is encountered. The next call will start after the separator. The `peekField` returns the same value but without moving the pointer. The `skipSep` method will skip all separator until a non empty field is found.
The `nextToken` has the same semantics as in Java, and is equivalent to a call to `skipSep` followed by a call to `nextField` using the same separator.

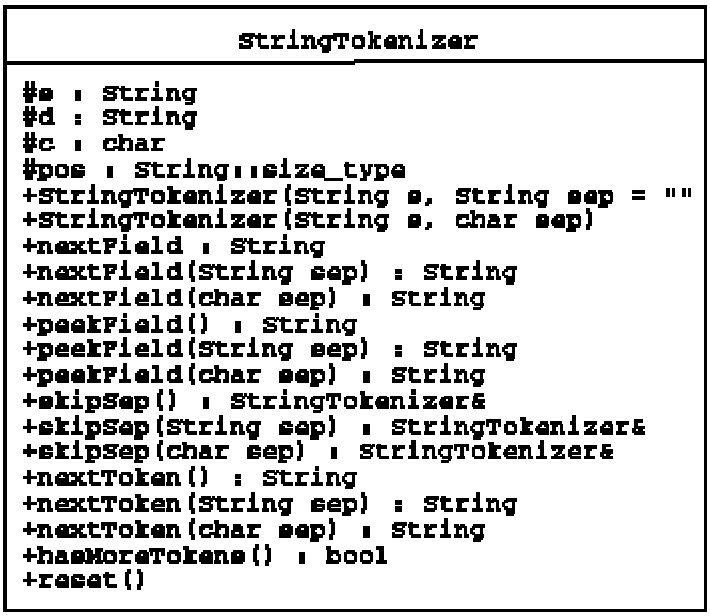

Table 9: StringTokenizer class to parse Strings

The user may use a `String` or a `char` as token separator. It must find an exact match. One may provide a default token separator at object creation or specify one at each `nextToken` or `peekToken` method call. If the `String` separator is the empty string `""`, the rest of the tokenized string is returned.

The StringTokenizer class uses an internal position value in the String. The `hasMoreTokens` method returns false if the pointer has not yet reached the end of the String. The `reset` method will reset the pointer to the beginning of the string.

6. Extending functionality

This section give some hints on how to extend this dataflow software to adapt it to client needs. Three type of possible extensions are covered:

- the definition of customized event transfer protocol thus defining a new set of classes derived of the IOElement base class
- the definition of customized processing task with customized additional parameters
- the implementation of a customized supervision protocol

6.1 Event transfer

The implementation of customized event transfer protocol requires the implementation of the `pullEventReq`, the `pushEventReq` method for the clients IOElement. In the `setCoreElement` method, clients should connect to the remote server object. The IOElement may find the address by accessing the `src` or `dst` variable of the `CoreElement`. Server `IOElement` can register their name using the naming service and get the component identifier by accessing the `id` variable of the `CoreElement`. Or they may use their own naming service which is more appropriate to their needs.

Client or server object may start a thread to process their protocol on their own.

6.2 Event processing

The current implementation requires to override the method `processEvent` in the `PTElement` class. This method receives as argument an `Event::Ptr` object. Currently the return value is ignored and the decision to reject or accept the event is taken on a random basis.

To implement a different behavior, rewriting a customized `PTElement` class using the existing one as an example should be considered.

Adding new arguments requires to derive a customized `CtlElement` class aware of the existence of the new parameters and the protocol used to communicate with the supervisor.

6.3 Supervisor interface

In order to define a new communication protocol for the supervisor, customized `CtlElement` derived classes must be defined. This must be done for all different `CtlElement` classes defined for each `CoreElement` type. Currently there are only two `CtlElements`, one for `FifoElements` and one for the `PTElement`.

If the `CtlElement` requires to be run by its own thread, it may be started inside the `setCoreElement` method.

Since only the UDP control protocol is currently used, this protocol is implemented in the `CtlElement` base class. If new protocols are required, this may be rearranged so that a `UDPCtlElement` class derived from an abstract `CtlElement` class is available.

7. References

[1] PC−based EF supervisor: design and implementation, Z. Qian et al. , Note in preparation

[2] A High Level Design of the Sub−Farm Event Handler, ATLAS DAQ/EF−1 Note 061, http://atddoc.cern.ch/Atlas/Notes/061/Note061−1.html