The Atlas Level 2 Reference Software

Reiner Hauser (rhauser@fnal.gov)

March 30, 2000

1 Introduction

The Atlas Level 2 Reference software has been developed as part of the Atlas Second-Level Trigger Pilot Project [1]. Its goal was to have a single common source code base for the various activities in the Pilot Project to avoid unnecessary duplication of effort.

The most important requirements were independence from the operating system and a modular design, which would allow to modify and optimize the software for different purposes. This includes the network technology studies in the various hardware testbeds, but also the physics algorithm developement and benchmarks.

To make it easier for interested institutes to take part in the developement, the software should run on commodity PCs with off-the-shelf operating systems (including Windows NT and Linux). The software itself does not depend on any other packages or infrastructure (although e.g. AFS is required to access the code repository). Emulators are available for each component in the final system which might be implemented in special hardware (like the supervisor and the ROBs). An emulator can be replaced by the real hardware if it is available.

A high-level view of the overall data flow is shown in Fig. 1. The various parts of the system can be executed in one or multiple processes.

The following scenarios were foreseen for the reference software:

- Single feature extraction algorithm. A user should be able to write an isolated fex algorithm and run it in a simple framework over a sequence of events, producing output in any format he likes. The algorithms can be used without change in a full system, if they are written according to some simple rules.
- Single node trigger processing. This mode allows a full menu based execution of multiple fex algorithms in a common framework. The data is read from a file and again the output is user defined. This mode is a full functional test of the trigger algorithms and the steering code.
- Multi node system with skeleton applications. This scenario implements the full data flow of the trigger system, but without executing any algorithms and using only dummy data. The main application is in network technology studies. Simplified versions of all major components (Supervisor, Steering, Rob) are available.
- Multi node system with algorithms. In this mode the system is running with emulators for both the supervisor and the ROBs (or the real hardware, if available) and the steering processors are executing the menu guided Level 2 algorithms. This corresponds as much to the 'real' system as can be achieved on a testbed with restricted size.

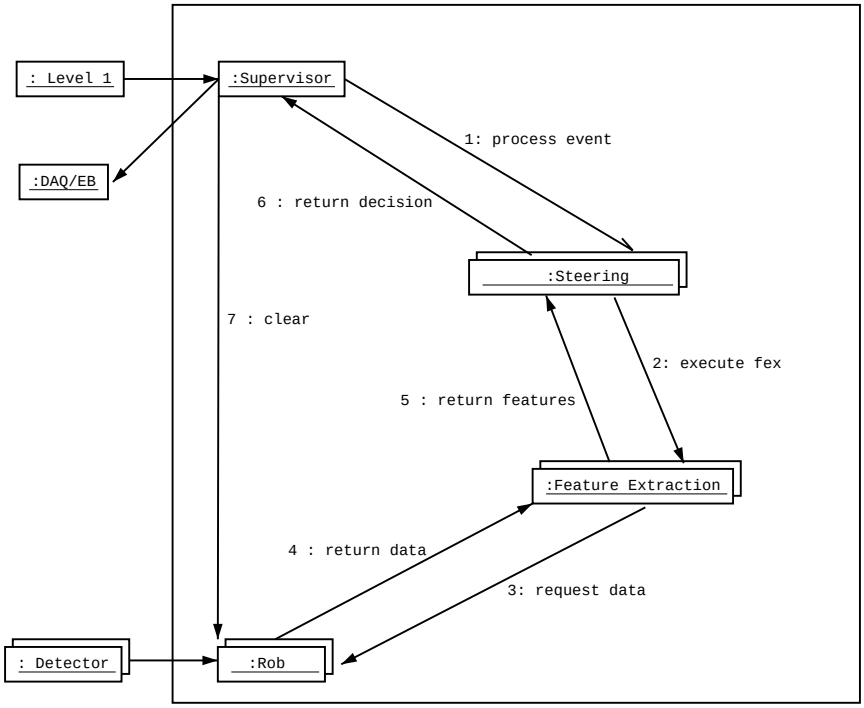

Figure 1: Overall System Data Flow

After about 18 months of developement (including requirements collection and design), the reference software consists of about 100k lines of C++ code. About 20 developers from 10 institutes contributed to the code, with many others taking part in the discussions.

2 Overall Structure

Managing a large piece of software, which is developed by many people who are geographically distributed, requires that the software itself is properly structured. It is essential than programmers can work in parallel on different parts and extend it in the future to accomodate new needs.

The LVL2 Reference Software is *modular* in the sense that it is divided into many *packages*. Each package corresponds to a well-defined domain and there is usually one person responsible for a package. People can work in parallel, e.g. on two different algorithms for two different subdetectors without major interference.

In many cases there are dependencies between packages, which make it necessary that some parts are available before others can be written. This is reflected in the *layering* of the Reference Software, where lower level packages were available first and are supposed to be used by all other packages. Examples are the operating system layer and the configuration or the error reporting package. It seemed desirable to define this basic functionality first, to make sure that there is a single way to perform a given task.

Usually lower level libraries can be defined in sufficient detail to implement them completely. This is often not possible for the higher layers. Here the system is supposed to be *open* in the sense that different people may want to change parts of the software to adapt it for their needs or to optimize it for a certain hardware technology.

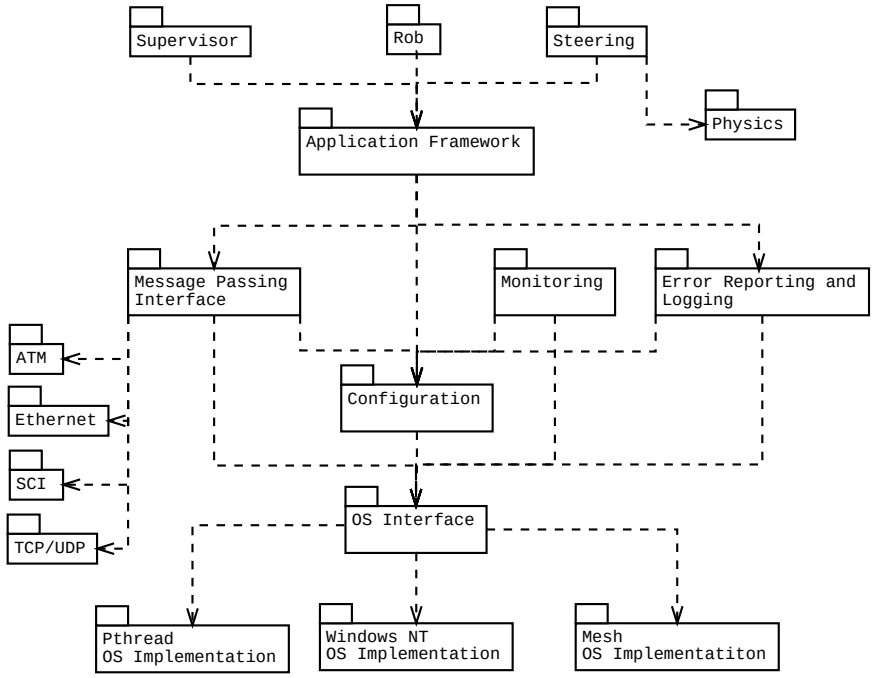

Figure 2: Layered structure of Reference Software

Using an object oriented design and language makes it possible to define *interfaces* for various domains, which can then be implemented in different ways. There are two major areas where interfaces have been been used in this way in the Reference Software: the message passing layer and the application framework.

The *message passing layer* is responsible for transporting a message from one node in the system to another. Different testbeds implemented this interface using their preferred network technology, e.g. ATM, Fast/Gigabit Ethernet, SCI, MPI. To make the system usable without any special hardware, a default implementation using either UDP or TCP was provided as well.

The *application framework* defines interfaces for the 'high level' objects, corresponding to the supervisor, ROB and steering processors. Again there are various implementations for each interface, ranging from very simple *skeleton* applications to emulators and systems using the 'real' hardware.

The other important task of the application framework is to achieve *transparency* regarding the distributed nature of the system for the applications themselves. E.g. the steering processing code is not aware of the fact that it may run in either a single-node or a multi-node system. In the latter version the data is provided by ROBs via the network instead of coming directly from a file. Furthermore the framework is able to run multiple copies of each application in a *multi-threaded* mode and to distribute the workload among them. This is important both for hiding the latency between the time events are requested from a ROB and the data is finally returned, and for making efficient use of multiprocessor systems.

3 Basic Libraries

The packages described in the following sections are the *basic* libaries in the sense that they are directly or indirectly used by all other packages. In many cases the

need for a certain package was based on the idea that there should be only one way to do a certain task. This means, that every time a programmer needs e.g. access to configuration information he should use the `Configuration` package instead of inventing his own file format. If he wants to output debug information or issue an error message, he should use the `ERL` (Error Reporting and Logging) package etc.

3.1 Operating System

The operating system package is supposed to shield the rest of the application from any OS specific issues. It provides interfaces for all services we expect to need from the underlying OS. This does not include interfaces which are already defined to be OS independent in various other standards, e.g. input/output operations.

By choosing both Windows NT and Linux from the outset, we hoped to make sure we could catch most non-portable parts of our code on the first try. This proved to be mostly correct and the OS dependent part has been ported to other in-house developed thread libraries in the meantime.

The non-portable aspects which were needed in the Reference Software were identified to be the *thread* interface together with its associated synchronization mechanisms, and an API to access the server and client parts of the TCP/IP protocol suite. In both cases the interface was defined as a set of C++ classes.

For the thread interface we tried to stay as close as possible to the POSIX thread interface. Parts which were missing in either operating system (e.g. Windows NT has no condition variables) were implemented in terms of other primitives. However, our interface does not claim strict conformance to POSIX semantics. In most cases we give even less guarantees than either POSIX or Windows NT. This avoids spending a lot of effort in trying to get either one or the other of the underlying paradigms to be emulated completely on the other system.

In all cases we defined only the basic functionality. E.g there are no recursive mutexes or read-only locks as can be found in newer X/Open Unix standards ([2]).

The thread interface defines the following basic objects: `Thread`, `Mutex`, `Semaphore` and `Condition`. Users can inherit from `Thread` and override the `execute()` method. This is all that is required to define a new thread of control from the user's point of view.

The implementation takes great care not to make any implementation details visible to the rest of the software. Apart from avoiding global namespace pollution, this allows to simply exchange the underlying shared library to switch to a different implementation (see Fig. 3.1)

The thread interface has been implemented on top of the following systems/libraries:

POSIX Thread (Pthread) This is the standard thread interface in the Unix world. It is available on all modern Unix versions.

Windows NT Microsoft's thread interface is available on NT only and different from anything else.

MESH A non-preemptive thread and communication library developed as part of the Ethernet testbed activities.

GNU Pth A non-preemptive and portable user-level thread implementation from the GNU project.

The TCP/IP interface is implemented using the BSD `socket` interface. This is available on every Unix system and in a slightly modified form on Windows in the form of the `winsock` library. The C++ implementation defines classes for client and server connection endpoints and hides the small differences between Unix and

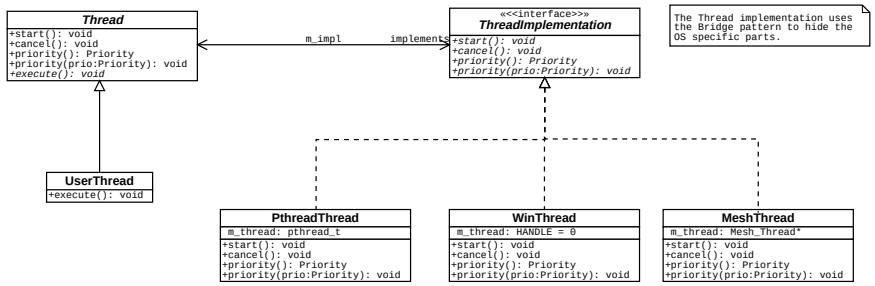

Figure 3: Thread implementation using Bridge pattern

NT. While the original socket interface is supporting multiple protocols, our version is simply providing access to TCP and UDP. The main goal here was to make it simple to write an Internet based server or client application.

The TCP/IP interface is used in two different ways: as communication channel for various run control and monitoring related tasks, and as an implementation of the *message passing interface*.

3.2 Configuration

The configuration package provides access to read-only configuration information. It seemed preferrable to have a single way of access than defining multiple incompatible configuration file formats for the different parts of the software.

To achieve this goal, the configuration mechanism must be flexible enough to accomodate for all types of information. It allows you to define integers, real numbers, strings and arrays of these objects and access them easily from inside the application.

Every configuration item has a *name* and a *value*. The value can be any of the types mentioned before. The *name* is used to uniquely identify each value. Every programmer can define his own set of parametes. To avoid name clashes the name space is hierachical: name/value pairs can be nested into so-called *contexts*. A name has to be unique only inside of its immediately surrounding context. In addition a value can be a *reference* to another parameter in the configuration file or to an environment variable.

The format is closely modelled after C-like initialization statements. The following example shows the format without going into the details:

```
# everything after the hash sign is a comment

AnInteger = 4;

AReal = 3.141;

AString = "Hello world";

AnArray      = [ 3, 1, 10 ];

# arrays can be nested
AnotherArray = [ 10, [ 1, 2, 3 ], 20, 30 ];

AContext = {
  # this is different from the parameter above
```

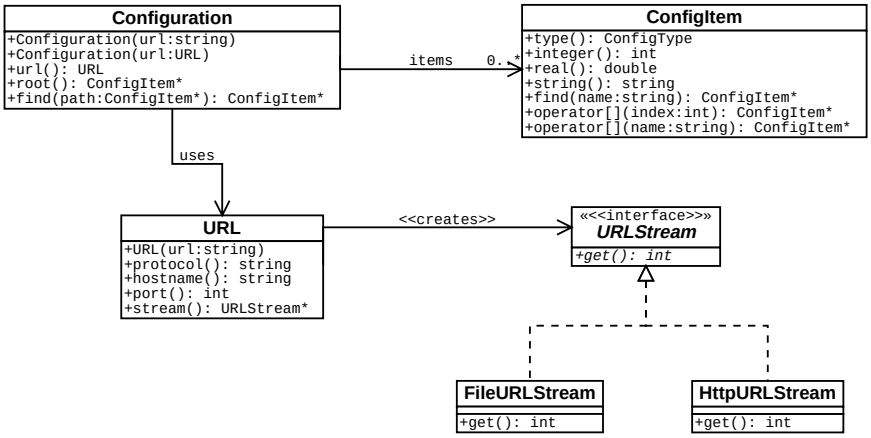

Figure 4: Configuration classes

```
  AnInteger = 5;

  # contexts can be nested, too
  NestedContext = {
     AnInteger = 6;
  };
};

AReference = $AContext/AnInteger;
```

From the point of view of the program, all configuration information seems to come from a single file. Since many different people have to contribute these parameters, there is an `include` facility to make these task more managable. Parameters can be defined in many different physical files, which are then pulled together by a master configuration file via include statements:

```
include "network.conf"
include "menu.conf"
```

Configuration files are specified via URLs. Both the `file` and the `http` protocol are supported. Names in include statements can be URLs again, or they are interpreted as a *relative* URL, just as in a web browser. This allows you to make the configuration information available either via a common file system or a web server. The location of the configuration file is defined in an environment variable `T2CONFIG`.

Inside the application the parameters can be accessed by their full name. While more elaborated access mechanisms are available, they are not used by most applications.

The way of accessing the data is independent from the underlying format as ASCII data files. That format can be changed in the future, e.g. to some kind of database. However, ASCII files are easy to understand and can be modified with a simple text editor.

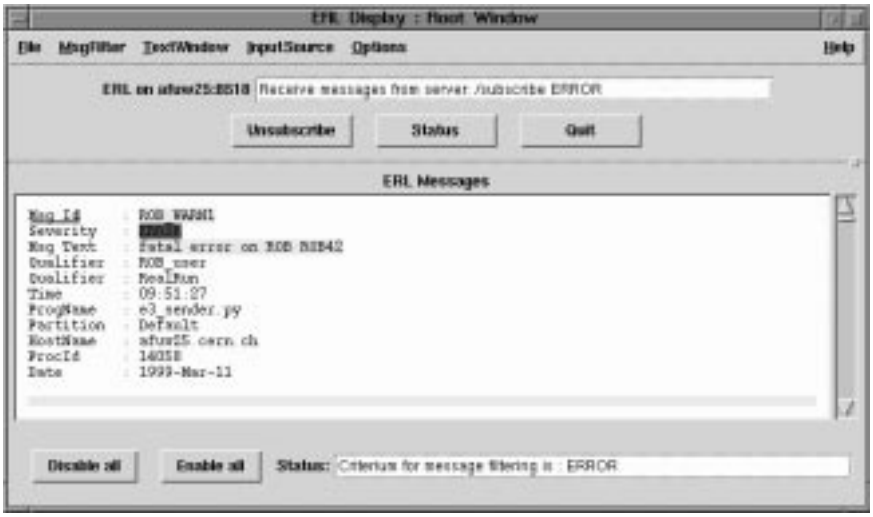

Figure 5: ERL Graphical Display

3.3 Error Reporting and Logging

The ERL (Error Reporting and Logging) package provides a way to issue status and error messages from inside the application. By using this mechanism, the user doesn't have to care if the application is running as a single program (in which case the output goes to the screen) or as part of a distributed system (in which case the output of all programs is collected at a central place). ERL is based on a client/server architecture. The ERL server is responsible for receiving messages from the various ERL senders (i.e. the applications) and forwarding them to multiple ERL receivers. In addition the ERL server can keep track of a global log file and implement several filtering options for the receivers, who might be only interested in part of the information provided.

An overview of the architecture can be seen in Fig. xx.

From the programmer's point of view, all one has to do is to declare an object of type `ERLStreamOut` and output error messages to it. Messages are distinguished by their *severity*, which can be one of `Fatal, Error, Warning, Info, Debug`. In addition they can contain arbitray text and multiple parameters.

The application uses the ERL library to communicate with the ERL server. Alternatively, it can run without a server and output everything to a local logfile. The ERL server receives messages from all applications and forwards them to all interested receivers. An ERL receiver is simply another application which is interested in the messages. Examples are a graphical display, which allows the user to monitor error messages based on some criteria (see Fig. 5)

The run control system of the reference software uses ERL messages to inform anybody who is interested about changes in the internal state of the application. These messages are intercepted by a run control application and used to update its internal information about the current system state.

Most programmers never use the ERL interface directly. Instead they use a set of convenience macros defined mostly for debugging purposes. The macros will only output an ERL message, if the current global debug level of the program is high enough. This allows the programmer to augment his code with debug macros corresponding to different severity levels, but switch them off at run time when they are no longer needed.

3.4 Monitoring

The monitoring package provides run-time access to data which is explicitly made available by the application. This is typically statistics information about various internal values, but it is also used in the run control system to get access to the current 'run state' of a node.

From the programmer's point of view, this package provides an easy interface to make any internal data accessible to the outside. The programmer only has to inherit from a base clase `Resource` and implement the `get_resource()` method. In addition it is possible to provide a `reset()` method which is supposed to 'clear' the values in the resource.

Finally, the `set()` method can be used to change the resource to a new value. However, it is completely up to the programmer as to how he wants to interpret the new values. The run control system makes use of this property and uses the `set()` method to issue commands to the application.

The programmer has no restriction as to what the resource represents. Typical examples are counters and histograms.

In contrast to the ERL package, which sends error messages to the server as soon as they are available, the monitoring package is based on a 'pull' paradigm, i.e. the data has to be explicitly requested from every node. This makes sure that the package will not introduce any unwanted overhead if the features are not used. Counters and histograms are updated only locally with no network traffic. Accessing the data is under user control and happens only very infrequently.

The monitoring package uses UDP instead of TCP, again to avoid the overhead of a permanent TCP connection. Furthermore this avoids the need of a well-known 'server' somewhere in the system. Any client who wants information from a node can simply send a request and ask for it.

Applications making use of the monitoring protocol are a graphical resource display written in Java and the run control system. A command line based program is also available to access an arbitrary named resource in the system.

4 Message Passing

The message passing layer provides the basic mechanism of data flow between the nodes in the system. By data flow we mean all exchanges involving real event data as well as the request and reply control messages. Note that this is different from the communication with the error reporting, monitoring and run control subsystems.

The message passing layer has its own interface, since this is the place where different network technologies want to provide their own optimized implementations. However, some parts are shared by all implementations, like the byte-swapping code needed in a heterogenous environment.

The interface defines the following abstract objects:

Buffer Represents a piece of memory which can be sent or received from the network. The underlying memory area doesn't have to be contigouneous. Access functions are defined to read from or write into such an area. Byte-swapping is handled automatically, if the pre-defined network byte order differs from the host byte order. The main reason for a network specific implementation of the buffer interface is that the underlying technology may require the area to be locked in memory, so that physical addresses are immediately available. In the reference software a *message* is always completely contained in a single buffer.

Node Represents one machine in the system, although in some cases one can run more than one 'logical' node on the same hardware, if the network technology

allows it. Each node is uniquely identified by a numerical 'node ID'. By convention each node also has a 'type'. In the case of the trigger software this corresponds to one of *Supervisor*, *Steering* or *Rob*. Nodes have both a send and receive method, the first taking a buffer as an argument and the latter returning a buffer as result. These two operations are abstract and implemented for each network technology. A node can therefore be considered to be the basic communication endpoint in the system.

Group Represents a group of nodes. Calling the `send` method of a group will send the buffer to all nodes in the group. Calling the `receive` method will return the next received buffer from any of the nodes in the group. There are two types of groups: dynamic and static. A dynamic group is one to which nodes can be added to or removed from at run-time. They are mainly used as a convenient way to send the same message to a number of nodes. Static groups are set up at initialization time and never changed. They correspond to a fixed set of nodes over the whole life-time of the program. Examples are e.g. all ROBs. If the underlying network supports multicast operations, it will typically implement the send operation of static groups with that optimization.

NodeFactory Since all objects described above are only defined in the form of abstract base classes, there must be a way to instantiate the correct version at run-time. This is done via the factory pattern ([3]). The `NodeFactory` is a singleton object, whose sole instance can be accessed via its static `instance()` method. The object itself provides methods to create buffers, nodes and groups.

In the following we describe the available implementations of the message passing interface.

4.1 UDP/TCP

The UDP and TCP versions of the message passing interface provide a 'portable' implementations which can be used on any machine. They also support more than one logical node on the same physical machine, making it easy to debug a multi-node system on a single PC.

The implementation is based on the OS interface layer and is therefore independent from the underlying operating system. The existing limits are mainly due to limits inherent in the protocols themselves:

- UDP has a maximum message size of 64 kB. The actual message size used in the implementation can be specified from the configuration file. Message delivery is not guaranteed and has the same characteristics as for UDP datagrams.
- TCP has an unlimited message size. It implements a special subclass of Buffer, which adds an additional 'length' word to each message. This is used on the receiving side to allocate enough buffer space.

None of these protocols impose any other restrictions. E.g. data can be transfered from and to any memory location. Due to the kernel based implementation of the protocols the latency is much higher than for the other implementations. Both Unix and Windows NT will copy the data at least once, making a zero-copy version impossible.

Port numbers are chosen internally according to some algorithm. This avoids that the user has to choose some port number which might possible conflict with

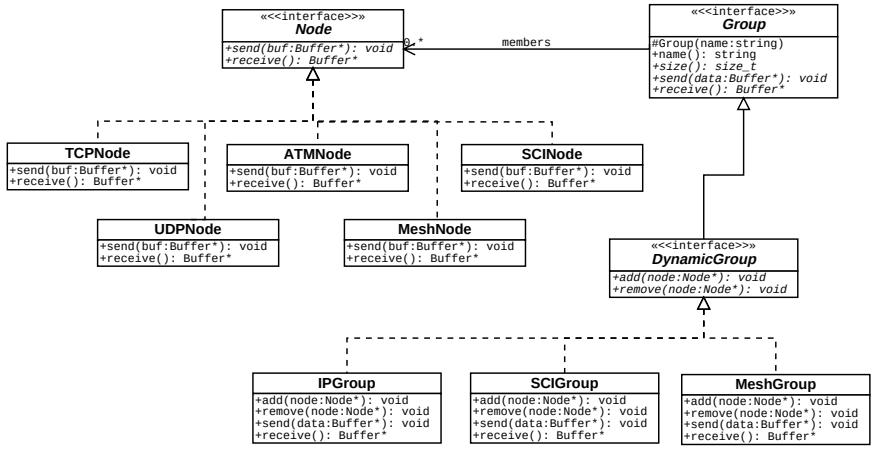

Figure 6: Message Passing classes and their relationship

other subsystems. At the moment there is an upper limit on the maximum number of nodes (100) in the sytem. This is a compile-time parameter which can be easily changed.

4.2 ATM

The ATM [4] version is described in detail in ref Saclay. It exists both in a library and a driver version. It is based on the ATM Adaption Layer 5 (AAL5), allowing a maximum message size of 64 kB. For connection setup PVCs (permanent virtual connections) are used, which requires a 'static' setup but avoids the overhead and complexity of SVCs.

The ATM implementation is available both for Windows NT and Linux as well as LynxOS. It supports multicast operations in hardware and can therefore take advantage of the static group interface. The underlying network buffers are actually implemented as a chained list of buffers, enabling any size message to be sent with little overhead.

4.3 Ethernet

The 'raw' ethernet implementation of the message passing layer is based on a user level library named MESH, which bypasses the kernel completely. It supports both Fast and Gigabit Ethernet and also includes a fast user-level thread switching library (see the information on the OS interface layer). Only Linux is currently supported.

To provide for messages larger than an Ethernet frame (1.5 kB) an additional protocol has been implemented which allows to transmit up to 64 kB as a single 'logical' message. Multicast is supported but has so far not been used in the testbed operations.

4.4 SCI

The SCI [5], [6] implementation uses a commercial driver from Dolphin [7] to setup the interface cards and create connections. Since SCI is used in shared memory mode, there is no more interaction with the driver required afterwards. The message passing interface is implemented as shared memory operations over SCI.

The software is available both for Linux and Windows NT. The maximum message size is a run-time parameter which can be changed in the configuration file

and is only limited by the address translation hardware. No hardware multicast is supported, instead a multicast is implemented as multiple writes to different destinations.

4.5 MPI

The MPI [8] version is running over any implementation of the MPI standard. It has been developed using the MPICH package from Argonne on a normal Unix cluster and then run in production mode on a 96 node system at the University of Paderborn [15]. This cluster implements MPI on top of SCI and is running the Solaris operating system.

5 Application Framework

The application framework is a set of interacting classes which provide a common environment for all 'real' applications. It takes care of tasks which are needed across all the different programs. This includes the setup of the error reporting and monitoring systems, the creation of multiple threads and all input handling and despatching. Furthermore the framework makes it transparent to the applications if they are running in a single or a multi-node environment. The latter is achieved by the consequent use of interfaces and so-called proxy objects. In a distributed environment, the proxy objects provide the same interface as a local object to the caller, but actually forward the arguments to a process on a remote machine.

5.1 Interfaces

Basic interfaces have been defined for the supervisor, steering and rob applications. The latency for a single event is typically much longer than the time the supervisor has to handle two consecutive events. Therefore the interaction between the supervisor and the rest of the sytem is of an asynchronous nature. The supervisor will *request* that a certain event is processed and then go on to the next event. When the event is finished the steering will execute a callback routine in the supervisor to inform it about the Level 2 result.

The basic interface of the steering consists therefore of a simple `request()` method with no return value, while the supervisor has an `accept()` method which is called back from the steering.

The interaction between steering and Robs on the other hand is of a synchronous nature. When the steering (or, to be more precise, a feature extraction algorithm) requires data, it will ask the Robs and then wait for the reply. There is nothing it can do in the meantime. It is the responsibity of the framework to make sure that another thread will be made active at that point, typically working on a different event. When the data has arrived, the original thread will be woken up and continue where it was suspended. The interface in the Rob consists therefore of a single `request()` method with the Rob data as return value.

The supervisor has to call the Robs, when a certain number of events have been accepted or rejected, so that the Robs can clear their memory buffers. Since this is a broadcast like operation with no feedback to the supervisor, it is represented as a simple method call with no return value.

All implementations need a way to be informed about the other objects in the systems, be it the 'real' ones or the corresponding proxies. Therefore each interface has one or more `inform()` methods which are typically called only once at setup time.

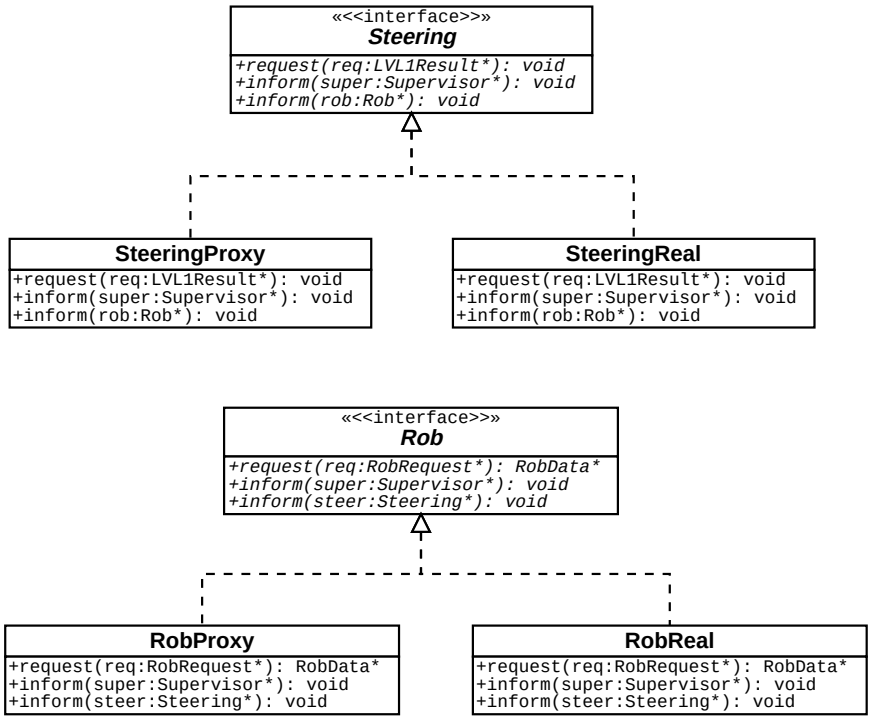

Figure 7: Interfaces for the major components

The class diagrams can be seen in Fig. 5.1. The typical sequence of calls can be seen in the interaction diagram Fig. 8

5.2 Proxies

Since the major parts of the reference software communicate only via their public interfaces with each other, it is possible to have multiple implementations for each. E.g. the 'Steering' can be implemented using all the physics algorithms or just a simple *skeleton* version.

The Proxy design pattern [3] takes this idea a step further and provides an implementation which actually represents an object on another physical machine. This makes it possible to have distributed system without the participants being aware of it.

Basically a proxy object implements a given interface, but instead of having code for a given method it will use the message passing interface 4 to send all parameters to a remote machine. A reply received will be returned as a normal function return value.

This design pattern is essentially an object-oriented version of the remote procedure calls which are used e.g. for Sun's NFS and other common Unix network services [9]. CORBA [10] is an industry standard for such an architecture, providing in addition interoperability and many common services, e.g. looking up objects by name etc.

CORBA and especially one implementation, ILU [11], were considered at the beginning of the design phase as a potential candidate to be used for the networking part. We finally decided against using them for the following reasons:

- CORBA normally runs over TCP/IP, imposing a very large overhead.

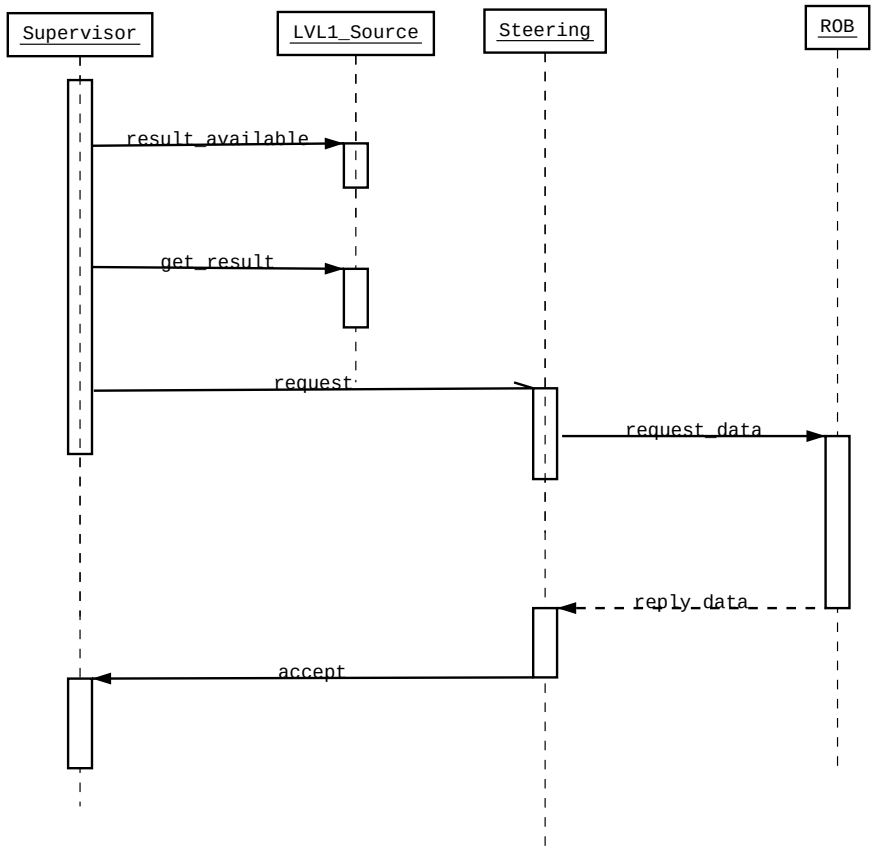

Figure 8: Processing of an Event

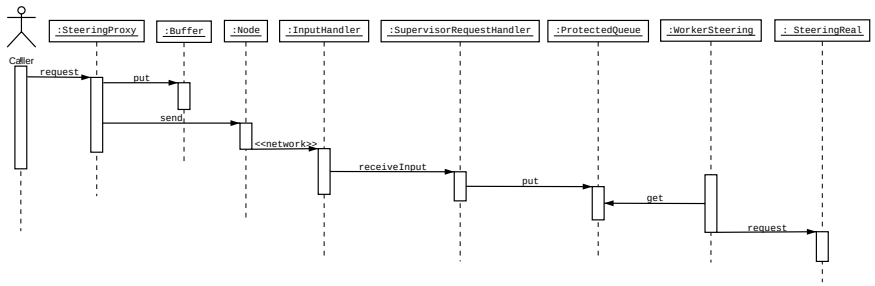

Figure 9: Calling sequence when using proxies

.

- It is a non-trivial task to port the networking layer of an existing CORBA implementation to our message passing interface (or have all network technologies implement the functionality required by CORBA).
- To achieve interoperability, CORBA has rules to define the format and byte-ordering of all exchanged messages. The marshalling and unmarshalling of parameters can be mostly avoided in closed system like the reference software. Again, it was not trivial to bypass that part of an existing CORBA implementation.

However, it still looks attractive to use the CORBA interface definition language (IDL) in the future and maybe generate customized code for the reference software framework. This is made easier with the availability of some useful building blocks like IDL compiler front-ends which have a backend that can be customized by the user.

A typical calling sequence in a distributed system can be seen in Fig. 9 using some of the classes from 5.1.

5.3 Input Thread and Input Handler

All network input is handled by the framework first and then dispatched to an appropriate *handler*. A separate input thread is waiting for incoming data from all other nodes. Clients can register an `InputHandler` object with the input thread. Despatching can be based on one of three criteria:

- the message type. This is used when the same handler is called for all messages of a given type. An example is the handler in the steering processor which receives requests from the supervisor and passes it on to a worker thread.
- transaction ID. Each request message contains a unique transaction identifier. This is used to associate incoming replies with the original requests, e.g. when Rob data is requested.
- finally there is a default handler which catches all other messages. This handler will typically just submit an error message.

5.4 Threads

The framework is responsible for the creation of multiple threads in the system. Examples are the input, the monitoring and the run control threads as well as the so-called worker threads.

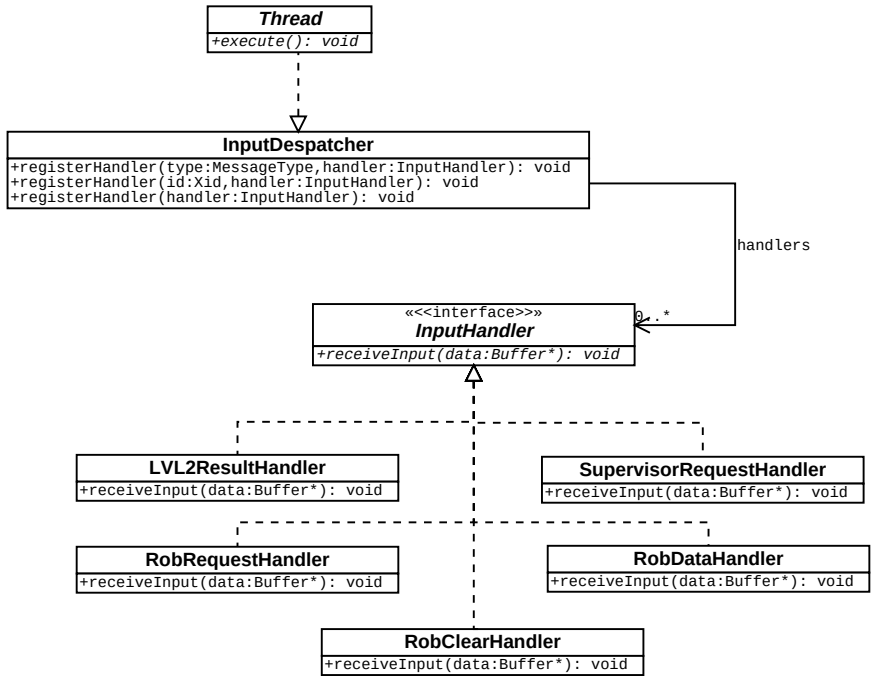

Figure 10: Input despatcher

Multi-threading in the application framework is based on a thread pool model. The configuration file allows to specify the number of worker threads which should be created. All these worker threads execute identical code. Typically this is an endless loop, with their first action being a request for new work to do.

The reference software uses a so-called `ProtectedQueue` object to synchronize the different worker threads. A `ProtectedQueue` is a fixed-size queue with `get()` and `put()` methods. Calling `get()` on an empty queue will suspend the calling thread, as well as a `put()` on a full queue. All worker threads will call the `get()` method of a single queue. Another thread will put new requests into that queue as they arrive. If there are worker threads waiting which are not busy, each `put()` will wake up exactly one thread.

An example is shown in Fig. XX. On receiving a new `LVL1Result` message, the input thread will call the `SupervisorRequestHandler`. This one will in turn put the new request on a queue. The next worker thread to finish its current work will pick it up and start to work on the new event.

The framework assumes that a single event will be handled by one thread only. This simplifies the code for the rest of the processing, since no more synchronization is needed. A thread will still be suspended when it is asking for ROB data and is waiting for the reply. However, from the point of view of the programmer this looks like a synchronous operation. He is not aware of the fact that other threads are running in the meantime.

5.5 Applications

An application is an executable providing a certain functionality in the LVL2 trigger system. Since applications share a lot of code a `T2Application` base class exists, which performs common initialization tasks. It is also the basic interface to the external run control. Applications simply inherit from the base class and provide

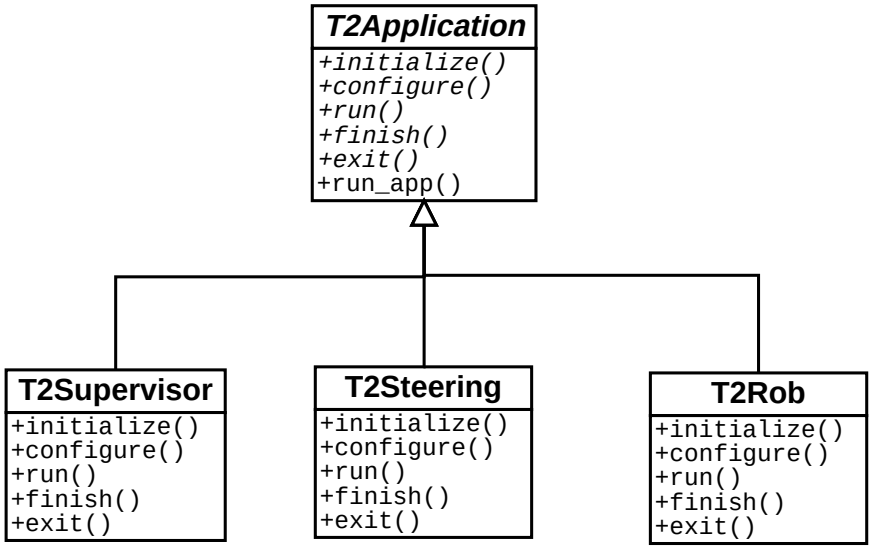

Figure 11: T2Application and subclasses

implementations e.g. of their specific configuration tasks.

The `T2Application` base class initializes the application framework. Depending on information from the configuration file it starts the monitoring and run control threads. It always creates the input thread. The creation of the worker threads is done in the derived class, since it depends on the type of application.

Furthermore the `T2Application` class provides several virtual methods which can be overridden by a subclass. These methods are called by the run control when an external command is given by the user.

`initialize()` this method is called unconditionally at program startup. The application cannot assume that any other nodes in the system are running. Local initialization can be done here.

`configure()` this method is called when all nodes are running. Therefore e.g. the network setup can now be completed. Worker threads are typically started here.

`run()` this method is called to actually run the current application.

`finish()` this method is called when the end of the current run has been reached. Results can be written out at this stage if desired.

`exit()` this method is called before the application is going to exit. Any resources should be freed here.

At every point in time the application is in one of five states. The interaction of the run control and the internal states are shown in Fig. 5.5. Note that the current implementation of most applications is not prepared for the transition from `Finished` to `Configured`, although this would be very desirable for measurement runs with varying parameters.

5.6 Skeletons

The so-called *skeleton* applications are essentially stripped down version of the real programs. They don't include any real algorithms, but are used to test and debug

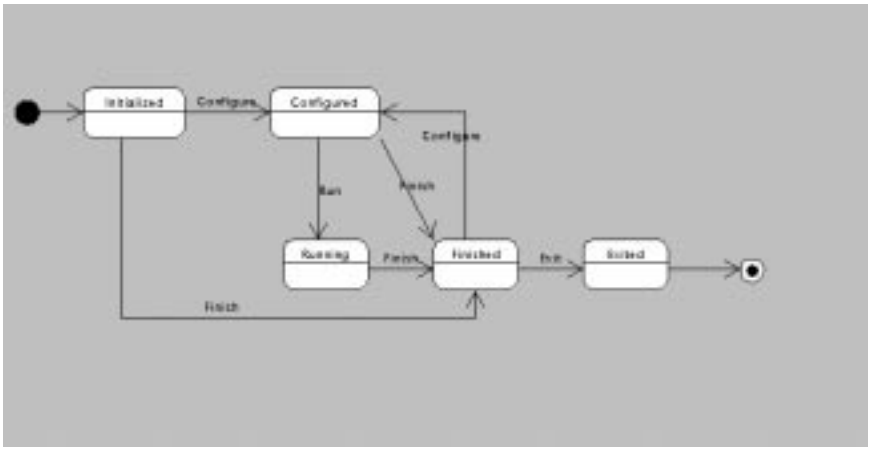

Figure 12: Internal States of T2Application

the application framework itself. They can be also used for basic network performance measurements, since they include the full message exchange like in a real system.

The skeletons can be parameterized to emulate the behaviour of the real applications. E.g. the steering skeleton has parameters for the execution times of the different algorithm steps. At the appropriate point the skeleton will simply burn CPU time by executing dummy floating point operations.

6 Supervisor

The supervisor application is able to emulate the real supervisor by either generating Level 1 ROIs according to a given distribution or reading event data from a file. It can also be connected to the real ROIBuilder hardware, therefore becoming a full supervisor system.

The supervisor consists of several classes shown in Fig. 6. The main class is the `SupervisorEmulator` which implements the `Supervisor` interface from the application framework. This object is responsible for preparing a new `LVL1Result` and sending it to one of the steering processors. The selection of an appropriate steering processor is the task of the *scheduler*. It also accepts the results of the Level 2 trigger via its `accept()` method.

The input part for Level 1 ROIs is separate and hidden behind the `LVL1Source` interface. This makes it possible to provide different implementations for Level 1 input without changing the rest of the system.

6.1 LVL1 Input

The `LVL1Source` class is the interface to the first level trigger. After initialization it provides essentially one method `get_result()` which returns a new `LVL1Result`. The `result_available()` method returns true if more results can be read.

There are three implementations of this interface and one can select them from the configuration file:

`LVL1_FileSource` will read event data from a file. The same data file can be used by the steering and the Robs to have consistent event data throughout the whole system. `LVL1_PreloadFileSource` is a variation of this class to load the whole file into memory at startup time.

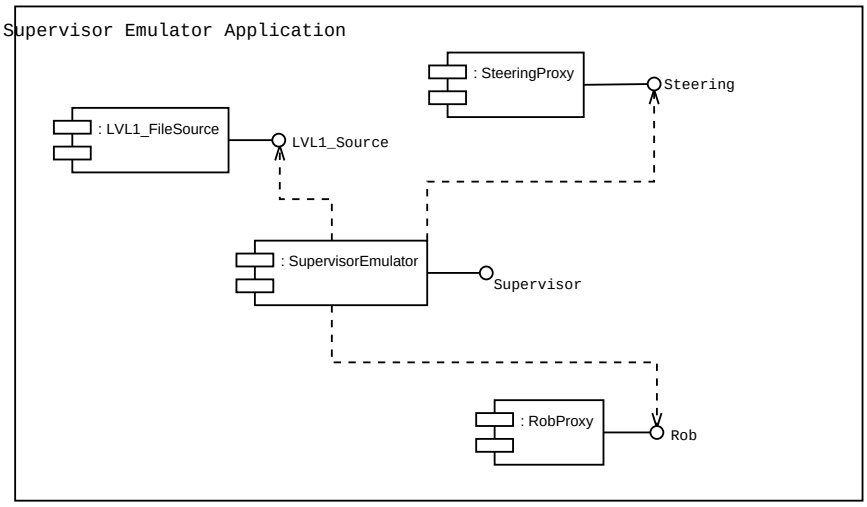

Figure 13: Supervisor

`LVL1_GeneratedSource` will generate random level 1 ROIs according to a distribution specified in the configuration file. Since there is no corresponding event data available, this option only makes sense for the skeleton applications.

`LVL1_SlinkSource` will read the level 1 ROIs from the real ROIBuilder. Again there is will be no corresponding event data. This option is mainly useful for a 'component test' of the full supervisor in a real system.

6.2 Scheduling

While in practice the scheduling code is part of the application framework and completely hidden from the supervisor itself, it logically belongs to this application.

The scheduler interface is defined in class `Scheduler`. After initialization the only method used is `nextProcessor()`. A future version of the scheduler will include additional interfaces, e.g. to inform it about dead nodes or nodes which have come back into the system after an external intervention.

Several implementation are available:

`ScheduleRoundRobin` does a simple round-robin scheduling of available nodes.

`ScheduleRandom` selects an available node randomly.

`ScheduleWeighted` schedules nodes according to their relative performance. Nodes with a higher performance will be scheduled more often than slower nodes.

7 Readout Buffer

At the moment only a ROB emulator application is available as part of the Reference Software, although it is intended to have a version with real hardware running at a later stage.

The ROB emulator reads its data from an external file and stores all events in memory. It implements all operations described in the ROB interface class. The ROB is actually used by multiple components in the system, typical use cases can be seen in Fig. 7.

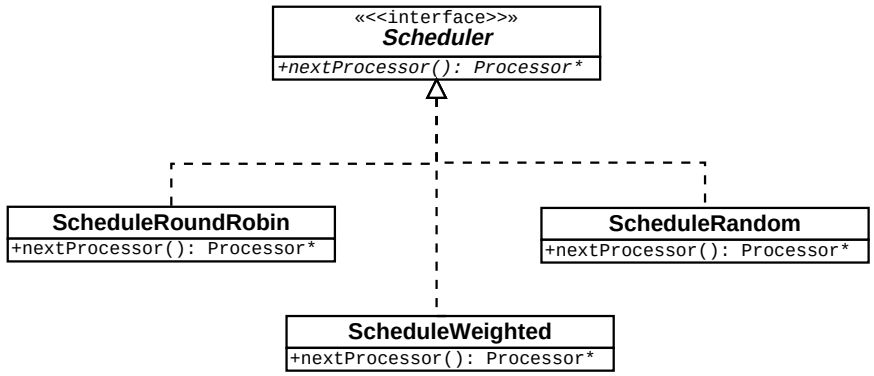

Figure 14: Scheduler

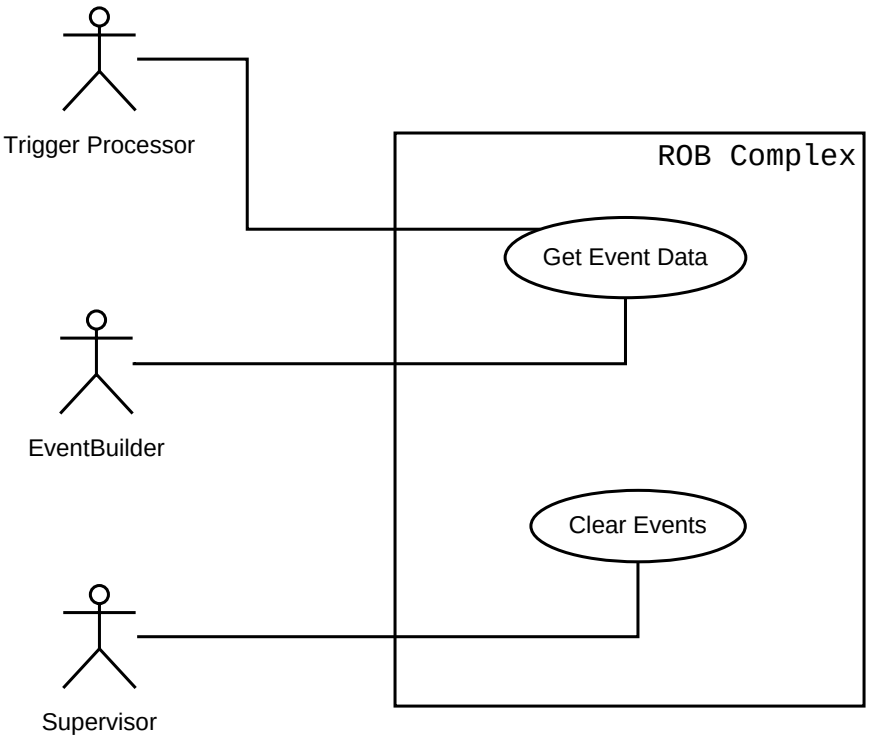

Figure 15: ROB use cases

7.1 Data

8 Trigger Processor

The trigger processor is the place where the actual second level algorithms are running. The *steering* is the menu-guided process by which the level 2 decision is made. It is embedded in the usual application framework and makes use of its services.

Typically there will be multiple steering worker threads to either make use of the processor when another thread is waiting for event data, or to take advantage of multiple CPUs in the system. As already mentioned the processing of a single event is always handled by a single thread.

8.1 Steering

The steering itself is guided by trigger menus and algorithm tables. Menus specify the physics signatures for which the event will pass the level 2 trigger. Every physics signature consists of one or more elements, like electrons, photons, jets, missing energy or invariant mass elements. In the reference software these elements are of type `RequestedElement` and its subclasses. The `PhysicsSignature` class represents one line in a menu.

On the other hand each event will have a number of trigger objects, called `TriggerElement`s which are created during the execution of the various algorithms (starting with the level 1 ROIs).

To check if an event passes a given line in the menu, the steering has to match the existing physics objects against the physics signatures. In each step the *algorithm table* specifies one or more feature extraction algorithms and one hypothesis routine which are run for each trigger object. Typically the feature extraction routines will create clusters, tracks etc. The task of the hypothesis routine is to use these features and confirm or reject the current trigger object, e.g. by applying cuts or other criteria.

The remaining `TriggerElements` will then be matched against the `PhysicsSignature`s of the current menu. If the event passes this step the whole procedure is repeated in the next step, with a different algorithm table. This allows to start with rather simple algorithms in the first steps and add more complex algorithms only later.

After each step the event can be rejected, which means the rest of the processing can be aborted. This sequential processing reduces the overall computing capacity required since no unnecessary calculations are performed.

Menus and algorithm tables are specified in the configuration file, an example is given in the following:

```
MenuStep1 = [
  [ "e(p,15)" ],
  [ "e(p,10)", "e(p,10)" ],
  [ "j(p,100)" ],
  [ "j(p,40)", "j(p,30)" ]
];

MenuStep2 = [
  [ "e(p,15)" ],
  [ "e(p,10)", "e(p,10)" ],
  [ "j(p,100)" ],
  [ "j(p,40)", "j(p,30)" ]
```

```
];

Menu = [
  # an array of Menus, one for each Step
  $MenuStep1,
  $MenuStep2
];


#
# algorithm table for steering
#

Builder1 = [
  [ "e(p)",    [ "calo_e_fex" ], "e_calo" ],
  [ "j(p)",    [ "calo_j_fex" ], "j_calo" ]
];

Builder2 = [
  [ "e(p)",   [ "trt_fex" ], "e_trt" ],
  [ "j(p)",   [ "trt_fex" ], "j_trt" ]
];

# full list of builders
Builders = [
  $Builder1,
  $Builder2
];
```

The `T2SingleNode` class provides a framework to run a full steering plus feature extraction algorithm application on a single node. This is a single-threaded non-networked version which is mainly used for functional tests. The `t2single` executable includes all available algorithms and can be completely configured by the user.

8.2 Feature Extraction Algorithms

Feature extraction algorithms are routines which take raw event data and produce 'physics' features like clusters and tracks. They are called when needed from the steering layer. Their task is well-defined and they never make directly any decision about an event. This is always left to the hypothesis routines.

The reference software provides a simple framework in the form of the `T2SingleFex` base class to run and debug a single fex algorithm. Data is read from file and each ROI is passed to the algorithm if it's type is acceptable. Output can be generated any way the user likes, including calls to third-party libraries like cernlib (which are normally forbidden inside the reference software).

The following algorithms are currently available as part of the reference software:

- EM, Jet and τ calorimeter algorithms, generating clusters of an appropriate type as output [12].
- TRT algorithms for both ROI guided and low-luminosity operation (fullscan), generating a list of tracks [13].

- LUT based precision detector for both pixel and SCT detectors [14].
- Kalman filter based algorithm for the precision detector.

All algorithms inherit from a templated base class `BaseAlgorithm`. The template parameters are the type of event data the algorithm expects as input and the result type of the algorithm. In addition some arbitrary user defined data can be passed to the `extract()` method of the class, which does the actual work. This is used when an algorithm requires e.g. data from a previous processing step, like the track finding in the inner detector which is seeded from the TRT in the low luminosity case.

8.3 Data

When feature extraction algorithms require event data, they use the `DataCollector` interface to request it. There are two implementations for this interface, the `FileCollector` and the `RobCollector`. As their names imply, the first one uses a file to read the event data, while the second one issues a request to a ROB. The `FileCollector` is used in stand-alone and single-node mode, while the latter is used in a distributed system. However, since these objects are setup at initialization time, their use is transparent to the algorithms themselves.

All data requests are actually handled outside the algorithm itself, only the appropriately formatted data is then passed down to the `extract()` method. This makes the algorithm code completely independent from the rest of the system, and allows the insertion of arbitrary reformatting code just before its execution.

9 Program Manager

The program manager package is responsible for starting all the necessary processes on a distributed system. Since it never interacts directly with the application software itself, it can be considered an external utility. In fact, on systems where there are other mechanism available or required (e.g. the MPI based Paderborn cluster), the program manager functionality is replaced by some native tool.

The design of the program manager is modelled after the corresponding backend software design, so that in a later tigher integration with the DAQ/BE in can be replaced by the respective components without major changes. It is based on an *agent* helper process which is running on each machine in the system. The program manager client will contact the local agents and instruct them to do certain actions, which can't be done remotely, like starting a process, killing it, checking if it still alive etc.

In addition to the basic functionality of starting and stopping processes, it also keeps a process database, which can be queried about the current status of each process.

10 Run Control

The run control system makes it possible to control a distributed system from a single place. It can issue commands to the applications in the system and query their state (as described in 5.5). It is responsible for ensuring that all applications execute their internal steps in the correct order.

The run control makes use of existing protocols instead of inventing a new one. All information is already available either through the ERL or the monitoring subsystem. ERL is used to inform the run control of any internal state changes (e.g.

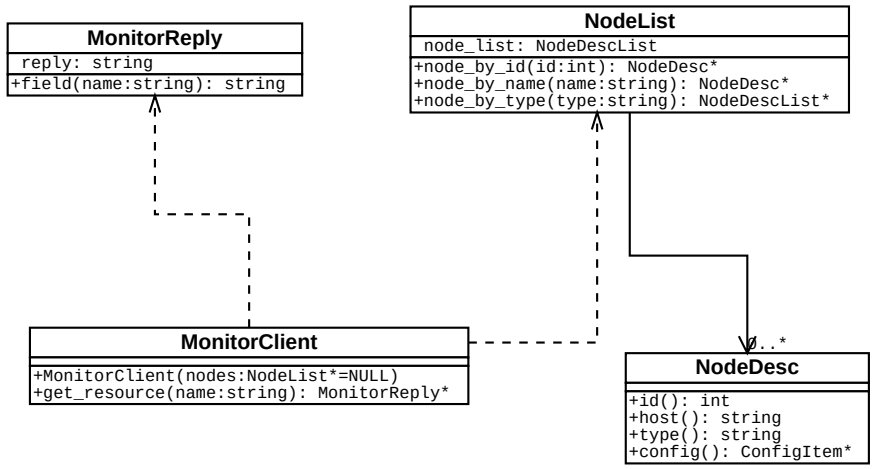

Figure 16: Monitor client classes

from `Running` to `Finished`). The monitoring system is used to query the states explicitly or to issue commands (via the `set()` method of an appropriate resource).

Although there is a single application `t2rc` which provides a command-line based user interface for the run control, all the necessary elements are also available in the form of reusable C++ classes. This makes it possible to implement the run control interface in a different form, e.g. as a graphics based one, without reinventing most of the functionality. These classes are known as the *client interfaces* to the ERL and monitoring package.

10.1 Client Interfaces

The monitoring client interface consists of the `MonitorClient` and the `MonitorReply` classes. There is an additional helper class `NodeList` which is also used by the ERL client classes.

The `NodeList` class simply represents all nodes in the system. A `NodeDesc` (for node descriptor) can be obtained for each node, containing all the information from the configuration file. One can query for single nodes or lists of nodes by ID, hostname, type etc.

The `MonitorClient` class provides the basic interface to request resources. It has methods to request, reset or set a resource on a specified node. In the case of a request, a `MonitorReply` object is returned. This object gives access either to the original message text or to specific fields inside the message. In short, it helps in parsing the reply from the monitoring system.

The ERL client interface consists of three classes: `ERLMessage`, `ERLCallback` and `ERLReceiver`. `ERLMessage` represents a message which has been received from an ERL server. Like `MonitorReply` it is essentially a helper class to access the fields in the message in a more convenient way. Queries for certain fields are possible, as well as accessing the (optional) parameters in the message by name.

The `ERLCallback` class is an interface that interested receivers have to implemented to be informed about incoming ERL messages. There is only one (pure virtual) method here: `receive(const ERLMessage& msg)`. An `ERLCallback` object can be registered with the `ERLReceiver` class and will then be called for every new message.

The `ERLReceiver` class implements the communication with the ERL server. Apart from the methods to register or unregister an `ERLCallback` object, the class

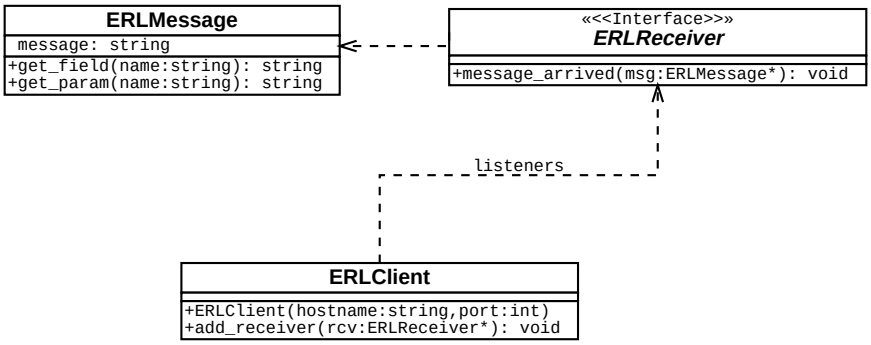

Figure 17: ERL client classes

has only a `run()` method. This will enter an endless loop, where the object will wait for incoming messages and then despatch them to any listeners. Therefore `ERLReceiver` will be typically used in a separate thread, so that it doesn't interfere with the rest of the application.

10.2 Run Control Clients

The `RCClient` class pulls together all the pieces mentioned so far, and provides an easy interface to all the run control functionality. It mostly hides the underlying implementation in terms of ERL and monitoring protocols. Instead there are methods to issue the available run control commands, either to a single node or a group of nodes, and virtual methods which are called in case of an event (like a state change) and which can be overridden by a subclass.

`RCClient` provides no user interface in itself, instead a full 'run control application' can be built on top of it by implementing the interaction with the user. An example is the `T2RC` class, which is a command-line based user interface. The user can type commands which are interpreted and forwarded to `RCClient`. Changes of the state of an application are reported to user by printing a message on the screen etc. Commands can also be read from an external file, therefore making it possible to write simple scripts to control the overall execution of the system from beginning to end.

References

[1] The Atlas Level 2 Pilot Project, `http://atlasinfo.cern.ch/Atlas/GROUPS/DAQTRIG/L2PILOT/l2pilot.html`

[2] X/Open `http://www.xopen.org`

[3] Gamma et. al, Design Patterns, Addison Wesley, 1994.

[4] An integrated system for the ATLAS High Level Triggers:Concept, General Conclusions on Architecture Studies,Final Results of Prototyping with ATM, ATL-COM-DAQ-2000-016, 2000.

[5] Implementation of the Message Passing Layer over SCI for the ATLAS Second Level Trigger Testbed, ATLAS internal note, ATL-COM-DAQ-2000-026, 2000.

[6] Evaluation of SCI components for the ATLAS Second Level Trigger, ATL-COM-DAQ-2000-030, 2000.

[7] Dolphin http://www.dolphinics.no

[8] Message Passing Interface Forum http://www.mpi-forum.org

[9] Srinivasan, R., RPC: Remote Procedure Call Protocol Specification Version 2, RFC1831, Sun Microsystems, Inc., 1995. http://www.ietf.org/rfc/rfc1831.txt

[10] Object Management Group, http://www.omg.org

[11] Inter Language Unification ftp://ftp.parc.xerox.com/pub/ilu/ilu.html

[12] First Implementation of Calorimeter FEX Algorithms in the LVL2 Reference Software, ATLAS internal note, ATL-COM-DAQ-2000-013, 2000

[13] Global Pattern Recognition in the TRT for the ATLAS LVL2 Trigger, ATLAS internal note, ATL-DAQ-98-120, 1998.

[14] Performance of a LVL2 Trigger Feature Extraction Algorithm for the Precision Tracker, ATLAS internal note, ATL-DAQ-99-013, 1999

[15] Running the ATLAS Second Level Trigger Software on a large commercial cluster, ATLAS internal note, ATL-COM-DAQ-2000-027, 2000.