Linux AEM
FAQ

Draft0.3

frederic.rossi@ericsson.ca

This is a list of questions I have received directly or indirectly while discussing about AEM. The purpose is to clarify and help understand what is AEM and what it is not.

YOUR QUESTIONS AND REMARKS ARE IMPORTANT. Because it helps to make AEM better to answer your needs. So don't hesitate to send me questions or remarks, I will be pleased to answer or to add them to this faq.

What is AEM?

AEM (Asynchronous Event Mechanism) is a set of building blocks providing the needed operating system support for the implementation of a native event−driven user interface.

The current implementation provides:

1. Asynchronous notification for accept, read, close, TCP state and socket state.

2. An interface for timers.

Other interfaces can be provided too. It mainly depends on you if you are interested in using other interfaces like an asynchronous file system interface (and related like dnotify), overload notification and so on.

AEM is a kernel patch for the Linux kernel 2.4.6 and 2.4.19. Where are also porting to the Linux kernel series 2.5/2.6. and stabilizing the support for SMP architectures.

How to use AEM

You can download a patch for your kernel from the main page (http://sourceforge.net/projects/aem) and recompile your kernel.

Lot of examples of how to use AEM are provided in the *testsuite* package.

What kind of applications is AEM targeting??

AEM is targeting multi−layer applications where a huge number of threads is usually started to handle events and where notification between layers can become a bottleneck under high load. This is typical of distributed applications.

Also, AEM exports a very simple interface to the user which is an advantage for everyday programming. In particular, it is quite straightforward to implement a server using AEM compared to *select,* even when it is a single−threaded server.

How does AEM compare to Posix AIO?

AIO works on file and socket descriptors providing asynchronous notifications for read/write operations using signals. When an application receives a notification with AIO, it has to get the active socket or file descriptor identifier. Because the notification provided by AIO only tells you that something happens: it doesn't say where. Traditionally you receive the same SIGIO even if you have activity on thousands of sockets.

When using the asynchronous socket interface provided by AEM you are given the possibility to declare a handler for read operations. Then when some activity are detected on the socket, the handler is directly executed by the kernel. You even don't have to search for the correct descriptor and fetch for your data. Everything is directly given in parameter of your callback functions for all events managed by AEM.

What happens when a socket is closing and AEM tries to read data on this socket? It will execute the event handler for the read and returns −EPIPE in the *len* variable in parameter. Other error numbers can be provided to user space like −ENOMEM if no memory was available for data allocation. Applications are notified even in case of failure.

How does AEM compare to epoll?

Epoll is an alternative to *select*. *Epoll* reports activity occurring on descriptors and had proved to be very scalable for simple software configurations. *epoll* is not a generic event mechanism (but AEM is) although *epoll* is an efficient notification mechanism. *Epoll* can be used in conjunction with AIO for example to find active descriptors. We are currently trying to make AEM/*epoll* work together like we did for AEM and *select*.

Like *epoll*, AEM has no notion of *searching−for−the−active−descriptors*. Why? Because it is fined−grain and each active descriptor provides direct notification to the application. It looks like each descriptor is executing its own handler.

Both *epoll* and AEM do receive notification very quickly. But then after notification, the application has to handle the descriptor, get data and run a thread for managing the transaction. *Epoll* stops the job at this point, whereas AEM continues to deliver by going and fetching the data from the kernel and give it to the application.

Even if *epoll* is very efficient at returning active descriptors quickly, it doesn't save the application designers building large scale software to make use of a heavy multi−threaded framework. This is unfortunately where a large part of the latency between the arrival of a notification and the execution of the proper routine is. This is true especially under high load.

AEM executes automatically a user callback upon data completion. The execution can take place either in the same execution context or in a new execution context. This has

a much more impact than simply providing notification to user space. AEM is taking care of the execution of handlers and how these are to be executed. This is completely transparent from the user point of view.

Why do we need to execute handlers directly? Because we want to be sure they will be executed in some fixed interval of time and we want to have full control on this. For critical events like overload notification this is even more important since it is usually a matter of few milliseconds.

How does AEM compete with AIO/*epoll*?

We can have hybrid solutions based on AEM/select. For example you can use AEM on the first callback and use *select* inside event handlers. We are also trying the hybrid solution based on AEM/epoll.

It is possible anyway to give a *functional* comparison of both just for curiosity:

1. AEM uses a new object as event references (not descriptors),
2. AEM is not restricted to file and socket descriptors,
3. AEM implements sporadic or periodic events monitoring,
4. AEM makes use of priority enforcement,
5. AEM manage memory on behalf of processes,
6. AEM can execute handlers in the same execution context or in a new process context.
7. AEM provides routines to stop/start or shutdown events explicitly.

AIO/epoll doesn't provide any support for concurrent access to shared data. This means you have to design and implement your own exclusion mechanisms. If you are used to design your applications using a multi−threaded architecture, you have to manage threads and their synchronization.

What does AEM compare to?

AEM belongs to a group of applications providing asynchronous execution of processes. Other kinds of similar mechanisms are the *x−kernel, active messages* or even *Microsoft IOCP* to some extent.

AEM provides basic building blocks to build a higher level asynchronous interface. AEM is not limited to socket operations and network protocols. It is possible to implement callbacks for overload notifications or for whatever critical events. Of course, even if it is a generic and very flexible, the support in the kernel must be given for all events.

How does AEM interact with signal handlers

Execution of signal handlers and event handlers go through different paths in the kernel. When a signal handler execute, the execution of event handlers cannot happen until the signal handler exits. On the other hand, when an event handler executes nothing prevent the execution of a signal handler. Both can work together.

How does AEM interact with *select()* and *epoll*?

AEM interacts very well with *select()* and *epoll*. You can use the asynchronous socket interface provided by AEM and/or the traditional socket interface (*select*, *read*,...). You can even develop hybrid applications that uses both in the same time. You can use a blocking *select()* or *epoll* inside an asynchronous event handler for example.

What about *async_select()*?

The good news is that there is no *async_select* at all, because it cannot exist in an event−driven world. The basic behavior of select is to wait for activity on some descriptors. Whereas the semantic attached to the asynchronous paradigm is to do something useful instead of waiting. So when you use the asynchronous socket interface provided by AEM you just have to define a handler for the (async) accept call . Once a connection is coming in, the handler for accept if executed asynchronously. You don't have to wait for it explicitly. On the other hand, if you don't want to receive incoming connections you can suspend or shutdown the corresponding event (using the identifier).

Hopefully, since a newly created socket descriptor is provided in parameter of the handler for this connection, you can use *select* on the descriptor inside the handler. It is possible to mix the AEM event−model and the traditional sequential model without any problem.

Does AEM provide a general publish/subscribe mechanism?

AEM doesn't provide a generic publish/subscribe mechanism by itself. But AEM is the operating system support for such solution. If you need a full publish/subscribe mechanism as defined by Siena or Corba, then consider AEM+TIPC (tipc.sourceforge.net) altogether. Support for AEM into TIPC on Linux is planned for the near future.