The Benefits of an Event-Based Mechanism in Carrier-Grade Linux[1]

by

Frederic Rossi
Ericsson Research Canada
Open Systems Lab

This article is the first of a series describing a new event-based mechanism for Linux. This particular one is focusing on the motivation, requirements and the benefits of such mechanism for carrier-grade Linux.

Introduction

The work for supporting a native event-based system in the Linux kernel started as a research project in 2001 at the Open Systems Lab (Ericsson Research, Corporate Unit) in Montréal, Canada. The goal was to provide Linux with an event-driven environment that could deliver better performance compared to existing solutions in the context of telecom applications.

There has been a growing interest to bring the Linux operating system to a carrier-grade level. As an example, the OSDL Carrier-Grade Linux working group [10] is currently drafting the set of requirements that will turn Linux into a solid carrier-grade server for Next Generation Networks.

Operating systems for telecom applications must ensure they deliver a high response rate with a minimum down time (less than five minutes per year - 99.999% of uptime) including hardware, operating system and software upgrade. In addition to this goal a carrier-grade system must also take into account characteristics such as scalability, high availability and performance.

For such systems, thousands of requests must be handled concurrently without impacting the overall system's performance even under high load. Subscribers can expect some latency time when issuing a request but are not willing to accept an unbounded response time. Such transactions are not handled instantaneously for many reasons and it can take some milliseconds or seconds to reply. Waiting for an answer reduces applications' ability to handle other transactions.

Many different solutions have been envisaged to improve Linux's capabilities using different sorts of software organizations like multi-threaded architectures,

[1]Published in *Linux Journal* , July 2003, #111, p32-36.

by implementing efficient POSIX interfaces or by improving the scalability of existing kernel routines. We think that none of these solutions are adequate for true carrier-grade servers.

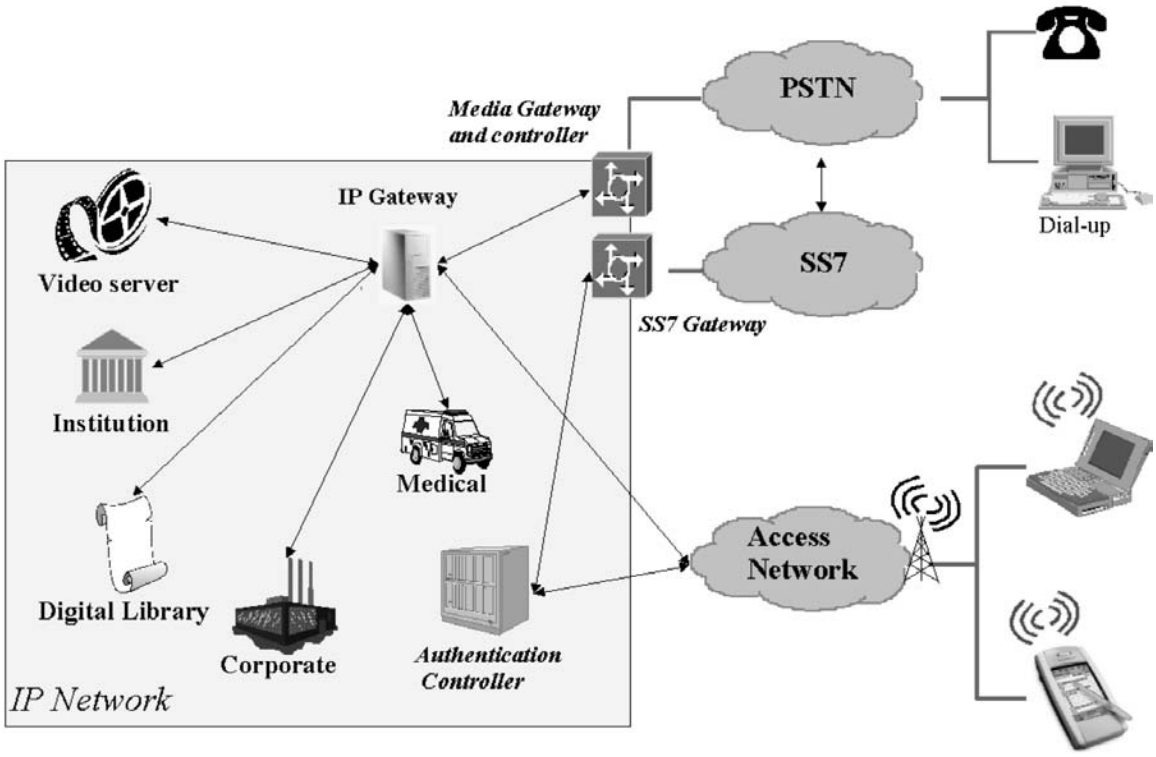

Figure 1: *Architecture and interoperability between the PSTN and the IP networks.*

In order to understand our point, this article will first give an overview of telecom networks. The purpose is to try to clarify the requirements for a carrier-grade operating system. After this introduction we will show the benefits of a native asynchronous event mechanism to better support carrier-grade characteristics.

Carrier-Grade Requirements

Telecommunications is concerned with the establishment of telephone calls between two devices and the transport of voice over wire links. This is (see figure 1) the traditional *Packet Switched Telephone Network* (PSTN). More specifically carriers use the term *signaling* to indicate the establishment of a telephone call between two devices. Signaling in the PSTN is done through the *Signaling System 7* (SS7) protocol stack. SS7 performs call routing and build a path to the destination telephone through the circuits. The two phones are connected once the path is established and the voice can be carried over. The SS7 protocol is able to handle call routing, call forwarding and error conditions.

The flexibility, cost performance and continuous growth of the Internet are driving

a migration of many telecom services to the Internet. This helps establishing IP technologies as the new standard for all communications services. These two types of networks are based on different technologies and require the utilization of signaling and media gateways for inter-working purposes.

Gateways perform the translation of information between different types of network. For example, an SS7 gateway is used to encapsulate signaling over the IP network through the SCTP protocol. A media gateway is used to encode and decode the voice coming from the PSTN network and going to the IP network and vice-versa.

The signaling and the media gateways provide connectivity towards the IP network for calls coming from the PSTN network. Signaling gateways must perform protocol encapsulation in order to carry the syntax and semantics of SS7 messages over an Internet protocol, such as SCTP (*Stream Control Transmission* Protocol) over IP or UDP over IP. Signaling servers must be able to scale with respect to their system capabilities as the number of concurrent requests increase.

With a media gateway, operators can implement transport of streaming data between the PSTN and the IP networks. The number of connections they can accept is dependent on their hardware which can vary in size from one to thousands of interfaces. Media gateways must support a large number of connections in real-time.

Authentication, Authorization and Accounting (AAA) servers maintain databases of user profiles. Typically, one or two AAA servers may contain the information about several millions subscribers which belong to a particular network operator. It is common to observe peaks of thousands of concurrent authentication requests per seconds. Such variation in the number of connections is difficult to plan in advanced. AAA servers have this critical role of controlling the access to the IP network and are not allowed to fail. They require soft real-time capabilities to be able to reply within few milliseconds for most requests.

Media servers provide specialized resources and services to end-users such as video conferences, video servers, applications, emails. An important aspect of these carrier-class systems is scalability. These platforms are able to accept an linear increase of the number of transactions with respect to the number of processors, interfaces or bandwidth. Telecom operators speak of *linear scalability:* the cost per transaction or user should not increase while scaling-up a server.

In case of failure or unplanned interruption carrier-grade platforms are able to recover automatically or fail-lover to another server through networks redundancy procedures. Live software upgrade or hot swap of hardware devices are also part of the 99.999% service availability.

Linux has proved to be stable and consistent over the years and it is already an attractive option for carriers. But in order to become a key element of telecom networks it must be enhanced with components providing these much needed carrier-grade capabilities

Matching Performance and Architecture

In the traditional programming model, software components explicitly synchronize with others. This is the common model when lot of interactions are needed. For example, the typical approach is to use *select()* or *poll()* to listen to file descriptors. A generic implementation of select will scan the entire array of descriptors. This is not scalable because the time it takes to detect activity on a descriptor is proportional to the size of the array. This increases the application latency and leads to a decrease of the overall system performance.

Scanning an array of descriptors or waiting for data consumes processing time. A common idea in the design of efficient algorithms is to handle system events asynchronously. Some examples of mechanisms that provide event notification to user space applications are the POSIX AIO [6], *epoll* [7] or the BSD *kqueue* [5].

When describing the efficiency of such mechanisms it is common to compute the average time it takes from the moment an event is detected in the kernel to the moment it is effectively handled by the application. One of the main reason is that micro-benchmarks for this type of method are not relevant. Such mechanisms can be very efficient locally but inefficient when combined with others that are not necessarily well adapted like multi-threaded architectures. As an example, many web servers use a pool of threads, started when the application is launched. A typical architecture is to use one dedicated thread to manage incoming connections and one thread per transaction. This is usually quite efficient for a few number of incoming connections but inefficient under high load.

Multi-threaded applications are needed when a high level of concurrency is required between objects competing to gain the CPU. Well known examples are found in high performance computing applications where the speed of execution of every thread is important but where the number of threads to run is not high.

Threads provide a sequential and synchronous model of development and has become the standard way of implementing applications where a high level of concurrency is needed. But flaws in the design of applications or flaws in handling synchronizations can easily create system contentions and impact the overall system performance. It has been established that programming with threads is quite difficult [8] and mainly leads to applications unable to execute properly under high load.

In telecom applications there is no competition between threads. But concurrency occurs when handling common objects like distributed data structures. For these applications threads are needed to provide concurrent accesses to shared data.

Telecom applications are used to face thousands of transactions per seconds and hundreds of simultaneous connections on the same processor. In addition, system events are to be taken into account like database accesses, applications faults, overload notifications, alarms, state change of system components, etc... Thousands of events can be generated in the same system during the execution of an application. Managing events with threads would be inefficient.

Traditional asynchronous mechanisms try to solve this scalability issue either by preventing applications from waiting unnecessarily or like *epoll* on Linux aim at improving the detection of active descriptors. Unfortunately these solutions are limited to file descriptors which represents only a fraction of events of interest. Also, starting a huge number of threads like for web servers to handle these events would create a bottleneck and aggravate the situation.

Scalability is not the only issue to tackle if one wants to improve efficiency of large scale servers as we will see in the next section.

Event-Based Architectures

The development of complex distributed software architectures demands for the implementation of a mechanism that is suitable to take benefit of system resources at run-time.

A promising solution that is more appropriate to address the above issues is the introduction of an event-based mechanism in Linux. Such mechanisms allow a real cooperation between the operating system and the applications. They provide components that are able to register for events and later be asynchronously notified through the execution of handlers.

If we compare signal handlers and event handlers, we find that these later are more informative because they bring the data directly to the application. Basically, an asynchronous event mechanism can be used to implement generic user-level handlers triggered by system events or to implement periodic monitoring components like timers. The first case is particularly interesting if an application don't know when an event occur. When receiving events asynchronously the application can take action without recovering all the necessary data because they are supplied in parameters.

Some investigations have already been done in the past regarding fast message passing mechanisms which are based on the same principles as asynchronous events. For example in *active messages* [1] messages execute asynchronously

on the stack of the receiver process, in *popup threads* [2] a thread of execution is created for every handlers and in *single-threaded upcalls* [3] a dedicated thread is created on each processor. AEM [4] is an emerging mechanism, which offers a native environment for the development of applications requiring real asynchrony. For example, we used AEM to implement a native asynchronous socket interface for TCP. In AEM, which provides asynchronous execution of processes, the choice is left at registration time to define a handler that will be executed on either the current execution task or a new thread of execution. Some other research projects have proposed similar solutions to improve web servers capabilities under high load [9].

The main benefit of the event paradigm is the integration in the same mechanism of event handling and thread management. Concretely it gives full control on resource consumption.

Performance is really a goal for event-based mechanisms. Decoupling event management from the application permits to increase locality, to take benefit of different memory allocation schemes or to influence the scheduler decision. For example, soft real-time responsiveness is ensured by enforcing process priorities depending on pending events.

This emerging paradigm provides a simpler and more natural programming style compared to the complexity offered by multi-threaded architectures. For example it proves its efficiency for the development of multi-layer software architectures where each layer provides a service to the upper layer. This type of architecture is very common for distributed applications.

In figure 2 we illustrated a typical distributed application based on an event-driven model. It is composed of many software components and a process represents one layer of the application. In distributed applications a lot of local and remote communications are engaged either at the same level or at a different level. In this figure, synchronous communications are represented by plain lines and asynchronous communications are represented by dash lines.

In many situations such applications have to provide services that must operate world-wide with very good performance. It is essential that these applications are able to take benefit of hardware resources and scale linearly with respect to the platforms capabilities.

The design of these software must ensure that no deadlock and race condition are possible between all components. The impact of such design flaws on the system integrity can be really catastrophic. This situation is difficult to solve when using a multi-threaded approach because it is hard to detect and to correct due to the high number of possible configurations. An event-based mechanism reduces the chance of introducing points of failures by controlling the number of threads started asynchronously. It is easier to guaranty atomicity of handler

executions since the mechanism is kept in the kernel.

The system resources are limited and the number of processes that can be started is always limited. The alternative that is given to choose at registration time the type of handler to execute permits to produce applications more robust as the load increases. The main advantage for applications is the possibility to mix sequential code and asynchronous code. It is then possible to design applications that exploits capabilities of both strategies.

An event-based framework offers to operators dynamic reconfiguration with a minimum of impact on the system uptime. Hardware hot swap and dynamic software upgrade must be possible without restarting the system. Distributed applications are built up of a large number of interacting components and upgrading such software is a critical operation. Telecom platforms require

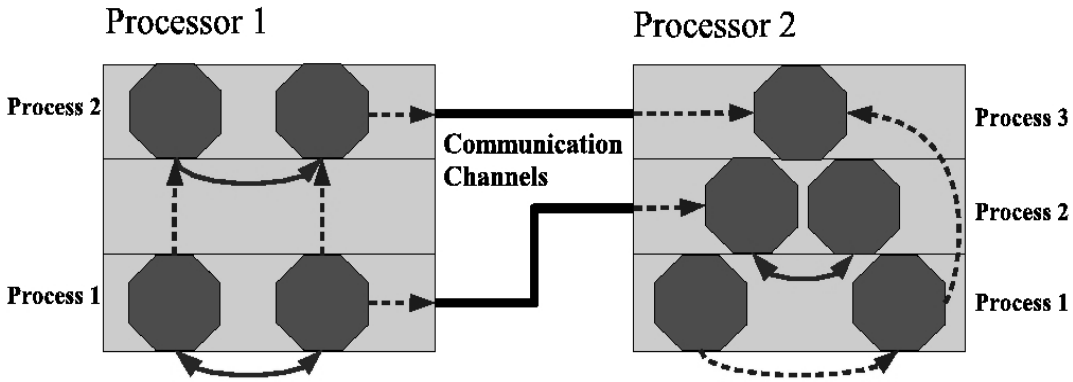

Figure 2: *A multi-layer distributed application designed with an event-based model running on two processors . Each layer is single threaded and the communication between application components is either synchronous (plain lines) or asynchronous (dash lines).*

99.999% uptime for all services. During maintenance operations the services cannot be stopped, or it would impact other service platforms and subscriber requests connected to it. Software upgrade must be performed gradually. Event-based mechanisms introduce the potential for such capability to distributed applications. As we can see in figure 2, there is no direct dependencies between software layers if communication is performed asynchronously. It is then possible to replace some of the application parts without major disturbance.

Conclusion

An event-based mechanism provides a new programming model which offers a unique and powerful support for asynchronous execution of processes to software developers. Of course, it radically differs from sequential programming styles we are used to but it gives a design framework more structured for software development. It also simplifies the integration and the interoperability of complex software components.

The strength of such mechanism is to be able to combine synchronous code and asynchronous code in the same application or even mixing these two types of models into the same code routine. With such a hybrid approach it is possible to take advantage of their respective capabilities depending on the situation. This model is especially favorable for the development of secure software and of the long-term maintenance of mission critical applications.

In this issue we presented the benefits of using event-based mechanisms in carrier-grade servers. In the next coming two issues we will show how AEM has been implemented to provide such support into the Linux kernel and how to use it for software development.

Acknowledgments

The Open Systems Lab for reviewing and approving the publication of this article, Laurent Marchand at Ericsson Research Canada for useful comments and Philippe Meloche student at Sherbrooke University.

About the author

Frederic Rossi is a researcher at the Open Systems Lab at Ericsson Research, Corporate Unit, in Montréal, Canada. He is involved in research activities leading to designing kernel components for the advancement of carrier-class operating systems. He can be reached by email at frederic.rossi@ericsson.ca.

References

[1] T. von Eicken, et al, "Active Messages: A Mechanism for Integrated Communication and Computation", In *19th International Symposium on Computer Architecture,* pp. 256-266, Australia, May 1992.

[2] N. C. Hutchinson and L. L. Peterson, "The x-kernel: An Architecture for Implementing Network Protocols", *IEEE Transactions on Software Engineering,* 1(17), pp 64-76, 1991.

[3] R. Bhoedjang, et al, "Friendly and Efficient Message Handling", In *29th Annual Hawaii International Conference on System Science*s, pp. 121-130, Hawaii, USA, January 1996.

[4] Linux AEM - Home Page, http://aem.sourceforge.net/.

[5] J. Lemon, "Kqueue: A Generic and Scalable Event Notification Facility", *FreeBSD Project*.

[6] The Open Group, "Posix Asynchronous I/O", *The Single UNIX Specification*, http://www.opengroup.org/.

[7] D. Libenzi, "/dev/epoll", http://www.xmailserver.org/linux-patches/nio-improve.html/.

[8] J. Ousterhout, "Why Threads Are A Bad Idea", *USENIX Technical Conference*, January 25, 1996.

[9] G. Banga, et al, "A Scalable and Explicit Event Delivery Mechanism for UNIX", *USENIX Annual Technical Conference,* pp 253-265, June, 1999.

[10] OSDL CGL - Open Source development Lab - Carrier-Grade Linux, http://www.osdl.org/projects/cgl/.