Component-based Active Network Architecture

S. Schmid, J. Finney, A.C. Scott, and W.D. Shepherd

Distributed Multimedia Research Group
Computing Department
Lancaster University, UK

{sschmid, joe, acs, doug}@comp.lancs.ac.uk

Abstract

We believe that the 'killer application' of active networking will be the provision of services which are not known to either the hardware manufacturer or the vendor at the time of network deployment. This would enable the rapid deployment of new services by third parties, and an unprecedented level of customisation by network administrators and end users. This vision, however, requires a degree of flexibility not yet seen in any of the currently proposed active router architectures.

The lack of flexibility found in current active router designs has inspired us to develop a novel component-based active router architecture enabling extensible network programmability based on a progressive service composition framework. This document presents this architecture, LARA++, discusses its rudimentary concepts, and illustrates by example how its component architecture can benefit network programmability.

1. Introduction

Experiences in network operation has shown that the lack of flexibility and extensibility in current access and edge networks hinders or delays the evolution and deployment of advanced network technologies. Active networking aims to overcome these limitations by replacing current network devices with programmable nodes. From experiences with our first-generation active router, LARA [1], we have learnt that the programming capabilities of these early active router architectures can be too restrictive.

For example, many active network applications require only simple modifications or extensions of existing protocol stacks and/or services in order to optimise or improve the operation of the node (for example, to support a new protocol option or control message type).

However, a study of several available active network architectures has shown that most existing platforms do not offer the degree of flexibility required for the dynamic deployment of such services. This inherent lack of flexibility is caused in some active router architectures due to drives for simplicity or for performance reasons, others are limited due to the targeting of the platform toward narrow ranges of end-user applications.

1.1. An Example Active Network Scenario

The following scenario illustrates the importance of a flexible programming framework for active routers. Consider an administrator of a site area corporate network, who wants to enable a simple congestion-control mechanism, which supports differentiated classes of service based on a certain packet marking technique (for example, based on a new IP option). The administrator would then want to upload active code onto the active router which intercepts all relevant packets in order to apply the local congestion control algorithm. At the same time, there might be an end user on the network, who wants to upload fast mobile handoff support[1] (as suggested in [2]) for data streams to its mobile device. The challenge here is to allow active applications from various users (with different privileges) to co-exist and inter-operate. For the example above this means that packets streamed to the end-user's mobile device benefit from the "new" congestion control mechanism, in addition to the fast handoff support.

The above scenario demonstrates the type of active network services LARA++ aims to support. We will use this example as a reference scenario throughout this paper to illustrate the benefits and drawbacks of the LARA++ architecture.

1.2. The Aims of LARA++

The following bullet points list the primary objectives of LARA++, in no particular order.

[1] This active application temporarily optimises the routing to a mobile node's new network location until the mobile routing protocol converges.

1530-1346/01 $10.00 © 2001 IEEE

- *Flexibility*. Several proposed architectures trade-off generality of active computation for simplicity of the active node implementation and sometimes performance. However, we believe that most flexibility is required on routers located at edge networks, where a higher processing power to packet throughput ratio is possible. We therefore believe that flexibility is more important than performance, provided that performance of the architecture resides within reasonable bounds (e.g. sufficient throughput to support large LANs and small MANs).
- *Multiplicity*. We believe it is vital for multiple services to co-exist, even when operating on the same data stream. Active routers designed in accordance to the DARPA Active Node Architecture [3] tend to support only a few *distinct* Execution Environments (EEs). While this architecture suits the integrated approach to active networks, where active packets carrying code in-band are processed *once* by the corresponding EE, the more generic use of active networks discussed above calls for a more flexible architecture. The active node design must account for network packets being processed by *multiple* "active" programs or software components on the same active node.
- *Extensibility*. Many active router implementations lack flexible program integration and service composition mechanisms and hence restrict extensibility of active services. For example, most "conventional" architectures do not account for the application scenario described above, since they lack a flexible composition framework allowing multiple active applications operating together. Therefore, a sophisticated composition framework allowing multiple active programs or components to process network packets in a customisable sequence is required.

In summary, little emphasis has been placed on the problem of how to design truly flexible and extensible active nodes, nor how to make use of the benefits of component-based programmability (i.e. modularity, reusability, extensibility, etc.) for active networking.

2. Component-based Node Architecture

LARA++ provides a flexible and dynamically programmable platform for the development of active applications and network services based on the concept of service composition. It exploits the benefits of component-based architectures, namely code modularity, reusability and dynamic composition, to facilitate development and deployment of customized network services.

The motivation for our component-based approach arises from the fact that "conventional" active node architectures usually lack flexible extension mechanisms. While *integrated* active network architectures[2] usually limit capsule programmability because of their size restrictions and the fixed programming interface offered by their execution environment[3], *discrete* architectures usually restrict node programmability due to the lack of interoperability support for individual software components and service composition mechanisms.

LARA++ tries to resolve these limitations by providing a flexible composition framework for active software components. Yet, it enables active services to be split into many simple and easy to develop functional components. This "divide and conquer" approach simplifies extensibility of router functionality. It eases the component design and development, because only "small" programs with reduced functionality are built, and support for dynamic extensibility and replacement of individual components is provided. Furthermore, the flexible component approach facilitates development of customized and adaptive network services to better meet the requirements of end applications.

The remainder of this section provides a basic overview of the LARA++ architecture. Further details on the component architecture and an in-depth discussion of LARA++ internals are discussed in the LARA++ design report [4].

2.1. Node Architecture

LARA++ is constructed by layering active network-specific functionality on top of an existing router operating system. The architecture is *generic* in the sense that it can be implemented on any underlying router platform.

A key difficulty in designing active networks is to allow nodes to execute user-defined programs while providing reliable network services for everyone. An active node must protect co-existing network protocols and services from each other and securely control shared resources. The framework for our safety model is based on a layered architecture, where the top layer provides a safe execution environment for active user code. Figure 1 provides a graphical illustration of the LARA++ layered architecture:

(i.) The *NodeOS* provides a set of low-level service routines and system policies to enable controlled access to node-local resources and system services (i.e. device configurations).

(ii.) *Execution Environments (EE)* form the management unit for resource access and security policies, which

[2] Active packet (or capsule) based active network architectures are an *integrated* approach, whereas programmable switch architectures are a *discrete* approach.

[3] Note that current active packet based approaches do not support extension of the programming interface.

are enforced on every active program executed within an EE.

(iii.) *Processing Environments (PE)* provide the code space where active components with mutual trust relationships are processed.

(iv.) *Active Components (AC)* comprise the actual 'active programs'. They are processed within PEs alongside with other trusted components. The active code is executed by means of one or more LARA++ user-level threads.

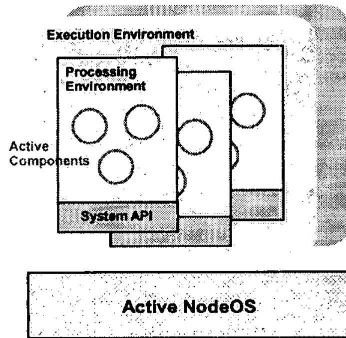

Figure 1. The layered active node architecture of LARA++

LARA++ introduces the concept of processing environments as an extra protection layer between the actual active programs and the EE. The idea behind the PEs is to apply the benefits of hardware supported protection mechanisms (i.e. virtual memory management and task switching) developed for modern operating systems to active networking.

2.2. Components

Components are the program units developed and configured in LARA++. LARA++ components are dynamically loadable onto active routers where they provide additional or extended services. Each component provides a useful service – either self-contained or as a result of composition.

LARA++ components can be either *active* or *passive*. Active components perform the 'active computations' based on one or more execution threads, whereas passive components provide merely static functionality that can be used by the active components (similar to support libraries). Components can be further differentiated by their function into *user components* and *system components*. User components are those active programs or libraries that are injected by users of the active network. System components, in contrast, are the components that constitute LARA++ and expose the API to low-level resources and system calls.

LARA++ user components have a minimal well-known interface similar to the **IUnknown** interface known from the Microsoft COM component model. It enables other components to query the fully exposed interface. Figure 2 illustrates this:

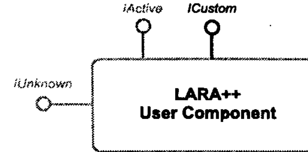

Figure 2. LARA++ Component Interfaces

Active components can be distinguished from passive components based on their **IActive** interface, which is used to initialise (**IActive.Initialize**) and run a component (**IActive.Main**).

LARA++ components are built like normal shared or dynamic link libraries. LARA++ specific functionality (for example, the LARA++ system API and user-level scheduler) is dynamically linked as support libraries when compiled. At component installation time, the *Component Loader (CL)* extracts the active component code from the link library and loads it into a processing environment.

Active components are distributed either in the form of pre-compiled machine code or as source code. In the latter case, the active code is Just-in-Time (JIT) compiled at component loading time.

2.3. Processing Environments

LARA++ processing environments provide a *safe,* and yet *efficient* execution environment for groups of active components maintaining a mutual *trust relationship.* Trust relationships are defined in terms of several criteria, but initially limited to code signatures and user authentication. Thus, depending on the code producer of a component and the user who installs it, the component might be regarded trustworthy for execution within the PE or not.

Although the focal point of the LARA++ PEs is in agreement with the execution environments defined by the DARPA active networking group, PEs are a specialisation with useful additional features.

First, PEs provide a *safe* code execution environment to protect the active node from untrusted machine code based on the principles of virtual memory management, pre-emptive multitasking and system call control. They define the 'visibility boundaries' for active code that prevent malicious code from breaking out of the protection environment.

Second, PEs are designed for *efficient* active code execution of trusting ACs. To improve the execution performance of active components, we developed a low latency user-level thread scheduler for the PEs. Thereby we exploit the advantages of trust relationships, namely

that trusted active components can be executed within the same protection environment without creating a security threat.

The LARA++ interface is dynamically linked to the PEs similarly to the case of system interfaces of conventional operating systems. However, since active components are dynamically loaded into the PEs, a dynamic binding mechanism for the LARA++ interface is required. We accomplish late binding through passing the LARA++ API handle to the active component when calling the `IActive.Initialize` function.

Moreover, the specialisation of execution environments into PEs also accounts for fine-grain use of *module thinning*[4]. Each PE can provide a specially customized system API depending on the trust relations between the active components being processed and the active node OS.

2.4. Active NodeOS

The LARA++ *Active NodeOS* provides access to low-level system service routines (for example, device configurations) and node-local resources. It controls access for high-level components based on their privilege level and the system policies defined. The Active NodeOS forms an independent layer of service on top of the router operating system. For performance reasons, however, the LARA++ NodeOS is tightly integrated with the router OS.

The core components of the LARA++ Active Node OS are the packet interceptor/injector, the classifier, the memory mapping mechanism, and the system trap control:

(i.) The *packet interceptor* and *injector* respectively are directly integrated with the network stack of the router OS. Depending on the active network service, data packets (or frames) can be intercepted (and re-injected respectively) either on the link-layer level or on the network-layer level.

(ii.) The *packet classifier* determines whether or not a packet requires active computation and by which active component(s). We developed a hierarchical, multi-stage classification mechanism allowing flexible ordering of packet filters (see section 2.5 for further details).

(iii.) A key technology within LARA++, developed to speedup data packet handling, is the *memory mapping mechanism*. It avoids expensive copy operations when passing the packet data into user space for the active processing and back down again by mapping the physical memory, where the packet data were stored by the network device driver, directly into the virtual address space of the active component(s) that have to process the packet (see Figure 3).

(iv.) The *system trap control mechanism* ensures node safety by restraining the system call interface. It checks whether system calls to the Active NodeOS (or the router OS) have been invoked through the LARA++ system API. As illustrated in Figure 4, malicious system calls trying to avoid the LARA++ system API (i.e. direct calls to the system API or bogus software interrupts) are strictly blocked.

[4] Module thinning secures the programmable system by individually tailoring the programming interface exposed to a software process based on the privileges of the user or program.

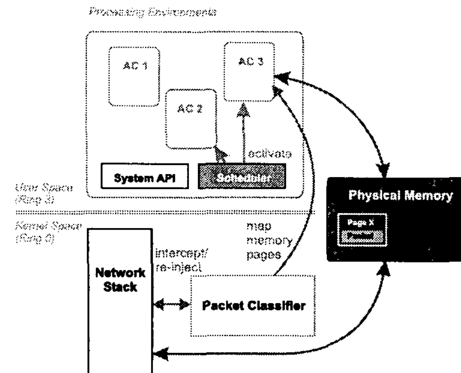

Figure 3. Zero-Copy Data Path: Packet memory is directly mapped into the virtual address space of the active component(s) processing the data.

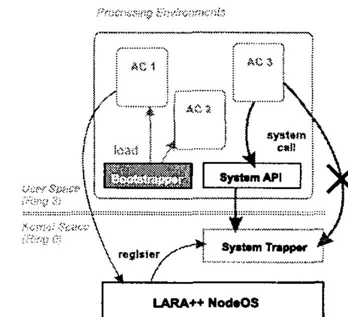

Figure 4. System Call Control Mechanism: Malicious system calls avoiding the LARA++ interface are blocked to ensure node safety.

2.5. Service Composition

The component-based approach proposed by LARA++ aims to be a highly flexible platform for dynamic network services. As further described here, service composition on LARA++ nodes is based on a sophisticated packet classification mechanism.

LARA++ services are composed out of many light-weight components where every component typically adds a small amount of functionality to the overall service. Ser-

vices are composed through insertion of packet filters into the *packet classifier*. Active components typically register their filters with the packet classifier at instantiation time (and if necessary again at run-time). The classifier determines based on the set of packet filters currently installed whether or not a packet demands active processing, which active component(s) are required and in which order.

Service composition within LARA++ is a co-operative process based on dynamic and conditional bindings. It is called *co-operative*, since various network users might be allowed (depending on the local security policies) to install packet filters matching the same data streams (or subsets). The fact that active services are composed through insertion or removal of packet filters (for example, when components are instantiated) makes the component bindings *dynamic*. Since the service composite depends on the actual data in the packets, the bindings are *conditional* (note that a binding is only in force if the filter for the binding matches the packet).

The packet classifier maintains a flexible representation, called the *classification graph*, to structure the packet filters of the active components instantiated on the router. Figure 5 illustrates a simplistic representation of the classification graph. Each node in the graph consists of a set of packet filters that define the sub-branches of the node. Depending on the node's properties, only the best match or multiple filters matches are considered.

In order to support flexible extension of router functionality based on active components, an "elastic" means to describe at which point in the processing chain a component must be inserted is required. For this purpose, we chose a simple notion to describe the graph structure which provides sufficient flexibility to incorporate new protocols and extend current protocol stacks.

The basic structure conforms to the TCP/IP layer model. It ensures that active components providing low-level services are processed before components dealing with higher-level computations. For example, network protocol options must be processed prior to transport protocol operations (see Figure 5). The fine-grain structure accounts for the layer-specific protocols. Extension headers in IPv6, for example, must be processed in a predefined order.

The packet classifier filters incoming and outgoing data packets based on the classification graph. The classifier traverses the graph starting from the root node. In the case of an IPv4 packet, for example, it continues the search following the **/netin/ipv4** path. If the search hits a node where none of filter matches, the default filter defines how to continue. Note that the search space defined by the classification graph can often be reduced by collapsing nodes with only few sub-branches into a single node through multi-field classification.

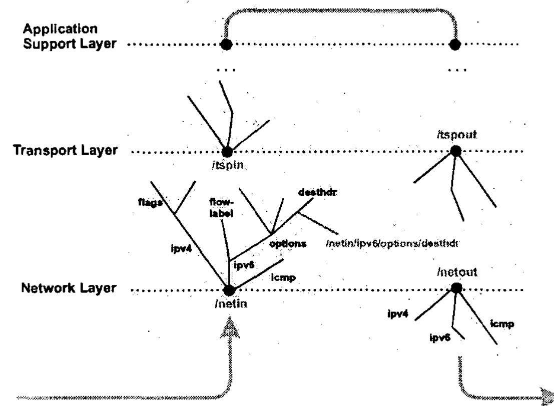

Figure 5. ***The Classification Graph.*** **The grey arrow present the general route data packets take when passing through a LARA++ node. The network layer part (input and output paths) of the classification graph shows how the classifier organises the packet filters for the ACs.**

A packet that is matched by a filter is passed to the corresponding active component. After completion of the active processing, the classifier continues at the same point in the classification graph or an optionally specified point (defined by the filter) in the classification graph[5].

As a demonstration on how the LARA++ classifier operates, we refer back to the example scenario introduced in section 1.1. The first application, namely the local congestion control mechanism, requires access to all IPv4 packets marked by use of a particular IP option. A packet filter matching the respective mark is defined and inserted into the classification graph. Since the congestion control mechanism drops packets dependent on the class of service in the case of congestion, it is best done early on the incoming path (before much processing is done). A suitable place to insert the packet filter would be the place where IP options are processed, namely **/netin/ipv4/options** or **/netin/ipv6/options**.[6] The second example application, namely the handoff optimisation service, redirects packets sent to a mobile node's previous destination to its actual destination after a network handoff. A recommended place to insert the packet filter for this service (which filters packets based on a specific IPv6 destination address) is where the IPv6 protocol is processed, namely **/netin/ipv6**.

[5] Availability of this option depends on the user privileges and filter type (i.e. protocol or flow specific).

[6] Note that an AC processing a certain protocol field must not necessarily be processed at the point in the packet processing chain where the respective field would normally be processed; the location depends on the processing task only.

These example applications illustrate how independent active services (developed by different code producers and installed by different users) can co-exist, and yet inter-operate on LARA++ routers, even without knowledge of one another.

3. Elementary Design Evaluation

This section presents early evaluation results regarding the additional protection layer provided by the processing environments.

Despite the benefits of safe execution of potentially malicious active code, the PEs introduce a processing overhead. First, network data passing through a LARA++ router must be passed to the ACs for computation. In order to avoid expensive memory copy operations (which many user-space active node implementations incur), we have developed special kernel-level support, which efficiently maps data packets from the network device memory into the virtual address space of the respective PEs (see section 2.4). Second, as the PEs provide a safe execution environment for many concurrent ACs, additional scheduling overhead emerges for active threads. To minimize the performance hit resulting from active thread scheduling, we have implemented a lightweight user-level scheduler for the PEs (see also section 2.3).

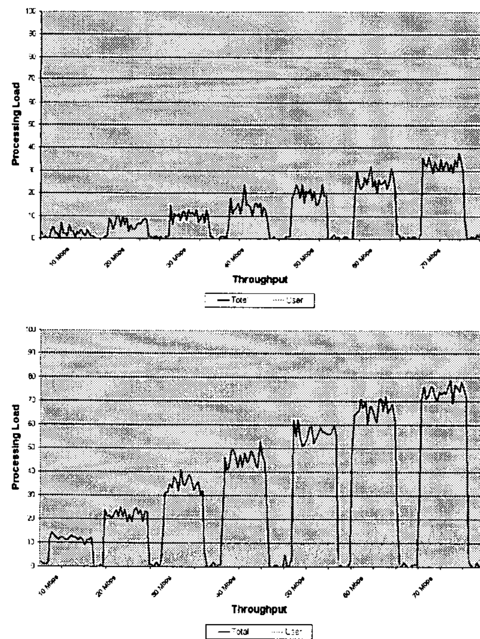

Figure 6. Comparison of the processing load – total and user load – on a standard Win2K router while forwarding packets (top), and our Win2K LARA++ router implementation, which performs the minimal house keeping to receive and forward all packets by an AC (bottom).

The first set of experiments estimate the performance penalty arising from the memory mapping mechanism. For this, we compare the processing load on our MS Windows 2000 based router in two cases. First, without LARA++, packets are simply forwarded by use of the default routing mechanism. Second, with LARA++ support, every packet passing the node is intercepted by the active NodeOS and mapped into the virtual address space of the PE executing our test component. The test component performs the minimum amount of house keeping required to receive and send packets, before the NodeOS un-maps and finally forwards the packet as in the first case.

Figure 6 shows that mapping network traffic demanding active computation (which is likely to be a small percentage of the overall traffic) into user space only approximately doubles the processing load on the router compared to default forwarding.

The second set of measurements (Figure 7) show the performance gain arising from our lightweight AC thread scheduler. The cutback of scheduling overhead benefits the execution of trusted active components within a single PE. The results show that the AC scheduler performs context switches between active threads approximately one order of magnitude faster than the Win2K switches between normal user threads. Even context switches among kernel-level threads are about 5-6 times more expensive.

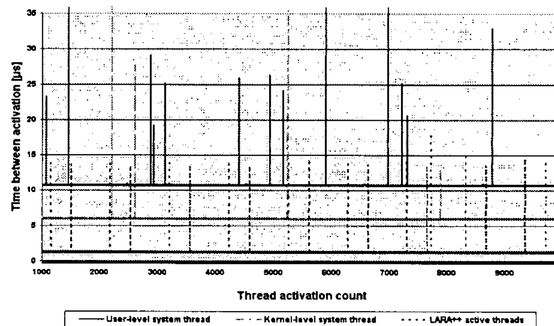

Figure 7: The graph compares the context switch times between different threading approaches. Three experiments each measuring the times between activation of a thread while competing with a second thread of the same kind are shown. Win2K user-level threads take about 10.5µs to switch a thread out and in (inclusive housekeeping); LARA++ active threads require only approx. 1µs (based on PII-233 MHz).

The performance measurements presented here are limited to our LARA++ implementation for Win2K, simply because it was more advanced than our Linux implementation at the time of experimentation.

4. An Example – Local Congestion Control

This section demonstrates by means of an example application how LARA++ active components are developed. For this, we recall the local congestion control example introduced at the beginning.

The following code fragment demonstrates how LARA++ components are programmed and shows how the API is used. The **ACMain** function is called when the active component is instantiated. After the registration of the active component with the active NodeOS and the packet filter(s) with the classifier, the active thread loops until the component terminates. In the while loop, the component blocks on the **LReceivePacket** call until it receives packets that match the packet filter (i.e. include the respective class of service mark) and sends or drops the packets depending on the congestion condition and the class of service.

```
ACDLL_API int ACMain(void)
{
 [Initialise variables & def. packet filter(s)]
 if (LRegisterAC(&ACInfo) == LARA_FAILURE)
  return LARA_FAILURE;
 while (Run) {
  pLaraPacket = LReceivePacket(&ACInfo);
  pBuffer = LGetPacketBuffer(pLaraPacket, &bufLen);
  pIPHeader = pBuffer + sizeof(TEthernetHdr);
  // get output interace
  OutputIF = LRouteLookUp(pIPHeader);
  // get current length of output queue
  QueueLen = LGetQueueLength(OutputIF);
  if (pIPHeader->Type == IPV4)
   CoS = GetCoSMarkIPv4(pIPHeader);
  else [...]
  if (QueueLen < THREASHOLD_PREMIUM)
   // no congestion
   LSendPacket(&ACInfo, pLaraPacket);
  else
   if (QueueLen < THREASHOLD_GOLD && CoS == PREMIUM)
    // output queue fills; service premium CoS
    LSendPacket(&ACInfo, pLaraPacket);
   else [...]
   else
   // output queue full, drop low CoS packets
   LDropPacket(&ACInfo, pLaraPacket);
 }
 LUnregisterAC(&ACInfo);
 return LARA_SUCCESS;
}
```

5. Related Work

Many active network architectures provide safety for mobile code execution through specialized script-like languages (for example, NetScript [5], PLAN [6] and NFL [7]), type safe languages such as Java or Caml [8], or even Proof Carrying Code (PCC) [9]. LARA++, in contrast, allows users to distribute active programs in the form of binary programs or as source code. Safe active code execution is ensured by the use of processing environments especially designed for this purpose.

Unlike active network solutions based on in-band active capsules, such as ANTS [10], LARA++ does not rely on a specialized protocol and service supported homogenously by every node in the network. LARA++ routers can be deployed isolated on strategic beneficial locations (for example, as boarder routers), and its filter-based classification approach enables LARA++ routers to apply active computations transparently to any kind of packet passing the node.

The *x*-kernel [11] and Scout [12] provide a framework for composing network protocols. Router Plugins [13] and Click [14], which both propose a modular router architecture, share many communalities. Central to all these systems is a configuration graph of processing nodes (or modules) that defines the path for packets passing the router. In much the same way, LARA++ uses the classification graph to compose active services from individual components. However, unlike these flexibly configurable router architectures, LARA++ focuses more on the aspects of dynamic programmability through various network users.

The Bowman active NodeOS and the CANEs execution environment [15] is another related approach. Bowman however does not incorporate any form of run-time safety and security. Comparable to LARA++, CANEs provides a composition framework for active services based on the injection of active code at specific points, called *slots*. While LARA++ uses a flexibly extensible graph structure in order to describe at what point in the processing chain an active component needs to be processed, a generic "underlying program" defining the slots is used to customize services within CANEs. While the Georgia Tech solution clearly divides the functionality of the NodeOS and EE into two elements, LARA++ incorporates both into a single architecture, based on the principle that the high cohesion between these elements benefits performance, and the loose coupling between LARA++ components improves flexibility.

6. Future Work

The main focus for the near future lies on completion of our LARA++ implementations for MS Windows 2000 and Linux. An experimentation implementation for Win2K is already available, whereas a prototype implementation for Linux is under way.

As LARA++ provides the basis for several active and programmable network projects at Lancaster, further work will involve the implementation and evaluation of active services supporting mobility, QoS and multicasting. In parallel to the active node implementation, we are currently also working on the development of several active components to support fast network handoffs and nomadic communication in Mobile IPv6 environments.

7. Conclusion

This paper has introduced LARA++, a novel component-based architecture for active network nodes. We have examined how its component-based design benefits flexibility with regard to programmability and extensibility.

We have shown that LARA++ surpasses "conventional" active platforms in many regards: *(i.)* The component-based design eases development of lightweight components, facilitates re-usability of code components, increases customisability of service composites, and enables gradual extensibility of functionality. *(ii.)* Support for co-operative composition of active services enables independent user groups to participate in the programming process of the active node in a controlled fashion. The LARA++ composition framework enables flexible extension of functionality at any point in the packet processing chain of a network node. *(iii.)* The extra protection layer, referred to as processing environments, extends the active network architecture by a safe execution environment for potentially malicious active code based on the principles of virtual memory and task scheduling, and a secure system interface. *(iv.)* A novel kernel-level system extension, described as the memory mapper, provides fast network access for user-level active computations.

Finally, we have demonstrated how LARA++ facilitates active programming through the support of *(a)* standard user-space development tools and programming languages (without trading off safety); *(b)* dynamically loadable user libraries enabling extensibility of the programming interface; and *(c)* late-binding mechanisms for software components accounting for dynamic and flexible service composition. Based on a small example active component, we have shown how to implement a simple congestion control mechanism.

8. Acknowledgements

The LARA++ work presented here was undertaken as part of the Microsoft funded LandMARC collaboration between Lancaster University, U.K. and Microsoft Research, Cambridge U.K. [16].

9. References

[1] R. Cardoe, et al., "LARA: A Prototype System for Supporting High Performance Active Networking", In Proceedings of IWAN '99, Berlin, Germany, June 1999.

[2] S. Schmid, J. Finney, A. Scott and D. Shepherd, "Active Component Driven Network Handoff for Mobile Multimedia Systems", in Proceedings of IDMS '00, Enschede, The Netherlands, October 2000.

[3] K. Calvert, "Architectural Framework for Active Networks", Active Network Working Group Draft, July 1998. Work in progress.

[4] S. Schmid, "LARA++: Design Specification", Lancaster University DMRG Internal Report, MPG-00-03, January 2000.

[5] Y. Yemini and S. da Silva, "Towards Programmable Networks", IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Italy, October 1996.

[6] M. Hicks et al., "PLAN: A Programming Language for Active Networks", http://www.cis.upenn.edu/~switchware/ papers/plan.ps, 1998.

[7] Y. Yemini et al., "The Network Flow Language: A Mark-based Approach to Active Networks", Technical Report, Columbia University Computer Science Department, submitted, July 1999.

[8] Caml Home Page. http://pauillac.inria.fr/caml/index-eng.html.

[9] G. Necula, "Proof-carrying code", in Twenty-Fourth Annual ACM Symposium on Principles of Programming Languages, New York, January 1997.

[10] D.UJ. Wetherall, J.V. Guttag and D.L. Tennenhouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols", IEEE OPENARCH '98, April 1998.

[11] L. Peterson, S. Karlin and K. Li, "OS Support for General-Purpose Routers", in HotOS Workshop, March 1999.

[12] D. Mosberger and L. Peterson, "Making paths explicit in the Scout operating system", In Proceedings of 2nd Symposium on Operating Systems Design and Implementation (OSDI '96), pages 153--167, October 1996.

[13] D. Decasper et al., "Router Plugins: A Software Architecture for Next Generation Routers", SIGCOMM '98, Vancouver, CA, September 1998.

[14] R. Morris, E. Kohler, J. Jannotti and M.F. Kaashoek, "The Click Modular Router", in Proceedings of the 17th ACM Symposium on Operating Systems Principles, pages 217--231, December 1999.

[15] S. Merugu et al., "Bowman and CANEs: Implementation of an Active Network", 37th Allerton Conference on Communication, Control and Computing, September 1999.

[16] "The LandMARC Project", available via the Internet at http://www.LandMARC.net.