Ease of Network Programmability based on Active LARA++ Components

IEE Savoy Place, London
November 2000

Stefan Schmid, Doug Shepherd
{s.schmid,d.shepherd@comp.lancs.ac.uk}
Lancaster University

http://www.LandMARC.net/

Overview

- Motivation
- LARA++
  - Overview
  - Architecture
  - Design Objectives
- Ease of Active Programmability
- Performance Optimisations
- Conclusion

Motivation (1)

- Active Code Execution

User Space / VM	Kernel Space
▪ Ease of Active Programmability ▪ Simplifies Safety	▪ Performance

- Trade-off?

$$\text{Ease of Use} \longleftrightarrow \text{Performance}$$

What is the Problem?

Current User Space/VM Implementations:

- Virtual AN Architectures use socket interface
- Active Routers copy packets (up & down)

⇨ **Performance hit through multiple <u>copy</u> operations**

Proposed Solution:

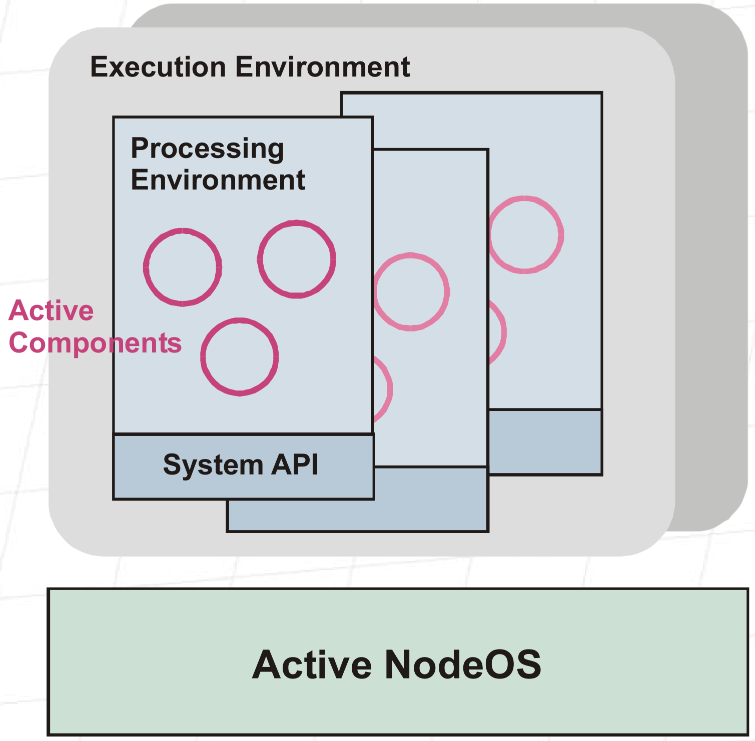
Memory mapping physical memory in user level virtual address space

LARA++ Overview

- Second generation active router architecture
- Programmable platform supports service composition based on small components
- Active Components are …
  - dynamically (un)loaded onto LARA++ routers
  - extending the functionality on the router
  - flexibly integrated into packet processing chain

LARA++ Architecture

- Layering active network specific functionality on top of node OS
- Safety and security is achieved by a four-layer architecture
- PEs provide process-like protection for active code
- LARA++ implementation is split across kernel & user space



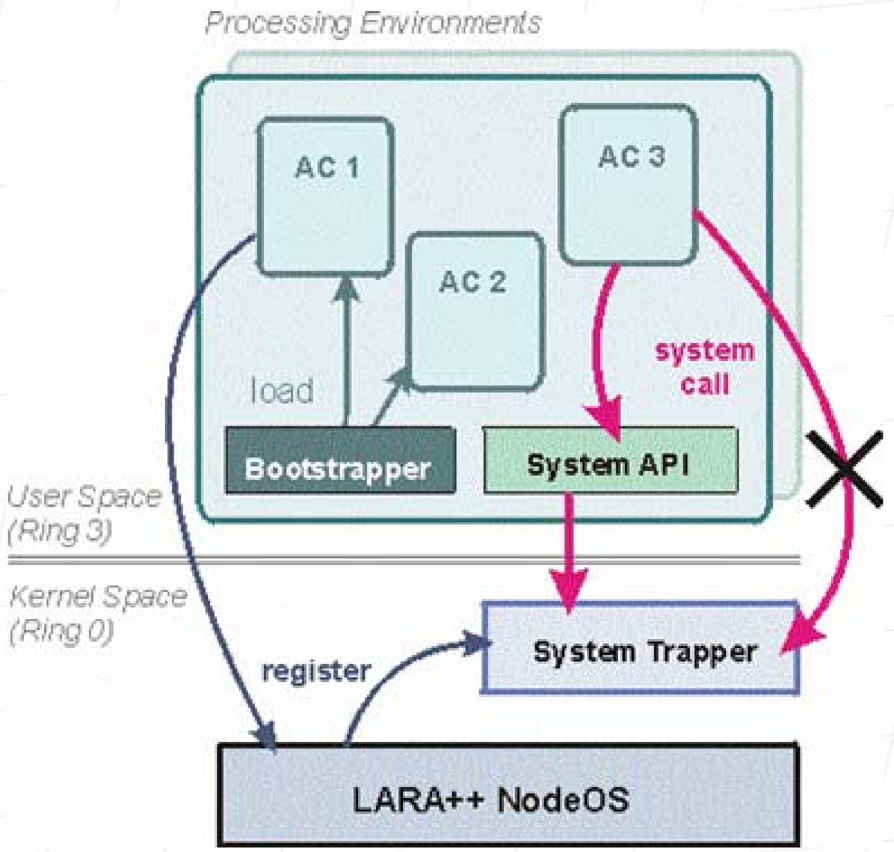
Core Design Objectives

- High Performance
  - Native active code processing
  - Fast data packet handling
- Ease of Component Development
  - Developing ACs based on standard tools
  - Flexible composition framework (allowing development of “small” AC)

Performance Optimisations

- Active Code Processing
  - Native code execution rather than interpretation
  - AC executed within PE like shared or dynamic link libraries
  - Node safety based on sandbox
- Data Packet Handling

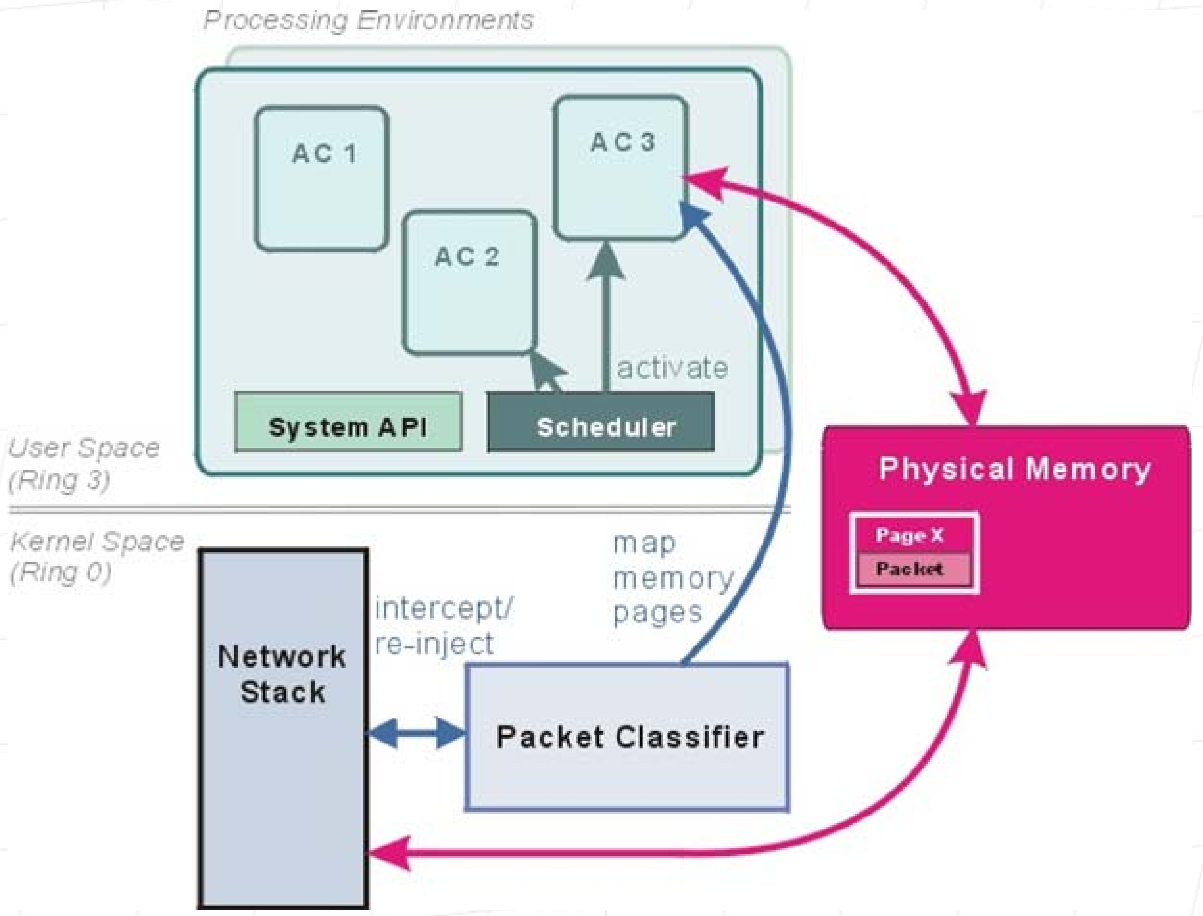
System Call Control



Performance Optimisations

- Active Code Processing
- Data Packet Handling
  - Zero-copy packet handling
  - Memory mapping of packet memory into PE virtual address space
  - Processing load approx. doubles (with no optimisation)

Data Packet Handling



Performance Optimisations

- Active Code Processing
- Data Packet Handling
  - Zero-copy packet handling
  - Memory mapping of packet memory into PE virtual address space
  - Processing load approx. doubles (with no optimisation)

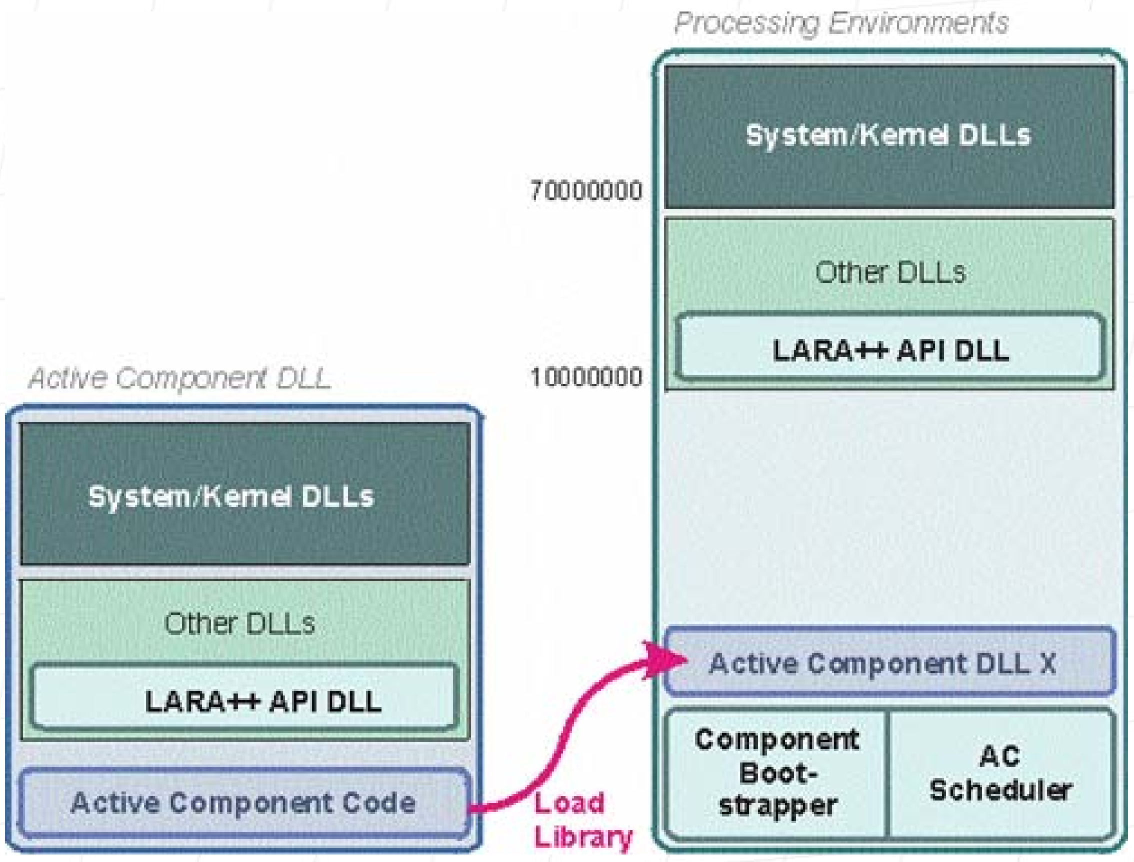
Component Development (1)

- Convenient user space programming
- Standard languages
- Standard tools (compiler/debugger/IDE)
- Active Components are built like “normal” shared/dynamic link libraries
- LARA++ API is linked to Active Component code like “standard” libraries

Example AC Code

```
ACDLL_API int ACMain(void)
{
   [Initialise variables and define packet filter(s)]
   if (LRegisterAC(&ACInfo) == LARA_FAILURE)
        return LARA_FAILURE;
   while (Run) {
        pLaraPacket = LReceivePacket(&ACInfo);
        pBuffer = LGetPacketBuffer(pLaraPacket, &bufLen);
        pIPv6Header = pBuffer + sizeof(TEthernetHeader);
        [Packet processing]
        Status = LSendPacket(&ACInfo, pLaraPacket);
   }
   LUnregisterAC(&ACInfo);
   return LARA_SUCCESS;
}
```

Active Component Code



Component Development (2)

- Component Debugging and Testing
  - “Minimal” LARA++ Node OS support can be installed on Development Machine
  - Debug Processing Environment provided

⇨ Active Components can be debugged like “normal” applications

Conclusion

- User space active processing simplifies programming and safety
- Performance trade-off for user space active processing can be minimal
- LARA++ achieves …
  - high performance through native code execution and fast memory mapping
  - ease of active coding based on standard programming languages and development tools