A Highly Flexible Service Composition Framework for Real-life Networks

S. Schmid, T. Chart, M. Sifalakis, A.C. Scott
Distributed Multimedia Research Group
Computing Department
Lancaster University, U.K.
{sschmid, chart, mjs, acs}@comp.lancs.ac.uk

Abstract. This article introduces a novel approach for service composition on active and programmable routers. The proposed composition framework enables flexible programmability of a router’s data path through dynamically loadable software components, called ‘active components’. The composition framework promotes transparent, dynamic and incremental deployment of network-side functionality which renders active network technologies capable of fully replacing the existing routing infrastructure by supporting all the legacy network protocols and services, and the integration of novel ones. On the other hand, the composition framework provides the necessary structures and mechanisms to deal with services composed by independent users potentially unaware of each other. The latter is particularly important as network routers are regarded as shared resources.

In order to evaluate the proposed service composition framework, we have developed an implementation for the LARA++ active router architecture [1] and deployed several of these active routers in a real-world service network. The evaluation demonstrates the flexibility of the composition framework by enabling the alteration of service composites at run-time and provides scalability in the generation of such composites by users or administrators. This work also reports on lessons learned while deploying several active services in a production network. Besides a qualitative evaluation of the service composition framework and the LARA++ architecture, we provide a number of quantitative results, which show that the overhead of our composition model does not adversely affect the performance of the router, despite the increased flexibility achieved.

1 Introduction

In recent years, many diverse and variously-focused frameworks sporting elements of active network solutions have emerged and established themselves. The majority of these platforms address only a subset of the issues that determine their utility outside a laboratory environment, resulting in most being inflexible, poorly performing, not scalable or insecure. Few active network solutions have considered the importance of a proper service composition model within real-life network environments. Current systems therefore provide only limited flexibility for the composition of network services. Yet, in order for active networking to be considered a suitable technology for wide deployment in future inter-networks, these issues must be addressed. Furthermore, to be successful, active and programmable network architectures must be sufficiently flexible, extensible and generic to accommodate a diverse range of not only current services, but also future ones that have yet to be conceived. The importance of

1 This research is funded by the EPSRC (grant number GR/R31461/01).
flexible and generic programmability is emphasized by the fact that routers are shared resources, providing network services for end-users with different application and service requirements.

A recent study by Hicks and Nettles [2] has revealed that most of today’s active router platforms lack sufficient flexibility to allow for true evolution. Their study shows that many modular or ‘plug-in’ based architectures that advertise extensibility, actually limit the scope of future changes by providing fixed, pre-defined interfaces. Hicks and Nettles suggest that true extensibility should not be limited to a fixed set of modules or plug-ins, but rather should allow modification and replacement of all components contributing to a service composite.

Our own evaluation of current active router platforms agrees that most of them fall at this hurdle. It turns out that many platforms are built with only a limited range of services in mind, for example, network management [3,4], signalling [5], and multicast [6,7]; Others fail to provide the necessary flexibility or extensibility to dynamically deploy new types of services, not considered by the developers, for example, Scout [8], Click [9] and Router Plugins [10]. Despite these shortcomings, we believe the most neglected requirement in today’s active router architectures is support for appropriate service composition mechanisms that enable the installation and management of a diverse range of services by users that may be unaware of each other.

Our work proposes a novel approach to service composition on active routers that provides the necessary semantics to control the interaction of different network services and allows independent users to partake in the composition process of the overall service provided by a router. The challenge is to find a way of asserting this cooperation over the same stream of packets without causing undesired interference between the services. Our composition framework has been evaluated through experimentation in a real edge network with several network services as shown in section 6.

The remainder of this paper is organised as follows: Section 2 gives a brief overview of the background of this work. Section 3 continues with a detailed description of the service composition model. Section 4 and 5 describe the prototype implementation of the LARA++ classifier and the measurements made from that prototype. Section 6 describes three evaluation services for our service composition model that have been developed, deployed and tested in a real-life service network. Chapter 7 compares the composition model with other architectures. And finally, we conclude in section 8 by summarising the results of this study.

1.1 Motivation

Before we describe the proposed service composition framework, we outline some scenarios that adequately encompass many of the problems of service interaction and composition on programmable nodes. Scenarios such as those described in this section precipitated the desire to provide a highly flexible composition model.

Suppose an active node has several ‘active components’ installed, as depicted later in Figure 2. One of the components might be a Web cache component that could be installed by a network or system administrator to enforce Web caching for all users (in order to reduce off-site traffic). However, it is also possible that a Web cache is re-
quired by an ordinary end user, who simply wants to benefit from the caching effect of frequently visited pages. While in the first case, the Web cache component would have to process all Web traffic traversing the node, in the latter case, only traffic that is related to the specific end-user’s network flow should be processed. This scenario depicts the need to flexibly filter and associate traffic with ‘active components’.

A second component providing a network address translation [11] service might be installed alongside the Web caching component. Since the NAT service is only required for off-site traffic it should logically be placed “after” the Web caching component, on the egress path of the router. In other words, Web traffic serviced locally through the cache should be processed by the cache component first, as there may not be a need for NAT. This example shows the need to “structure” the order of active services simultaneously installed on a node to optimise performance and more importantly to avoid any potential problems caused by undesired interactions.

Finally, a component installed on the active node may provide firewall services for egress traffic. To ensure no undesired traffic leaves the edge network, not even traffic originating from a malicious component installed on the active router itself, such a component would have to be called subsequently to all user-defined components. That is because only after all un-trusted components have executed can the system ensure that no undesired traffic leaves the node. This example depicts the need to control the processing order of components. Moreover, it shows there is a need to authorise the overall service composite based on user and component privileges.

The problem of service composition is principally one of managing competition and co-operation between components in the processing of packets. The LARA++ service composition model described in this paper provides the structure required for a distributed (i.e. involving more than one entity) and dynamic (i.e. allowing service composition at runtime) composition model without affecting flexibility, programmability or security on the active node.

2 Background – The LARA++ Architecture

LARA++ [1] is a software architecture that evolved from the predominantly hardware oriented Lancaster Active Router Architecture (LARA) [12]. The software architecture is designed to augment commodity operating systems with active network capabilities. Implementations currently exist for Microsoft Windows XP, and Windows CE. Porting to other platforms, while time consuming, is expected to be relatively straightforward.

LARA++ embodies a framework that exposes a programmable interface, which allows active programs (also referred to as active components in this context) to provide network and end-user services on LARA++ nodes. While low-level functionality of the architecture is directly integrated with the router operating system in order to maintain good performance for active processing, high-level execution environments, referred to as processing environments (PE), provide the necessary safety measures required for the execution of dynamically loadable active components. Moreover, a well-known programming interface exposed by the processing environments enables uniform component development across LARA++ platforms developed for different router systems.
LARA++ treats an active node as a resource shared by all of its users. The extent to which a node is programmable by each individual user (or group of users) is configurable by the node administrator. By means of node-local policies, the administrator can restrict access to packets (or flows of packets) that traverse the node and the set of resources that are available to each active component installed on the node. In addition to the security provided by resource and access control, users are protected from each other’s actions through the isolation of the processing environments. Because this safety is provided at the operating system level, LARA++ does not need to restrict programmability using language features like dynamic type checking, or by requiring use of type-safe languages such as Java or Caml [13], although such features offer additional protection to threads running in the same PE.

In order to enable the transparent interception of packets passing through the node, LARA++ is hooked directly into the host operating system. After processing, packets are re-injected back into the host operating system. The placement of these hooks is crucial for the interoperation of LARA++ active components with the conventional network services of the node. Since packets are intercepted and then re-injected, they can still pass through the host routing and forwarding engines. This allows LARA++ to flexibly augment the functionality of the node’s conventional network services. Although not a necessity that packets use host-provided routing and forwarding procedures, since these functions can be fully supported by LARA++ components, it enables lightweight augmentation of existing network services and allows for gradual replacement of conventional router functions.

Another generic feature of LARA++ is that it does not limit the way in which code is loaded onto active routers. Any component with the authority to instantiate other components may do so. This flexibility allows code to be instantiated in various ways, namely from a local copy of the component already held on the active router, by explicit out-of-band code loading techniques, or using in-band code directly from packets traversing the node. As part of the LARA++ project, we have developed a generic active service discovery and deployment protocol (ASDP) [14], which is an open “pluggable” protocol supporting user-definable out-of-band code loading.

The main contribution of the LARA++ active router architecture is the novel, service composition model discussed throughout this article. This model enables service components from many sources, including users, administrators and applications, to be merged into a single composite service. Figure 1 provides a conceptual overview of the proposed approach. The vision of the LARA++ platform is to provide a framework upon which complete router functionality can be provided as individual components (for example, a firewall component, a NAT component, a routing component, etc.), which are then composed into high-level network services at runtime. Componentisation has the advantages that it allows a “divide and conquer” approach to be employed for complex functionality, and that software components can be dynamically extended and replaced due to their well-defined interfaces.

Service composition is achieved through packet (re-)classification. Each component that is to become part of the service composite installs packet filters into the nodes of an extensible directed graph, referred to as the classification graph. The classification graph defines the structure and semantics for binding active components together in
a meaningful way; in other words, providing the ‘glue’ for the composition of the overall service. Packet filters inserted by active components into the nodes of the classification graph specify the kind of packets a component wishes to process.

This model of service composition offers several major advantages over other active router classification systems. The use of a dynamically configurable and extensible classification graph allows LARA++ to intercept and process packets on any type of packet-based network, from standard IP networks, through networks with ANEP-based active packet encapsulation [15], to bespoke signalling networks. LARA++ is simply configured with an appropriate classification graph for the types of packets intended to be processed by the node administrator.

The filter-based packet interception, in conjunction with the classification graph, render LARA++ a platform on which active components with overlapping interests, in terms of the packets they wish to handle, can operate without disrupting each other. For example, if one component installs a filter to express an interest in all Web traffic (i.e. all TCP packets with destination port 80), and another expresses an interest in all outbound TCP traffic in order to provide NAT, the classification graph delivers packets that match both filters to both components in the order defined by the graph. Therefore, service composition on a LARA++ active router is defined implicitly by the classification graph, and it can be exploited as a medium to manage both co-operation and competition between active components.

3 LARA++ Composition Model

The LARA++ composition model plays a central role in the overall architecture, as it provides the foundation for the flexible, extensible, and dynamic component-based development of active services.

Service composition on LARA++ nodes is carried out on two levels – the service level and the component level:

- At the service level macro composition is supported via a filter-based composition model. Active components dynamically integrate themselves into node-local service composites by inserting packet filters into the classification graph (using a system call to the NodeOS API).
- At the component level micro composition is supported using an explicit, lightweight composition model, which enables the construction of active services from pre-existing passive components.

In much the same way as normal user libraries, passive components export functionality that can be directly “glued” together to provide a new service. Since micro composition models are already well understood, and widely used for service composition in conventional component systems, the remainder of this section focuses on the macro-level composition model. The latter is of particular interest, for active networks as composition takes place right on the forwarding path.

Macro composition within LARA++ is largely packet driven. Depending on packet content, a different composite service may be used for the processing of that packet. The packet classifier plays a key role in the service composition process. A set of packet filters currently installed in the classifier determine whether or not a packet passing through the active router requires active processing, which active component(s) will process it, and the order in which processing should take place. The classification graph, which is managed by the packet classifier, maintains the key data structures for the composition framework. It organises the packet filters of the active components according to their computational function, and thus, provides the basis for the classification process. The following sections describe each of these elements in more detail.

3.1 Packet Classifier

The packet classifier defines a “route” through the active component space (see Figure 1) for packets traversing a node. Figure 2 presents an example classification graph.

The classification process starts at a root node, typically a unique device, for example, Ethernet in Figure 2. At each node in the graph the packets traversing the node are checked against the packet filters installed there (if any). Packets matching a filter are intercepted and delivered to the corresponding active component. After completion of
the active processing, the classifier continues classification from the same point (or optionally, a point specified by
the packet filter\(^2\)) in the classification graph. After all active components interested in the packet in a classification
node have processed the packet, it follows one of the “default” paths specified by graph filters (which inter-
connects the nodes in the classification graph) and continues the multi-stage classification process there. Finally,
when the packet finishes traversing the whole classification graph it gets forwarded to the next hop router. Using
this model, the whole functionality of the network stack can be re-implemented (and dynamically extended) as a
set of active components scattered at various nodes of the classifier.

3.2 Packet Filters

Packet filters are extremely flexible from the developer’s point of view as they allow detailed description of the
packets that will be processed by each component. Packets can be described using a powerful pattern matching
mechanism as well as by performing calculations based on information in the packet. Once a packet is matched,
the filter specifies how it gets directed further in the classification graph or sent to components for processing.
The packet classifier distinguishes two main types of packet filter: active component filters and graph filters.
Figure 3 illustrates how these filters are used within the classification graph. Active component filters are used by
the components to specify the packets of interest. These filters are typically registered with the packet classifier at
component instantiation, or at run-time if necessary using the LARA++ API. Graph filters, in contrast are used to
define the structure of the classification graph.

Packet filters installed by active components can be also devided into general filters and flow filters. Since a single
active component can install multiple filters, and given that there may be many components running on an active
router, it would be very costly to check all these filters. Flow filters (a specialization of general filters) have been
introduced to allow the number of filters to scale well. Flow filters are always bound to a specific user\(^3\) flow. They
have the advantage that they can be looked up instantly in a hash table based upon the flow characteristics of the
packet. Due to the hashing technique used to lookup flow filters, a LARA++ router can handle large numbers of
these filters with little impact on performance (see results in section 5).

Filters consist of rules that can be logically combined to build up complex packet descriptions. A rule is used to
define patterns in a packet which the classifier looks for, and are described by a four-tuple of \{packet offset, bit
pattern, bit mask, pattern length\}. The specification of packet filters may be facilitated by ‘well-known’ reference points and pre-defined tags. For example, a filter for HTTP traffic might use the
TCP\_HEADER reference point: \{TCP\_HEADER + TCP\_PORT, 0x0050\(^4\), 0xffff, 2\}; this is useful for

---

2 Availability of this option depends on the user privileges and filter type.
3 A user (or end-to-end) flow is defined by the source, destination, or both end-points. An end-point may be identified by
packet fields such as the network layer addresses and transport layer ports, or any other flow labeling techniques.
4 0x0050 (80 decimal) is the TCP port to which HTTP traffic is directed.
variable sized headers. Other types of rules evaluate that packets have taken a certain route through the classification graph, or that an arithmetic condition is be met (e.g. for the sum of specific packet fields).

**Fig. 3.** Active component filters select packets for active processing, whereas graph filters define the structure of the graph.

Graph filters and privileged active component filters may also define a *filter output node*, which provides an alternative *cut-through* route or path through the classification graph. Classification of packets matching a filter resumes at the corresponding filter output node after the packet has been processed by the active component.

The whole classification system is protected from misuse by a permission-based security scheme. For this reason, active component filters require the following properties: (i) an *access property* to express the packet access permissions required by the component, which may be read-only, read-write or write-only; and (ii) a *principal* or security credential that specifies the network user and possibly the code producer, on whose behalf the filter is installed. Based on the access property, principal of a component, associated privileges, and node-local security policies, the classifier decides whether or not to authorise the insertion of a filter.

---

**Fig. 4.** Filter mark-up to intercept HTTP requests carried by an IPv4 transport.

Packet filters are easily specified and installed by active components using an XML-based mark-up language that supports the complex nature of these filters (example in Fig. 4). The function of filters specified using this mark-up
is easily discernable, but in contrast to other high-level languages such as ANQL [16], our filter mark-up language does not depend upon a pre-defined or standardised naming scheme for packet fields.

3.3 Classification Graph Table

The classification graph table (CGT) is an abstract description of the classification graph in a form that can be communicated to other LARA++ nodes. A simple and extensible way of describing classifications nodes (e.g., IPv4, TCP, and UDP), and the branches of the graph (IPv4→TCP or IPv4→UDP) has been introduced. The basic structure of the classification graph described in the CGT conforms to the TCP/IP layer model, which ensures that active components providing low-level services are processed before components dealing with higher-level computations; a typical example being network protocol options that are typically considered before transport protocol headers. However it should be noted that the CGT is not necessarily an 1-to-1 representation of the classification graph but rather an abstract one that enforces a minimum common reference structure among LARA++ nodes. As a result, the actual classification graph locally in a node will often be more complex in order to satisfy the local node requirements (e.g. ordering of active components).

Since the CGT is expected to change over time (although changes need not be often), as nodes may be introduced, an automated mechanism is needed for communicating the CGT across a complete active network.

3.4 Composite Characteristics

Service composition within LARA++ is a co-operative process as it allows independent network users to install active components that match the same data streams or subsets of streams and is controlled by the local security policies in force. The classification graph provides the means by which independent users can integrate new active functionality or services in a “meaningful” way without knowledge of other users’ active components.

The fact that active services are composed by the insertion or removal of packet filters at run-time, as components are themselves instantiated or removed, makes the composition process highly dynamic. Since the service composite depends on the actual data in the packets (that match the filters), component bindings are conditional. Service composition within LARA++ is therefore a process that takes place on a per-packet basis.

4 Implementation

This section briefly outlines the current implementation of the LARA++ composition model, and highlights some of the implementation decisions that affect the overall performance.

4.1 Design Overview

The packet classifier is the central subsystem of the LARA++ architecture. It is responsible for dispatching filtered
packets to active components. When the classifier matches the filter owned by an active component, it inserts the packet into the input queue or packet channel of the corresponding component, and continues the classification process. This permits efficient classification of packets independently and thus asynchronously, to any processing by active components.

Figure 5 illustrates the architecture of the LARA++ classifier. Incoming packets, intercepted by the packet interceptor, are asynchronously queued on a circular buffer known as the external queue. The classification thread sequentially takes packets from the external queue and performs an initial classification on them.

Within each node of the classification graph, filters are processed starting with flow filters, followed by general filters, graph filters and finally the default filter. Graph filters are processed last to ensure that all active components can process the packet before the packet is passed to the next node in the classification graph.

When a packet has been matched by a component filter, it is immediately queued in the packet channel of the corresponding active component, so that the classification engine can move on to classify more packets; this continues until all packets are classified or the scheduling quantum is exhausted. Once classified packets have been processed by an active component, they are returned to a second queue known as the internal queue. The purpose of this queue is to hold packets awaiting further classification and allow the classifier to favour packets that require re-classification over those on the external queue, this minimises latencies and resource utilisation.

![Fig. 5. The LARA++ Classifier Architecture](image)

When the classifier resumes the classification of a packet held in the internal queue, it continues with the next filter in the same classification node where it previously stopped. If a match occurs with a graph filter, or with the default filter (matched if no other graph filter is applicable), classification continues at the next classification node, which is defined by the output node attribute of the corresponding graph filter.

Several components will typically process a packet as it progresses through the active router. However, it is not possible to identify all the components to which a packet will be dispatched in advance as any component could change the content of the packet. Therefore, re-classification of packets between the processing of active components is crucial. This is an important feature that distinguishes LARA++ from active router implementations such as CANEs [26], Router Plugins [10] and Scout [8].

Packets being processed by a LARA++ node are given a packet context, which is used to store state information,
such as its progress through the classifier. On arrival at the classifier, the packet’s flow key, which facilitates flow filter lookups, is generated and stored in the packet context. When checking flow filters, the key is used to perform a lookup in the hash table for the node.

### 4.2 Filter Processing

The creation of a service composite for each packet is defined by the installed packet filters. Section 3.2 introduced the notion of filter patterns. Each filter type, whether a general, flow or graph filter, contains such a pattern as one of its attributes. While expressing packet filters in this way is convenient and extremely flexible, it comes with an inherent overhead. The position of any fields (such as, `TCP_HEADER`) specified within a filter pattern can change from packet to packet due to included headers and options. This means that the absolute offset must be recalculated for each packet. The impact of this operation can be somewhat reduced if the classifier maintains a list of packet characteristics, protocol offsets, etc. as they are identified during the packet’s journey through the active router. These *features* can then be (re-)used in subsequent calculations, rather than having to re-calculate their locations each time. For example, the classifier could store the offset of the IPv6 header in the packet when this header is first encountered allowing it to be re-used for offset calculations in subsequent filters. Thus, for each classification node a packet traverses on its route through the classification graph, the classifier records a corresponding feature offset in the packet context\(^5\). As a result, it is for this reason that most classification graph nodes correspond to protocol headers.

As an optimisation, graph filters are given an additional property known as the *focus translation*. A *focus* is essentially a reference point within a packet such as the start of a new protocol header. At the beginning of the classification process, the packet has an empty stack of *foci*. For each graph filter that is matched, a new focus is pushed onto the stack. The new focus increases/decreases the previous offset by the focus translation of the matched graph filter. For example, the focus translation of a graph filter connecting the IPv4 graph node and the TCP node would be the size, in bytes, of the IP header. A focus that pointed to the start of the IP header would point to the start of the TCP header after the processing of the graph filter.

In order to find a feature of the packet for use in offset calculation, the problem is reduced to one of searching for the desired feature in the stack of foci. The focus stack model was chosen because most references to packet features are close to the focus of the current classification node. Since the most recent foci are placed at the top of the stack, searches for them usually find a match within a few attempts. For example, a filter that checks the protocol field in the IP header will normally be placed in the IP header classification node. The filter can therefore use the top focus on the stack (i.e. the `IP_HEADER` focus, which points to the start of the IP header) to quickly locate the protocol field (i.e. “focus{IP_HEADER}+IP_PROTOCOL”).

\(^5\) An internal data structure used by the classifier.
Another heavyweight task in filter processing is the computation involved in calculating the packet offset of header fields. The fact that packet features are not always of constant length (a typical example being the way IP options and padding cause the length of an IP header to vary) creates a need for flexible expressions, such as the one above, in order to specify the focus translation for graph and default filters. Since the majority of network traffic can be categorised into just a few payload types, many filter patterns will need to be checked against every packet passing through the node. Given the frequency at which offsets are evaluated, and the fact that packet filters do not change after filter installation, it is best to evaluate the semantics of the expression at the installation time of the filter. As a result, we developed a just-in-time compiler that translates the machine-independent filter expressions into machine code at the time of filter installation. Consequently, execution of compiled expressions is very lightweight (only a few CPU cycles). This allows focus translations and packet offsets to be calculated efficiently and flexibly on a per-packet basis.

5 Performance Measurements

In order to evaluate the performance of the filter-based composition mechanism, we took throughput measurements on our prototype LARA++ router for three different scenarios. These scenarios are intended to provide proof of concept for the LARA++ dynamic composition model. Each of the scenarios operated over the same populated classification graph, albeit with different processing characteristics for each one.

The first experiment involved a type of packet chosen so the traffic passes through 5 classification nodes in the classification graph. The packets were checked against 10 general filters and 500 flow filters. The second experiment used packet content causing traffic to pass through 10 classification nodes in the graph. The number of general filters was thus doubled in comparison to the first test. However, the number of flow filters on the classification path was kept constant at 500. By comparing the throughput of the first and second tests, we expected to find the processing load to be proportional to the number of classification nodes through which the packet travelled.

The third scenario involved the same packet format and number of general filters on the packet path as in the second test, but the number of flow filters on the classification path was doubled. The objective of this last experiment was to confirm that adding extra flow filters does not proportionally decrease performance, all other things remaining equal.

Figure 6 presents the results of these three experiments calculated over 5 million packets. As expected, the throughput roughly halved between experiment one and two because the number of classification nodes on the packet path doubled. As expected, the results show that graph filters and general filters are expensive, however, their numbers are independent of the number of users of the active node. Furthermore, since their number is ex-
pected to reduce as we move in the core (due to the reduced complexity of the classification graph), there are no scalability implications. When increasing the number of users of the active network, it is the number of flow filters that will increase proportionally. Between experiment two and three, the number of flow filters doubled, yet performance was virtually unaffected. With an average drop in throughput of 0.43% between these experiments, we have shown that the use of flow filters allows the classification model to scale well even when the number of users rises significantly.

Further experiments were performed with the aim of measuring packet latency. Figure 7 illustrates a breakdown of the time taken to perform different stages of classification. Six states of processing have been selected to represent the complete passage of a packet through the classifier, and the figure shows how these states account for the total latency incurred by a packet. The average latency imposed by the classifier on an individual packet is 9.4µs, which is a tiny fraction of the latency seen by most packets travelling though a traditional router. For packets administratively excluded from active processing, cut-through paths in the classification graph can further reduce latency. Of the total latency introduced, more than half is accounted to the processing in the classification graph. The complexity of the path through the classification graph undoubtedly direct impacts the latency. However, the main determinant of packet latency is the “complexity” of the packet itself. The majority of packets that have only a MAC header, a network header, a transport header and a payload will simply take a “shortcut” through the classification graph, avoiding the extra classification nodes required to process optional protocol headers.

![Classifier Throughput for Pre-defined Packet Paths.](image)

**Fig. 6.** Classifier Throughput for Pre-defined Packet Paths. Tests performed on an Athlon XP1800 / 512Mb RAM running Windows 2000, using simulated packets, to reflect the Classifier’s capabilities untainted by other active routing activities.

The average latency incurred while processing flow filters is comparatively small (~1µs) in the context of the total packet latency. The measurements described above show that doubling the number of flow filters in the classification path reduces the throughput by less than 0.5%, and the impact of this should be barely noticeable. The classification latency of packets is therefore largely unaffected by a change in the number of flow filters installed on a LARA++ active router.
6 Evaluation of Composition Framework in Real-life Service Network

In order to demonstrate the ease and flexibility with which LARA++ active components can be deployed and composed to provide real-life network services, we have implemented a number of ‘typical’ components, some of which will be further discussed in the following paragraphs. The service network we used for the different evaluation scenarios is depicted in Figure 8.

Fig. 8. Evaluation Environment – A Real-world Service Network

6.1 Handoff Optimisation for Mobile-IPv6

In order to allow the mobile users of our wireless access network to freely move among different WLAN networks and also hand-over to other wireless service networks such as GPRS/UMTS, we have deployed an implementation of the recently approved Mobile-IPv6 protocol [17]. MIPv6 allows network hosts (mobile nodes) to move between subnets whilst remaining addressable via their permanent address (home address).
MIPv6 operation is based on the existence of a home agent at the mobile node’s home network responsible for intercepting all traffic destined to the mobile node and relaying it (using tunnels) to the mobile node’s current network address (care-of address) on a visited network. As a routing optimisation, the communication peer (correspondent node) can also be informed of the mobile node’s care-of address (using a message called a binding update), which allows it to send packets directly to the mobile’s current network location rather than through the home agent. While this optimization is necessary to prevent scalability issues and improve efficiency, it leads to an awkward situation during the brief period while the mobile node is migrating to a new subnet. Upon configuring its new address, the mobile node will send a binding update to the correspondent node informing it of a new care-of address. Since the correspondent node cannot address packets to the new care-of address until the binding update has arrived, packets will still be arriving on the previously visited subnet using the old care-of address for at least one RTT between the mobile node and the correspondent. These packets would be lost unless the mobile node was multi-homed, allowing it to maintain an interface on the previously visited subnet, which is typically not the case.

Using LARA++ active routers on our wireless access network, we circumvent this deficiency by augmenting the handoff process by means of an active component installed at a router close to the handoff. The active component learns about the mobile node’s change of address from the binding updates on adjacent links and “re-routes” traffic destined to the mobile node’s old care-of address to the new care-of address while the binding update traverses the wider network. This simple optimisation greatly reduces the packet loss during handoff procedure. The detailed operation of this mechanism, has been previously published by the authors [18].

This Mobile-IPv6 handoff optimisation can be deployed in two ways on an active router: pre-emptively or on-demand. In the pre-emptive scenario, a network administrator or service provider might roll out a component onto strategically chosen active routers as a service to network users. The component would listen for binding updates from all machines on the subnets, and perform the routing optimisation when it detects a mobile node handing off. In the on-demand scenario, the mobile node would dynamically instantiate a component on a convenient active router to redirect packets after its movement, or before it moved if it had advance warning (from a layer 2 driver, for example). LARA++ can support both approaches equally well.

The implementation of the Mobile-IPv6 handoff optimisation as a LARA++ active component was a simple task. On intercepting a home address option, the active component keeps a record of the home address binding for that correspondent. If, at some point, it sees a new care-of address being used for the correspond address/home address pair (i.e. a handoff has taken place), the handoff optimisation component dynamically inserts a short-lived filter for the reverse route that intercepts all packet from the correspondent to the mobile node’s old care-of address. For the lifetime of the filter, intercepted packets are re-routed to the correct (new) care-of address thus avoiding packet loss.

---

7 This route optimisation is only performed during the transition period, between the mobile node handing-off and the correspondent node adopting the new address for traffic.
loss caused by the round-trip latency. Exemplary of the implementation simplicity is the fact that the basic processing loop did not exceed 10 lines of code.

Figure 9 presents measurements taken on one of our LARA++ test routers\(^8\) while running the handoff optimisation service. The graph shows the number of packets that our component re-directed and hence saved from being lost for a range of different media streams (GSM and PCM audio, DivX, and MPEG1). The results indicate that the number of packets lost due to the propagation delay of the binding update increases linearly with the round trip delay between the active router at the edge of the mobile node’s network and its correspondent node. Providing this service in a mobile access network, where the majority of handoffs are localised, can therefore greatly improve handoff performance and thus multimedia streaming quality for mobile devices communicating with distant (in terms of delay) nodes.

Fig. 9. Measurements of packet loss during Mobile-IPv6 handoff in various network latency and traffic load conditions.

Other ways of addressing the packet loss caused by the end-to-end latency on mobile devices and their correspondents during network handoffs commonly rely upon middle boxes that must be pre-configured to act in a specific way. For example, Hierarchical Mobile-IPv6 [19] uses a “MAP” node to localise handoffs and thus reduce the end-to-end delay of the binding updates, and Fast Handovers for Mobile IPv6 [20] uses extra infrastructure in the network to initiate handovers and establish routing tunnels to the new location of the mobile node. Indeed, had our fast Mobile IPv6 handoff component not been implemented as an active service, it too would have required a dedicated, pre-configured middle box to provide the service. A key strength of LARA++ is that it can be used as a basis for the implementation of any of these three mobility solutions and even allows them all to co-exist on the same LARA++ node. Moreover, it enables the dynamic roll-out of all of these services as they are needed.

---

\(^8\) Windows XP based PC with a 1.7 GHz Pentium 4 processor, 384 MB RAM and three Fast Ethernet Interface Cards.
6.2 Network Address Translation on the Gateway Router

On our internal wireless access network at Lancaster University, we allow mobile users to access hosts on the
global Internet through network address translation (NAT) [11]. As a result, the private IP addresses used for the
wireless access network are translated to globally routable addresses for outgoing packets, and vice versa for in-
coming packets, on the gateway router that connects the two networks. This NAT service for our access network
has been implemented as a LARA++ active component in order to further demonstrate the flexibility and range of
applications LARA++ can support.

The NAT component installs packet filters to intercept outbound TCP, UDP and ICMP traffic, which originate
“inside” the access network. Upon intercepting packets using these filters, the active component looks for an ex-
isting association between the private address and port, and the correspondent address and port. If such an asso-
ciation does not exist, the component creates a new association and assigns a valid global address and port pair
from a pool of global IP addresses and port numbers. For new outgoing “connections”, the active component then
installs another filter for the reverse path from the correspondent address and port to the global address and port
for inbound traffic. These filters, although numerous, exploit LARA++’s scalable flow filters (see section 3.2),
allowing many thousands of such filters to be installed without adversely affecting the performance (as discussed
in section 5). The component is also able to insert a filter so that it can intercept and respond to ARP requests for
any addresses from the address pool that the router uses for NAT.

From then on, the NAT component simply intercepts inbound and outbound traffic. Packets intercepted on the
inside interface have their source address and port replaced with the global address for the connection, and packets
intercepted on the outside interface have their destination address and port swapped with the private address for
the connection. After the IP, TCP and UDP checksums have been recalculated, the packets are then handed to
the standard TCP/IP stack, which then routes the packets as normal.\footnote{This avoids re-implementing routing and forwarding functionality, although it could equally well be implemented as a set of
LARA++ components.}

Of the total 18.5 ms average RTT latency, 18.1 ms is caused by the latency of the network, existing protocol stacks
and the test application at each end of the connection. The LARA++ active NodeOS adds another 0.2 ms to the
RTT, and finally the NAT component itself is responsible for the remaining 0.2 ms.

It is interesting to compare the jitter for the different experiments. Packets processed by LARA++ components are
more likely to experience additional queuing delays. Consequently, we would expect streams that are processed by
LARA++ to have more variance in packet latency. However, the results indicate that LARA++ processing in-
creases the packet jitter only marginally. For example, while 92.4% of packets have less than 250 µs jitter under
normal conditions, with LARA++ and NAT processing this value drops only to 89.7%.

In order to measure the maximum throughput on our LARA++ test router, we ran the NAT component alone with
increasing traffic. With 1KB UDP packets the router was able to sustain approximately 82 Mbps. This compares favourably with a maximum throughput of just over 83 Mbps on the same machine without LARA++.

![Graph](image.png)

**Fig. 10.** Packet RTT (left) and jitter (right) of a UDP/IP stream across a LARA++ router with the NAT component instantiated and three other control cases.

6.3 Secure Deployment of Firewall Support

Considering the two previous examples of active components, it is clear that while users might legitimately want to install a mobile handoff optimization component to speed-up handovers for their own flows, there are some applications that could degrade the network if end users were able to install them. For example, an application such as NAT installed by a user could frustrate the network administrator’s attempts to limit access to other networks using conventional means. More specifically, NAT could be used to bypass any security built on the assumption that the network is non-routable, or else some simple variant of NAT could be used to bypass port blocking, which is typically used by most firewalls.

This problem is representative of a range of problems that arouse while we deployed our active routers in a real service networks. An important lesson learnt is the need to consider not only threats to the router itself, as an active node, but also those deriving from the use of active applications when configuring the router. Like in conventional routers, configuration of active networks must either systematically prevent undesirable behaviour by active applications or be based upon a well-tested, “safe” configuration, which is modified only with care.

Users could be prevented from using an active router to circumvent security, as in the case of the flow-specific NAT and port-blocking avoidance examples, by means of a firewall designed to counter such threats. With this in mind, we implemented a firewall active component designed to be installed and configured in a tamper-proof fashion to other active components. Much of the difference between this and a conventional firewall is not in the way the component is implemented, but rather the way in which it is installed and configured.

It is imperative that unprivileged users have no access to packets, not even of their “own” streams, until after the firewall has eliminated packets it regards as undesirable. The way we achieve this on LARA++ is by augmenting
the standard CGT-defined graph with a new classification node that accepts packets immediately prior to the node that processes network layer headers and installing the firewall component there. Furthermore, access to the classification graph can also be restricted through node-local policies such that ordinary users can gain access only to certain types of packets and not others.

Another important lesson learned from this exercise is that the network is no longer the sole untrusted domain. Filtering out undesirable ingress traffic no longer guarantees that all forwarded packets are non-malicious. Therefore, a “well-behaved” active router configuration that prevents its components misbehaving on its networks must also enforce checking of egress traffic.

In summary, the evaluation of the LARA++ service composition framework based on the above outlined example scenarios shows that the service composition approach is flexible and generic enough to cope with the unique, and even potentially overlapping, interests of the different services. Moreover it provides a security model that is capable of resolving undesired interactions between user and active component privileges and administrative policies.

7 Scalability of the LARA++ framework

The quantitative evaluation of our filter-based composition model showed that the processing latency imposed by the classifier on an individual packet is less than 100 µsec, which is a tiny fraction of the latency introduced by a normal edge router. Additionally, the use of the classifier (tree-like structure) allows the sorting and selective consideration of a packet by a limited number of filters only along the traversed path. The measurement results also demonstrate that the introduction of flow filters allows the composition model to scale exceptionally well in terms of performance as the number of users increases. Practically, the number of distinct flows for which LARA++ is asked to do active processing does not impact the performance of the active node (of course this does not account for the processing overhead by the service components themselves). These observations suggest that the overhead of enabling the LARA++ framework is minimal on one hand, while on the other hand it does not increase proportionally to the number of filters installed or users serviced. In other words the LARA++ framework is can scale very well in its deployment environment, which primarily is the edge network.

On the other hand the number of nodes along the path of a packet through the classification graph is a factor that has monotonically increasing performance penalty. As a result the length (or depth, if seen as a tree) of the classification graph has a proportional impact on the performance of LARA++. However, the size of the classification graph will be significantly reduced as one moves towards the core of the network due to the fact that closer to the core, where there is high level of flow aggregation, it is very unlikely that services of user-level granularity will be needed. This counterbalances the negative effect from the size of the classification graph in a way that maintains the scalability of the LARA++ composition framework even when moving towards the network core.

Our experiences with the current PC-based implementation LARA++ have taught us a great deal about the trade-
off between performance and flexibility. The line speeds we have been able to sustain, whilst good, are far short of those appropriate for core networks, yet very acceptable within typical edge networks. However, most user-level active applications are best suited to deployment in edge networks, and in edge networks the flexibility to deploy such functionality is more important than optimal performance. Finally, from the above analysis we are led to believe that a network processor (NP) implementation of the LARA++ framework could also provide a competitive (to commodity routers) high-end solution well suited for closer to the core environments.

8 Related Work

The main goal behind LARA++ was to build a highly generic and flexible active router architecture. While we have successfully demonstrated this with a range of very different active services (see section 6), we have not been able to find many related works that focus on the macro-level service composition aspect on active routers and deal with the evaluations of active routers in a real world service environment. As a consequence, we focus more on a comparison of LARA++ with other active network approaches.

A common objective of most active network approaches is to expedite network evolution through solutions that enable extensibility of network functionality by way of dynamically loaded code. Most active network approaches, such as ANTS [21], NetScript [22], PLANet [23], and SmartPackets [24], accomplish this through software plug-ins or a similar form of active code integration. The extensibility limitations of such plug-in based approaches have been revealed in a study by Hicks and Nettles [2]. In particular, the study shows that most of today’s plug-in approaches use a fixed “underlying” data structure or program that defines the “glue” for the service composition on the active node. The fact that these composition structures are specified at compile time of the kernel limits extensibility. LARA++, by comparison, allows the dynamic extension of the classification graph (i.e. allows the creation of new nodes at run-time) and also overcomes the limitation that only one plug-in can be incorporated per slot (i.e. many components can install packet filters to a classification node).

Starting with the CANEs execution environment [25], we further compare closely-related approaches to LARA++. CANEs, which has been designed specifically for the Bowman active node OS, provides a composition framework for active services based on the selection and customisation of a generic “underlying program”. This program can be tailored for a type of packets or set of streams by injecting customised code into well-defined slots in the underlying program. The packet filter mechanism, selecting the underlying program, can be configured to match arbitrary patterns in the packet. This flexible classification approach allows Bowman to dynamically deploy new protocols at run-time like LARA++ does. However, in contrast to LARA++, Bowman/CANEs encompasses a number of restrictions. First, Bowman/CANEs restricts classification to the selection of an underlying program; i.e., once an appropriate underlying program has been identified, the service composite is fixed and is solely dependent on the plug-ins. Second, although Bowman/CANEs appears to allow multiple underlying programs to be
selected, the literature implies that only a copy of the packet can be sent to each logical input channel which prevents implicit active program co-operation. Bowman/CANEs further restricts service composition. The static nature of the underlying program for any given execution environment is by definition inflexible. One needs to make assumptions about the customisable aspects of the program at instantiation time of the execution environment.

The Router Plugins architecture [10] also uses a plug-in based composition model, whereby an underlying program defines the “glue” for the service composites. Since these composition structures are defined at compile time of the kernel, extensibility is limited to predefined gates. LARA++, by comparison, allows the dynamic extension of the classification graph (i.e. addition of classification nodes at run-time) and overcomes the limitations of only one plug-in per gate (i.e. many active components can be inserted per node).

Further related works are the modular router architecture Click [9] and the configurable operating system Scout [8]. Both use a graph-based composition approach like LARA++ to support extensibility of the communication subsystem through so-called modules. However, since configurability in both cases is limited to the compile-time of the system, dynamic introduction of new services is not possible.

Finally, VERA [26], which in contrast to the latter two platforms strives to be highly extensible, appears to be limited in another aspect. It allows only one forwarder to be chosen for each packet that is classified. Even if packets could be (re-)classified several times, VERA still lacks a method for the chosen forwarders to compose an overall service in a cooperative and deterministic way.

This analysis demonstrates the need for the service composite of plug-in or component-based solutions to be extensible at run-time. Assuming an underlying graph structure or program that cannot be dynamically changed is not necessarily suitable for the lifetime of the system, and thus, limits extensibility unnecessarily.

9 Conclusions

In this article we have presented a novel framework for managing the creation of service composites in an active router. The LARA++ composition framework supports dynamic integration of router extensions (active components) at runtime. The classification-based service composition model enables flexible integration of extended router functionality at any point in the packet-processing path. The classification graph, representing the packet-processing path on the router, provides the necessary management structure for the integration of the software extensions enabling the selective consideration of packets for active processing. The use of packet filters as a means of binding the software components allows the composition mechanism to dynamically incorporate new functionality at run-time and at the same time at any traffic flow aggregation level. The selection of traffic for active processing is based on packet the content; any bit pattern can be used to trigger the processing by an active component, as can characteristics such as packet length.

The quantitative evaluation of our filter-based composition model showed that the processing latency imposed by
the classifier on an individual packet is less than 100 µsec, which is a tiny fraction of the latency introduced by a normal edge router. The results also showed that the introduction of flow filters allows the composition model to scale exceptionally well in terms of performance as the number of users increases.

We demonstrated that the LARA++ composition model can managing both competition and co-operation between users of an active router by allowing unrelated users to partake in the programming process of the active node in a structured fashion.

Our experience of developing active services has led us to realise that for an active router platform to become successful it is imperative that it offers a developer-friendly programming interface. This broad requirement encompasses the ability to write active components in common programming languages. LARA++ does not impose the use of a particular user-space language or development environment, whilst enables run-time debugging of active components when deployed on an active router. LARA++’s filter mark-up language also adheres to this principal with a clean and extensible syntax, making the design of new filters trivial.

Probably the most important result is that we have successfully deployed LARA++ in real-world situations commonly found in today’s networks. Our prototype implementation of LARA++ also allowed us to exploit existing operating system-provided network functionality and interoperate with it, without being constrained by it. LARA++ is generic enough to enable the overriding of the hosting OS forwarding subsystem as it provides the active service developer with the primitives and the means to re-define the network level functionality of the local node. This is essential for a smooth transition from conventional networks to active networks.

Concluding, the experience we gained from deploying our LARA++ active node platform in a service network, along with a number of real network services, have taught us that end-users will benefit more from active technologies when they are deployed towards the edge of the network in a way that transparently introduces their new capabilities. As a result, the LARA++ active router architecture, is perfectly well suited to meet the needs of today’s edge networks where most of the network services are needed and service customizability is a real concern. Nevertheless, we also believe that active networking can be useful when available throughout the whole network infrastructure as exemplified in [27]. Consequently, a future challenge for LARA++ and the herein presented service composition framework is to deploy it on high-end network processor (NP) platforms.

References


