Flexible, Dynamic, and Scalable Service Composition for Active Routers

Stefan Schmid, Tim Chart, Manolis Sifalakis, Andrew C. Scott

Distributed Multimedia Research Group
Computing Department
Lancaster University, U.K.
{sschmid, chart, mjs, acs}@comp.lancs.ac.uk

Abstract. This paper describes a novel model for the provision of service composites for active routers. The service composition framework enables flexible programmability of the router's data path through dynamically loadable software components, called 'active components'. The composition model promotes transparent and dynamic creation of network-side services and allows independent users to partake in this process. A prototype implementation has revealed that the composition model using packet filters and a classification graph structure as a means to integrate active components into the forwarding path enables the dynamic alteration of the elements of a composite at run-time and permits scalability in the generation of such composites. Furthermore, it allows the flexible provision of a unique service profile for each packet passing through an active router. We show that the overhead of this composition model does not significantly affect the performance of the router.

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 those platforms address only a subset of the issues that determine their utility outside of a laboratory environment, resulting in many being inflexible, poorly-performing, unscalable or insecure. Few active network solutions have considered the magnitude of the service composition model for real-life network environments and hence provide only a limited flexibility for the composition of network services. Yet, in order for active networking to be considered a suitable technology for wide deployment over inter-networks, these issues must be addressed.

A recent study by Hicks and Nettles [10] has revealed that even most extensible active router platforms lack sufficient flexibility in order to allow for true evolution. Such modular or plug-in based architectures typically limit the scope of future changes through pre-defined interfaces. Instead, true extensibility should not be limited to a fixed set of modules or plug-ins, but should rather allow modification and replacement of all components contributing to a service composite.

This paper presents the service composition framework – a key component of the LARA++ [6] active router architecture – which attempts to provide a secure, safe, flexibly extensible platform for the deployment of powerful active components. The main objective of the LARA++ architecture is flexible extensibility in order to facilitate network functionalities and services whose need may evolve in the future by making the entire forwarding path of the router programmable. The *packet classifier* is the LARA++ component responsible for the effective and flexible deployment of active components and the decentralised provision of an acceptable service composite.

The remainder of this paper is organised as follows: The second chapter gives a brief overview of the LARA++ architecture. Chapter 3 continues with a detailed description of the service composition model. Chapters 4 and 5 describe the prototype implementation of the LARA++ classifier and the measurements made from that prototype. Chapter 6 compares the composition model with other architectures. Finally, chapter 7 concludes on our findings.

1.1 Motivation

Before we describe the LARA++ architecture, we outline some scenarios that adequately encompass many of the problems of service composition. Scenarios such as those described in this section precipitated the desire to provide a flexible and scalable composition model for active routers.

Suppose an active node has two components installed to process packets arriving on a given interface. One of the components might be a lightweight firewall component installed by the router administrator, which is able to efficiently filter incoming packets to prevent processing of certain types of traffic. A second component installed by a network user might, for example, offer intelligent congestion control services for a custom protocol carried over UDP. The intelligent congestion control is likely to require much more processor time than the firewall, so it would be advantageous for the firewall component to be processed first, thus reducing the amount of traffic sent to the congestion control component. In order to ensure this optimisation, the active router must necessarily have a framework that allows an ordering of components to be maintained. Moreover, where two users dispute the ordering of components on the same stream, user privileges must be taken into account.

Another example might involve a network user who would like to co-operate with active components already deployed on the active router. For example, an existing active program on the active router might provide IPv6 transitioning support in the form of network address translation [7]. IPv6 packets entering the router from one interface might be converted to IPv4 packets before being forwarded out of another, and vice versa for the reverse path. This would enable IPv4-only hosts and IPv6-only hosts to converse. However, this would cause any applications that embed IP addresses in the payload of the packet to ‘break’. A network user wishing to provide support for such an application over a translating router would need to provide a component that co-operates with the network-layer translation active component. The co-operation would require both components to be processed, but in a strict order. The challenge would be to find a way of asserting this co-operation over the same stream of packets without causing undesired interference between the components.

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 restricting the flexibility and programmability of the active node.

2 Background

LARA++ is a software architecture that evolved from the predominantly hardware oriented Lancaster Active Router Architecture (LARA) [1]. The architecture is generic in the sense that it can be implemented on top of any router platform with a software forwarding engine. Platform independence is assured by virtue of the fact that prototype implementations have been developed for Windows XP and Linux.

The LARA++ architecture is designed to extend existing routers with active network functionality. Low-level functionality of the active router architecture is directly integrated with the router OS in order to maintain good performance for active processing. A high-level layer accommodates processing environments that enable safe processing of dynamically loadable[1] software components. Well-known interfaces are exposed at this layer in order to unify programmability across different platforms.

The processing environments provide the policing and code isolation necessary for a fair and safe platform. However, since this paper focuses on the service composition model of the LARA++ architecture, we refer the reader to previous publications [1].

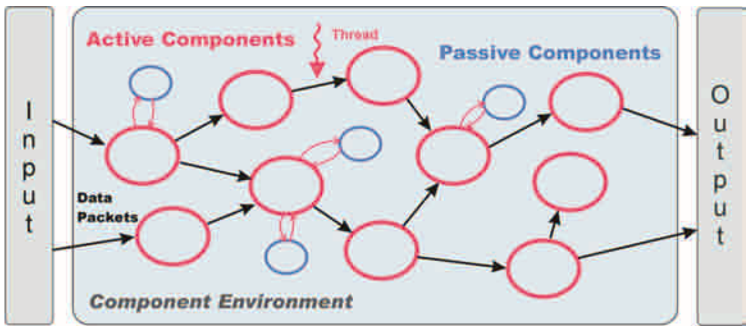

Fig. 1. A Conceptual View of the Active Component Space.

Figure 1 provides a conceptual overview of the approach taken. The vision of the LARA++ platform is to provide a framework upon which complete router functionality can be provided in component form (for example, a routing component, a filtering component, etc.), which are then composed into actual 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.

[1] Based on in-band or out-of-band code loading techniques.

Service composition is achieved through packet classification. Packet filters define the processing “route” through the component environment. This allows such “routes” to be appropriately tailored to the packet type or content.

3 LARA++ Composition Model

The LARA++ composition model plays a central role in the overall architecture, as it provides the foundation for the flexibly extensible and dynamic component-based programming model. Service composition is carried out in two manners:

- *Macro* composition is achieved at the service level via the 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 LARA++ NodeOS).
- *Micro* composition is achieved at the component level using an explicit, lightweight composition model, which enables the construction of active components from passive components. The fact that active components can be largely composed from functionality provided by passive components facilitates active component design and co-operation between active components (section 1.1).

Macro composition within LARA++ is largely packet driven. Dependent on the packet content, a different overall service may be composed for the processing of that packet. The *packet classifier* plays a key role in the service composition process. It determines based upon the set of *packet filters* currently installed whether or not a packet passing through the active router requires active processing, which active component(s) are involved, and in which order they should process the packet. Thus, the active components implicitly and collectively define a service composite, resolving competition (see section 1.1) in the process. 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 the “route” through the active component space for packets passing through a node. The classifier filters incoming (and outgoing) network packets based on the component filters installed in the classification graph. Figure 2 presents an example classification graph.

The classification mechanism traverses the classification graph starting at a root node (i.e. **`/netin`** in figure 2). At each node in the graph, the classifier tries to match the packet filters installed there. Packets matching a filter are passed to the corresponding active component. After completion of the active processing, the classifier continues classification at the same point (or an optionally specified point defined by the packet filter[2]) in the classification graph. When the classifier has applied all packet filters that have been installed by the active components at a node, it follows the classification graph based on the "default" or graph filters (for example, **`/netin/ipv4`** or **`/netin/ipv6`**) and continues the multi-stage classification process there. Finally, the packet is forwarded to the next hop router when the classifier runs out of graph filters.

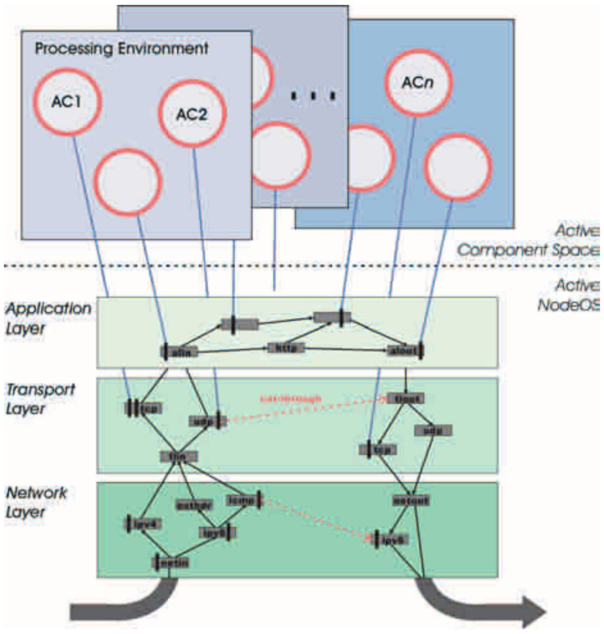

Fig. 2. The classification graph

3.2 Packet Filters

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 active components to define the network packets of their interest. These filters are typically registered with the packet classifier at component instantiation or at run-time if necessary through the LARA++ system API. The classifier uses these filters to determine the active components to which network traffic is sent for active processing. For example, a customisable firewall component might register active component filters for each protocol the user has asked it to filter. Graph filters, in contrast, are used by the classifier itself in order to define the structure of the classification graph.

[2] Availability of this option depends on the user privileges and filter type.

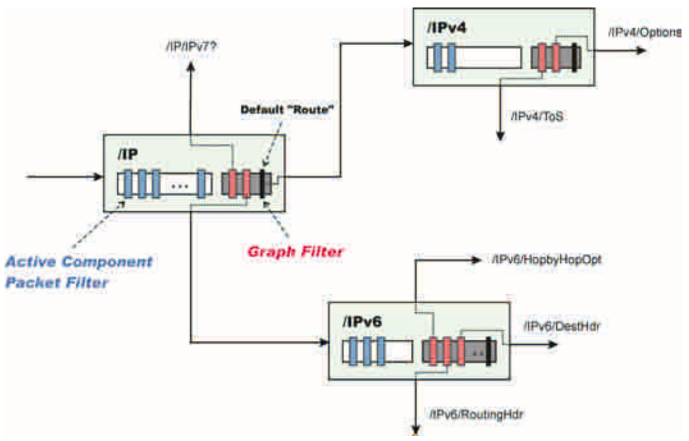

Fig. 3. The classifier manages packet filters within a filter graph structure, called classification graph. Active component filters are used to dispatch network data to ACs for processing, whereas graph filters are used to define the structure of the graph.

Packet filters installed by active components can be further divided into *general filters* and *flow filters*. Since a single active component could install multiple filters, and given there may be many components running on an active router, it would be very costly to check all of those filters. Flow filters (a specialization of general active component 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 based upon the flow characteristics of the packet being processed using a hash table. Consequently, no processing is necessary to reject most unmatched flow filters. Due to the hashing technique used to lookup flow filters, a LARA++ router can handle a potentially large number of these filters (which is fully sufficient for typical edge routers – the target platform for LARA++).

All types of filter define patterns against which the active node attempts to match characteristics of packets. Such characteristics are commonly fields in packet headers, and are described by a four-tuple of `{packet offset, bit pattern, bit mask, pattern length}`. The specification of packet filters is facilitated through well-known reference points and pre-defined tags. For example, a filter for HTTP traffic might make use of the `TCP_HEADER` reference point like so: `{TCP_HEADER + TCP_PORT, 0x0050`[4]`, 0xffff, 2}`. Packet filters are fixed in a single classification node, known as the *filter input node*, which must be appropriate to the filter. For example, the HTTP filter defined above could be placed in the classification node dedicated to TCP traffic because the filter requires TCP as a prerequisite. Graph filters and some specially privileged active component filters also define a *filter output node*, which provides an alternative route (*cut-through path*)

[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.

through the classification graph. For any packets matching those filters, classification commences at the filter output node.

Active component filters require additional properties: (i) an *operation property* to express the packet access permissions required by the component (i.e. read-only, read-write or write-only) and (ii) a *principal* (security credential) to indicate the network user, on whose behalf the filter is installed, and/or the code producer. The operation and principal properties permit the classifier to authorise the insertion of a filter based upon the node-local security policy and the privileges associated with the principal.

3.3 Classification Graph Table

The classification graph table (CGT) provides the means to describe the structure of the classification graph. Its main purpose is to make the graph structure globally available across a LARA++ active network. In order to support flexible extension of the classification graph (e.g. to incorporate new protocols or extend current protocols), an "elastic" means to describe the graph structure is required. For this purpose, a simple notion for defining the nodes (for example, **`ipv4`**, **`tcp`**, and **`udp`**) and the branches of the graph (for example, **`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. For example, network protocol options must be processed prior to transport protocol headers. The fine-grained structure accounts for the layer-specific protocols. For example, extension headers in IPv6 must be processed in a pre-defined order.

The active node security policy specifies the types of filters that may be installed inside the classification nodes. For example, it would make no sense to allow general-purpose filters to be processed before filters pertaining to a firewall component, or otherwise the security measure could possibly be circumvented. Classification nodes also define access permissions for fine-grained control of packet access.

Since the CGT is expected to change occasionally (for example, a new node in the global classification graph might be introduced when a new protocol becomes established), an automated mechanism to update the CGT across the active network is employed. At a first glance, it may seem that the overhead of updating the CGT every time a new protocol is introduced is heavyweight and makes the system inflexible. However, it should be noted that a CGT update is only required if a new protocol or protocol extension is "standardised" (i.e., globally announced such that it can be extended by others). The CGT does not require a global update in order to deploy and test the new protocol or extension locally.

3.4 Composite Characteristics

Service composition within LARA++ is a *co-operative* process; it allows independent network users to install active components that match the same data streams or subsets of streams (controlled by the local security policies). The classification graph provides the means for independent users to integrate new active functionality or services in a "meaningful" way without having to know about other users' active components. The decoupling of component bindings among active components through the classification graph hides component changes from other components.

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

4 Implementation

This chapter outlines the implementation of the LARA++ composition model, and some of the features of the implementation that affect overall performance.

4.1 Design Overview

The classification component of the LARA++ architecture is a key subsystem. Interactions between the classifier and the other components affect the overall performance of the implementation.

The classification component is responsible for the dispatch of classified packets to active components. When the classifier matches the filter of an active component, it inserts the packet into the packet channel (i.e. input queue) of the corresponding component and continues the classification process. This allows the classifier to efficiently classify packets without waiting for active components to be ready to receive them.

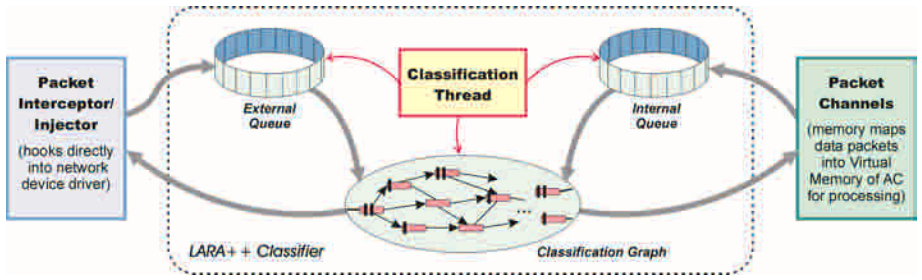
Packet Interceptor/ Injector
(hooks directly into network device driver)
External Queue
Classification Thread
Internal Queue
Packet Channels
(memory maps data packets into Virtual Memory of AC for processing)
LARA+ + Classifier
Classification Graph

Fig. 4. The LARA++ Classifier Architecture

Figure 4 illustrates the architecture of the LARA++ classifier. Incoming packets, intercepted by the packet interceptor component, 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.

After a packet has been classified (i.e. a component has been selected to perform active processing on the packet), the packet is immediately queued in the packet channel of the active component, so that the classification engine can continue to classify more packets until all packets are classified or until its scheduling quantum is over.

Once classified packets have been processed by the corresponding active components, they are returned to a second queue known as the *internal queue*. The purpose of the internal queue is to hold packets prior to the continuation of their classification.

Many components will often be identified to process a packet over the course of its passage through the active router. However, it is not possible to identify all components to which a packet will be sent in advance as the components could change the content of the packet. Therefore, re-classification of packets between the processing of active components is crucial. This important feature distinguishes LARA++ from active router implementations such as CANEs [8], Router Plugins [2] and Scout [11].

Packets requiring reclassification are separated from packets that are awaiting initial classification, so that the classifier can process packets waiting on the internal queue in preference to those on the external queue and thus minimise packet latency.

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, the user flow of the packet is calculated and a flow key is generated to facilitate flow filter lookups. This key is stored in the packet context. When checking flow filters, the key is used to perform a lookup in the hash table for the node. This operation yields a shortlist of candidate flow filters, which are then checked individually.

In each node in the classification graph, filters are processed starting with flow filters, then general filters, graph filters and finally the default graph. Graph filters are processed last in a node to ensure that all active components are processed before progressing to the subsequent node in the classification node. A classification is made if the patterns of any filter match the packet. If the match is made with an active component filter, the packet is sent to the associated component for processing, and resumes the classification process at the next filter in the same classification node on its return. If the match is made with a graph filter, or with the default graph filter (which is matched if no other graph filter could be applied), classification terminates in the filter input node and resumes at the start of the filter output node.

4.2 Filter Processing

The creation of a service composite for each packet is based upon the packet filters. Section 3.2 introduced the notion of filter patterns. Each filter type (general, flow and graph filter) contains such a pattern as one of its attributes. While the expression of packet filters in this way is convenient and extremely flexible, it comes with an inherent overhead. The position of fields that might be identified by the filter pattern (e.g. **`TCP_HEADER)`** can change from packet to packet due to extra headers and options. This means that the absolute offset must be recalculated for each packet. The impact of the operation can be somewhat lessened if the classifier maintains a list of packet characteristics (e.g. protocol headers) that have been identified during the classification of the packet in its journey through the active router. These *features* can then be used in the offset calculation, rather than having to parse the packet to locate these features each time they are required. For example, the classifier could store the offset of the IPv6 header in the packet when the header is encountered so that subsequent filters can use it in offset calculations. Because of this approach, it is not a coincidence that most headers have one or more dedicated classification nodes in the classification graph; this is a property of the composition model.

In order to facilitate this optimisation, graph filters are given an additional property known as the *focus translation*. The packet context contains a stack of foci, and the packet begins classification at the first classification node with a single focus of zero. If a graph filter is matched or default graph is encountered, a new focus is pushed on the stack. The new focus increases/decreases the previous offset by the focus translation of the matched graph filter or default graph. For example, the focus translation of a graph filter branching between an IP header and a TCP header 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 subsequent to 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 (identified by the classification node) on the stack of foci. The focus stack model was chosen because it is likely that most attempts to examine features of packets will be made closest to the focus of the packet in the current classification node. Since the most recent foci are placed at the top of the stack, searches for foci usually find a match within a few attempts. For example, a filter identifying the protocol field in the IP header will normally be placed in the IP header classification node, thus the operation to locate that field (i.e. "`Focus {IP_HEADER}+IP_PROTOCOL`") will find the target focus at the top of the stack.

Another potentially heavyweight task in filter processing is the computation involved in calculating the offset of packet fields. The fact that packet features are not always of constant length (e.g. IP options and padding can 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 and graph filters will need to be checked against every packet passing though the node. Given the frequency of the evaluation of offsets and the fact that packet filters do not change after filter installation, it is best to move the overhead of evaluating the semantics of the expression to the installation time of the filter. We use a just-in-time compiler that translates the safe, machine-independent expressions into native machine code at the time of filter installation. Consequently, execution of the 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

On completion of the prototype classifier, we took measurements of the throughput under three different scenarios. These scenarios were intended to lend proof of concept to the LARA++ model for flexible and dynamic composition. Each of the scenarios operated over the same populated classification graph, albeit with different processing characteristics for each one.

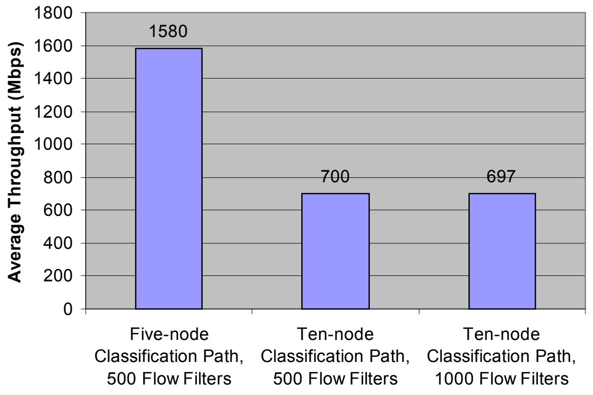

Fig. 5. Classifier Throughput for Pre-defined Packet Paths. These tests were performed on an Athlon XP1800 with 512Mb RAM running Windows 2000, using simulated packets, thus producing a reflection of the Classifier's capabilities untainted by other active routing activities.

Experiment one involved a type of packets chosen so that 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 a packet content that causes the traffic to pass through 10 classification nodes in the classification graph. The number of general filters was doubled in order to impose roughly the same processing load per node (as in test one), whereas the number of flow filters on the 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 the second test, but the number of flow filters on the classification path was doubled. The objective of this experiment was to confirm that adding extra flow filters does not proportionally decrease performance, all other things remaining equal.

Figure 5 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. The results show that the graph filters and general filters do not scale well. Fortunately, their numbers are not related to the number of users of the active router (unprivileged users can only install flow filters), and hence do not have to scale to large quantities. By increasing the number of users of the active network, mainly the number of flow filters will increase proportionally. Between experiment two and three, the number of flow filters doubled, but 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 as the number of users rises.

Further experiments were performed with the aim of measuring the packet latency. Figure 6 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 of a packet. The average latency imposed by the classifier on an individual packet is 9.4µs, which is a tiny fraction of the latency of most packets travelling though a passive router. For packets that do not require active processing, cut-through paths in the classification graph can further reduce that latency.

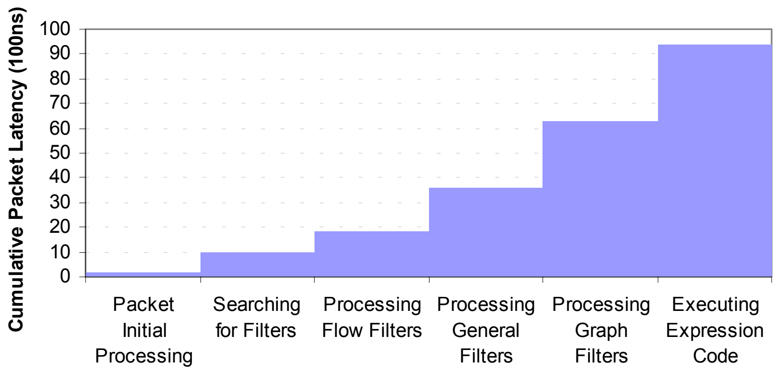

Fig. 6. Breakdown of the stages of packet classification, measured in tenths of microseconds. The classification graph and its population were chosen such that 500 flow, 10 graph and 12 general filters were checked. Of these, 1 flow, 2 general and 4 graph filters were matched.

Of the total latency, more than half is accounted for by the complexity of the classification graph. The complexity of the path through the classification graph undoubtedly has a direct impact upon the latency. Thus, the main determinant of the packet latency is the complexity of the packet itself. The majority of packets have only a MAC header, a network header, a transport header and a payload but rarely have many options. They would therefore take a "shortcut" through the classification graph, avoiding the extra classification nodes required to process these optional headers.

The average latency of the classification stage 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%. The impact of such a proportion added to the latency of this classification stage would barely be noticeable. The classification latency of packets is therefore largely unaffected by a change in the number of flow filters installed on an active router.

6 Related Work

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 [3], NetScript [9], PLANet [4], SmartPackets [5], accomplish this through software plugins or a similar form of active code integration. The limitations of such plug-in based approaches to extensibility have been revealed in a previous study [10]. The remainder of this section compares further, more closely related approaches to LARA++:

The CANEs execution environment [8], implemented on top of the Bowman NodeOS, provides a composition framework for active services based on the selection and customisation of a generic "underlying program". The underlying program can be tailored for a type of packets or set of streams by injecting customised code into well-defined slots in the 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++. However, in contrast to LARA++, Bowman has a number of restrictions. First, Bowman 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 only dependent on the plug-ins. Second, although Bowman 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. CANEs further restricts service composition. The static nature of the underlying program for any given execution environment is naturally inflexible. One needs to make assumptions about the customisable aspects of the program at instantiation time of the execution environment.

The Router Plugins architecture [2] also uses a plug-in based composition model, whereby an underlying data structure or program defines the "glue" for the service composites. The fact that these composition structures are defined at compile time of the kernel limits extensibility to predefined *gates*. LARA++, by comparison, allows the dynamic extension of the classification graph (i.e. allows the creation of new classification nodes at run-time) and also overcomes the limitations that only one plug-in can be incorporated per gate (i.e. many active components can be inserted per node).

Further related works are the modular router architecture Click [12] and the configurable operating system Scout [11]. 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.

This analysis shows that providing the service composite for plug-in or component-based solutions should 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.

7 Conclusions

In this paper 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. A classification graph, representing the packet processing path on the router, provides the necessary management structure for the integration of the software extensions. 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.

The service composition model is sufficiently flexible to allow the creation of a service composite for each packet passing through the router. Using packet filters to compose active services also has the advantage that active computation can be applied transparently. The application of an active extension is based on the packet content (i.e., any bit pattern can be used to trigger the processing of an active component).

The LARA++ composition model is capable of managing both competition and co-operation between users of an active router. It allows unrelated users to partake in the programming process of the active node in a structured fashion. The model provides sufficient semantics to structure independent software extensions in a meaningful way.

Finally, we have presented the evaluation results of the LARA++ composition framework. The results show that the processing latency imposed by the classifier on an individual packet is less than one hundredth of a millisecond, which is a tiny fraction of the latency introduced by a normal edge router. This shows that the inclusion of the LARA++ composition framework has a negligible impact on the overall latency of the packets passing such a node. The results also demonstrate that the introduction of the flow filters allows the composition model to scale exceptionally well in terms of both the throughput of the active router and the latency of packets being routed, and does so without the number of users significantly reducing performance.

References

1. R. Cardoe, et al., “LARA: A Prototype System for Supporting High Performance Active Networking”, In Proc. of IWAN 99, June 1999.
2. D. Decasper, Z. Dittia, G. Parulkar, B. Plattner, “Router Plug-ins: A Software Architecture for Next Generation Routers”, In Proc. of SIGCOMM, pages 229-240, September, 1998.
3. D.UJ. Wetherall, J.V. Guttag and D.L. Tennenhouse, “ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols”, In Proc. of OPENARCH, April 1998.
4. M.W. Hicks and J.T. Moore and D.S. Alexander and C.A. Gunter and S. Nettles, “PLANet: An Active Internetwork”, In Proc. of IEEE INFOCOM (3), 1124-1133, 1999.
5. B. Schwartz et al., “Smart Packets for Active Networks”, In Proc. of OPENARCH, 1999.
6. S. Schmid, J. Finney, A.C. Scott, W.D. Shepherd, “Component-based Active Network Architecture”, In Proc. of IEEE Symposium on Computers and Communications, July 2001.
7. K. Egevang et al., “The IP Network Address Translator (NAT)”, RFC 1631, May 1994.
8. S. Merugu et al., “Bowman and CANEs: Implementation of an Active Network", In Proc. of 37th Conference on Communication, Control and Computing, September 1999.
9. Y. Yemini and S. da Silva, “Towards Programmable Networks”, In Proc. of IFIP/IEEE International Workshop on Distributed Systems Operations and Management, October 1996.
10. M.W. Hicks and S. Nettles, “Active Networking Means Evolution (or Enhanced Extensibility Required)”, In Proc. of IWAN 2000, October 2000.
11. A. Montz et Al., “Scout: A Communications-Oriented Operating System”, In Operating Systems Design and Implementation, pages 200, 1994.
12. R. Morris, E. Kohler, J. Jannotti, M Kaashoek, “The Click Modular Router“, In Proc. of ACM Symposium on Operating Systems Principles, pages 217-231, December 1999.