GateScript: A Scripting Language for Generic Active Gateways

Hoa-Binh Nguyen and Andrzej Duda

LSR-IMAG Laboratory
Institut National Polytechnique de Grenoble
BP. 72, 38402 Saint Martin d'Hères, France
{Hoa-Binh.Nguyen, Andrzej.Duda}@imag.fr
http://drakkar.imag.fr

Abstract In this paper, we present *GateScript*, a scripting language for active applications to be executed on generic active gateways. Unlike other active networking platforms, it offers a simple scripting language for expressing custom processing of packets at different protocol layers without the need for interpretation of complex protocol data structures. In this way, the user writes statements in a script-like language while using protocol-specific variables and predefined function calls acting on the packet's content. From a textual description, we automatically create a packet parser and reassembler for a given protocol. The parser decomposes PDUs arriving in an active application into protocol variables that can be used in the script language. After processing, outcoming packets are reconstructed from the protocol variables. *GateScript* also enables active applications to react to the state of the environment: they can receive events from monitors and test variables reflecting the state of the environment.
We have designed an architecture for a *generic active gateway* (GAG) that supports *GateScript*. An active application can dynamically install/remove a packet filter that intercepts relevant packets and passes them to the application. We have implemented *GAG* on Linux: its packet forwarding part is implemented in the kernel and all other components as user space processes.

1 Introduction

In our work, we address the problem of customizing user flows in active gateways at the border of the network infrastructure. Unlike traditional proxy nodes, active gateways provide transparent processing of data streams without the need of configuring client hosts. An active gateway may be placed in the access network, for example in the last router connected to a LAN. Many applications may benefit from custom processing physically located close to the client host, especially if it has limited resources. Consider for example small mobile devices that require some adaptation or reaction to changing conditions, and pervasive environments with various devices such as sensors or actuators—an active gateway can provide additional processing in the fixed network infrastructure. In

some cases, we may even want to place the gateway functionality on the end system, so that the user can easily control, filter, or adapt flows arriving to the device.

We have designed and developed *GateScript*, a scripting language for easy programming of active applications that process packets in active gateways. Although there are several platforms for adding programmability to a network node, usually they are programmed in a full-fledged programming language such as Java [8,18], C [5,21], or TCL [1]. Moreover, many platforms require kernel modules or plugins to be developed [13,14], which can be done by experts, but it is too tedious for most of users. With *GateScript* we want to offer a simple scripting language for expressing custom processing of packets at different protocol layers without the need for interpretation of complex protocol data structures. In this way, the user just writes a script that uses variables relative to a given protocol and calls predefined functions working on the packet's content.

More specifically, *GateScript* provides a higher level view than traditional languages and automates the tasks of interpreting/constructing data packets. Coupling protocol variables to values in a received packet is automatically done by a packet parser generated from a formal description of a protocol. The variables available to script programs represent either protocol header fields (e.g. `$http.content_type` for a HTTP Reply or `$tcp.window` for a TCP segment) or elements of the packet data content (e.g. `$html.title` for the title HTML markup). When some values of variables are detected in a packet by the protocol parser, they are made available to a script program so it can take some action or modify them. Simple statements allow to test the values contained in a packet and invoke functions able to modify its content or perform other actions such as packet duplication or drop.

With *GateScript*, we also explore the possibility of coupling the behavior of an active gateway with the state of the environment. Some active applications that we call *proactive* are able to dynamically react and adapt to varying conditions [17]. They cooperate with *monitors*, special entities that observe the state of the network, routers, or hosts. *GateScript* proposes a statement for waiting for an event to execute some operations when a monitor signals an event.

To support *GateScript*, we have designed and implemented an architecture for a generic active gateway called *GAG*. An active application can install a packet filter that recognizes some packets according to the information in the packet header and passes them to the application. Then, it is parsed and the *GateScript* engine interprets the code of a script that processes the packet. Intercepting packets can be activated and disabled dynamically, so that there is no overhead for forwarding packets that do not require active processing.

We have implemented *GateScript* in Java and GAG on Linux. *GateScript* currently integrates two generators of packet parsers: one based on Flavor [6] oriented towards bitstream protocols and a second one based on JavaCC [12] for text oriented protocols. The packet forwarding part of *GAG* is implemented in the kernel and all other components, such as scripts written in *GateScript*, are user space processes. We have experimented with *GateScript* by implementing

several example active applications enhancing the behavior of transport and application level protocols. Even if the performance was not our primary goal, we have evaluated the overhead of intercepting packets in *GAG* and compared the processing performance of *GateScript* with a standard HTTP Java-based gateway such as Muffin [15].

In this paper, we present the main features of *GateScript* and illustrate their use by some examples. We do not cover many other aspects such as secure deployment of scripts on active nodes, control of active applications, node administration, event generation by monitors, and experimentation with active applications specialized for different protocols.

The paper is organized as follows. Section 2 introduces the architecture of *GAG*. We describe *GateScript* in Section 3 and present its implementation in Section 4. Section 5 reports on our experience and presents a first evaluation of the prototype. We discuss the related work in Section 6. Finally, we draw conclusions in Section 7.

2 Generic Active Gateways

A generic active gateway needs to provide general support for processing the content of different data flows and customizing the behavior of protocols. We consider transparent gateways that are network nodes acting in a similar way to routers: data packets are not directly addressed to them, rather they are forwarded to a destination after processing some of them. The gateway forwards packets in a usual way based on standard routing tables or according to the effect of active packet processing.

Usually a gateway implementing active applications performs some packet parsing, processing, and reconstruction while all these functionalities are combined in the same piece of code. Our approach consists of separating packet parsing and reconstruction from data processing to make them generic so that they can be used for any bit oriented or textual protocol. The generic part of an active gateway can be specialized for a given protocol or data flow based on the structure of a PDU (*Protocol Data Unit*) defined by the protocol[1]. Examples of such a use are intelligent HTTP, RTSP, or SIP proxies, media transcoding gateways (e.g. from HTML to WML), or adaptation gateways (e.g. from MPEG to H.263).

An active gateway needs to support the following functionalities (we illustrate them with examples in the context of HTTP when relevant):

- Active applications need to execute some code upon the arrival of a packet or when the state of the system changes (e.g. when receiving a HTTP Reply, check for the MIME type of the message body and filter out all images). The

[1] We use the term of a packet to designate the PDU entering an active gateway. A packet may contain encapsulated PDUs defined by higher level protocols, e.g. a TCP segment containing a HTTP Reply. When describing the protocol parsing part within *GateScript*, we will use the term of a PDU.

code of an active application should involve variables variables proper to a given protocol (e.g. an active application should be able to test the MIME type of the HTTP message body).

- The value of a variable used in an active application should be set to the value of a PDU field assigned when a packet is received by the gateway (e.g. variable `$http.content_type` should be set to the value `image/jpeg` for a HTTP Reply containing a JPEG image).
- A rich library of functions able to process specific data types should be available to active applications (e.g. `ReduceImageSize` or `TranscodeVideo` for processing objects in a HTTP Reply).
- We need means for dynamically enable or disable processing of packets passing through a gateway to obtain good performance when custom processing is not required.
- Active applications require support for reacting to changes in their environment such as network congestion, host disconnection, lack of resources (e.g. when a client host changes the access network, it may request to change processing of packets, because of the increased available bandwidth).

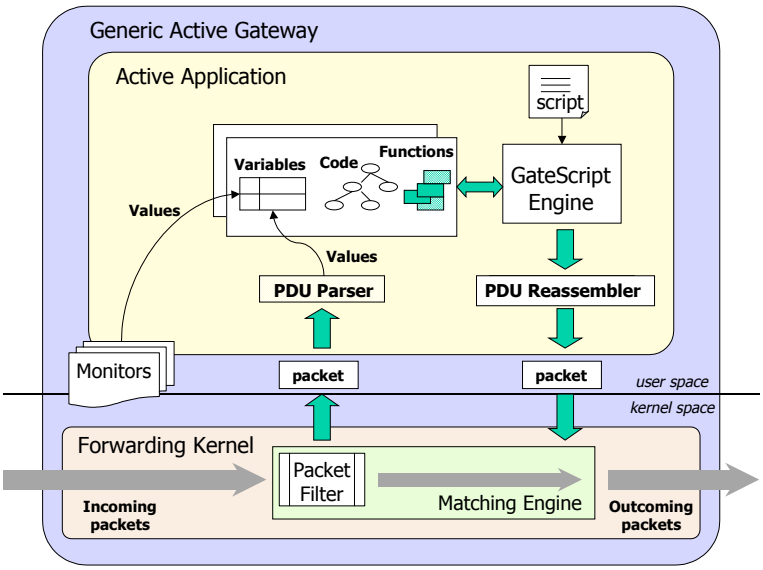

Fig. 1. Architecture of GAG

The architecture of *GAG*, a generic active gateway supporting *GateScript* is presented in Figure 1. *GAG* is composed of the following entities:

- *Active applications* that process some packet data. They are programmed using the *GateScript* scripting language. The script program involves variables proper to a given protocol or representing the state of the environment.
- A *GateScript engine* for executing a script program once the variables used in the program have their values assigned. It couples a script program with the variables recognized in data packets and with the functions able to process them.
- *Protocol variables* that represent fields defined in the PDU structure of a given protocol or some parts of the packet content. Protocol variables are predefined for any given protocol.
- A *PDU parser* for recognizing the structure of a given PDU contained in a packet, parsing the data contents, and setting up variables used by the script program of active applications.
- A *PDU reassembler* to reconstruct a data packet from the variables used by the script program (the inverse function to the PDU parser). The PDU parser and reassembler are automatically created from the description of a given protocol.
- *Processing functions*, an extensible library of useful functions that allow to process data packets. The functions are proper to a given protocol or to a data format. They are supposed to be developed by an expert Java programmer, because they may require an extensive knowledge of a protocol, system calls, and programming conventions (parameter passing, operations allowed on the *PDU context*, cf. Section 4).
- *Monitors* able to detect varying conditions in the environment (network, gateways, devices, services, hosts, users). In some cases it is important that an active application reacts to the change of the system state. A monitor can signal an active application by sending an event that can be tested in the script program.
- *A matching engine* that allows to dynamically install and uninstall *packet filters* responsible for intercepting packets and passing them to active applications. An active application can decide when to install or uninstall a packet filter so that when intercepting packets is not needed, there is no overhead of passing packets to the user space. Packets that do not match any filter are forwarded in the standard way.

Active applications can be loaded or unloaded dynamically into the active gateway. Some active applications that we call *proactive* cooperate with monitors and are able to dynamically react and adapt to varying conditions.

3 GateScript Language

GateScript is a scripting language for programming active applications that process packets in *GAG* gateways. Below we review the main constructs of the *GateScript* language (see Appendix for more formal description).

3.1 Statements

A *GateScript* program is composed of statements. Each statement can test the values of variables representing specific PDU fields and invoke appropriate functions. User defined variables can be declared and initialized using the `set` statement and substitute to their values when preceded by `$`. There are several types of statements:

- *assignment statement* to assign a value to a variable, e.g.

```
set State $AckState;
```

- *conditional statement* to execute one of two groups of statements based on the test of a condition, e.g.

```
if ($ip.destination_address = $Client) then
   WriteToCache;
endif
```

- *function call* to invoke a function with some arguments, e.g.

```
CheckIfExistPacket $tcp.Ack_Number
```

- *event statement* to wait for a condition related to an event and to execute a statement when the event is received, e.g.

```
onEvent $EventName = "ClientDisconnects" then
   PacketFilter "add $ClientIPAddress";
endEvent
```

  When a monitor signals event `ClientDisconnects`, the application executes function `PacketFilter` to install a packet filter for intercepting packets containing the IP address of the client. In this way, the active application starts receiving packets on behalf of the client, which can be for instance stored in a cache for later delivery.

3.2 Variables

There are three kinds of variables:

- *user defined variables* that are not related to any protocol, e.g. variable `$State` given in the example above.
- *protocol-related variables* that represent PDU fields or data content values, e.g. variable `$tcp.SYN` representing the SYN TCP flag. The PDU parser assigns values recognized in a packet to such variables each time a new packet arrives in the gateway and is passed to the active application.
- *monitor variables* that represent the state of some environment conditions, e.g. variable `$Disconnected` becomes true if a client host probed by a monitor cannot be reached (we assume that we use a monitor able to detect such a condition).

In *GateScript* PDUs arriving in an active application are decomposed into protocol variables that can be processed in script statements. After processing packets are completely reconstructed from the variables on the way out.

Variables can be combined by using operators to form expressions. Function calls in expressions are separated from operators with square brackets.

3.3 Events

When a monitor detects a modification in the state of the environment, it signals an application with an *event*. An event has a name and a list of variables. Consider the following example: an application subscribes to a congestion monitor that detects congestion conditions in the network and passes some information about the available resources:

```
onEvent $EventName = "Congestion" then
   AdaptEncoding $AvailableBandwidth;
endEvent
```

The monitor signals the `Congestion` event and makes the current value of the available bandwidth accessible. Upon this event, the monitor invokes a function to adapt encoding.

3.4 Static Attribute

Statements may be static or not. A static statement is executed only once per execution of a script, whereas a non static statement is executed each time a packet is received and parsed. Such an execution semantics is needed when we want to initialize some variables or start monitors. It allows keeping a limited state during the execution of a script. Any statement can be static. As packet processing is the main goal of active applications, statements are not static by default. Consider the following example:

```
if ($tcp.SYN = 1) then
   static set Client $ip.destination_address;
   set State $SynState;
endif
```

If the active application receives a SYN TCP segment, it stores the IP destination address in the variable `$Client` and the current state of the connection in the variable `$State`. The first assignment will be executed only once, while the second one, every received SYN segment.

We can characterize *GateScript* as an active platform supporting limited statefull packet processing—limited by the script language itself, because the static attribute only allows initializing some variables of a script. However, if required, it is extendable by functions such as `WriteToCache`.

3.5 Examples

The following three examples concern pervasive environments in which computer devices connected via different types of networks provide the user with some augmented functionalities. Due to energy or capacity limitations pervasive environments and mobile components usually require some additional processing to be done in the fixed network infrastructure by a proxy node or a gateway.

The *GateScript* program presented below corresponds to TCP snooping [2]. It operates in a gateway located between the wired and the wireless parts of the network. It caches TCP packets in order to respond more quickly to ACK packets from a mobile client.

```
static set State 0;
static set SynState 1;
static set AckState 2;
static set EtablishedState 3;
if ($tcp.SYN = 1) then
   static set Client $ip.destination_address;
   set State $SynState;
endif
if ($tcp.SYN = 1) and ($tcp.ACK = 1) and
   ($State = $SynState) then
   set State $AckState;
   ForwardPacket;
   return;
endif
if ($State = $AckState) and ($tcp.ACK = 1) then
   set State $EtablishedState;
   ForwardPacket;
   return;
endif
if ($State = $EtablishedState) then
   if ($ip.destination_address = $Client) then
      WriteToCache;
   endif
   if ($ip.source_address = $Client) then
      if ([CheckIfExistPacket $tcp.ack_number]) then
         ForwardFromCacheToClient $tcp.ack_number;
         return;
      endif
   endif
endif
ForwardPacket;
```

The script performs TCP snooping for one TCP connection with a given client host. At the beginning, it defines four variables to represent the state of a TCP connection: `$State`, `$SynState`, `$AckState`, and `$EtablishedState`. For each segment during the three-way handshake, the state is modified. When the connection is established, the active application caches all the packets going to

the given client host and forwards them to the destination. When it detects by means of the TCP ACK that the next not yet acknowledged segment resides in the cache, it forwards it directly to the client (the TCP ACK number corresponds to the next not yet received segment), and the ACK segment is dropped. In this way, the client quickly obtains a retransmitted segment from the gateway instead from the source.

The next example presents a caching service for a mobile host. It subscribes to a `$PresenceMonitor` that checks for the presence of a client host by periodically sending ICMP Echo Request. The state of the client host is represented in the variable `$Disconnected` updated by the monitor. When the state changes, an event is sent to the active application: `ClientDisconnects` or `ClientConnects`. Based on these events, the application enables or disables packet intercepting in the kernel. At the beginning, when the client host is connected, the application is running and packets go through the gateway without processing. When the monitor detects the disconnection of the client host, it signals the application that installs a packet filter for the IP address of the client. In this way, the application starts receiving packets. Each packet is stored in a cache. When the client host connects again, packets are forwarded to the host and the packet filter is deleted so that packets are no longer processed by the active application.

```
static set Client "client.host.edu";
static PresenceMonitor $Client;
onEvent $EventName = "ClientDisconnects" then
   PacketFilter "add $Client";
endEvent
onEvent $EventName = "ClientConnects" then
   PacketFilter "delete $Client";
endEvent
if $Disconnected then
   WriteToCache;
else
   ForwardCacheToClient;
endif
```

The following example shows an active application that detects high temperature and generates a fire alarm. First, it calibrates a raw measurement from a temperature sensor, then it tests to detect whether it is higher than a predefined threshold, and generates an event handled by applications that subscribed to it. If the temperature is low, the packet is dropped. We assume a simple packet structure with two fields: the sensor id and the raw measurement of the temperature.

```
static set FireAlarmThreshold 50;
set Temperature   [Calibrate $RawMesurement];
if $Temperature > $FireAlarmThreshold then
  GenerateEvent "FireAlarm" [GetLocalization $SensorID];
else
  DropPacket;
endif
```

The last examples illustrate a HTTP gateway developed using *GateScript*—it scans the HTTP traffic on behalf of a user and performs customization (filtering out ad banners, reducing image size, etc.). Table 1 lists the functions developed to process HTTP typed objects.

Table1. Processing functions for HTTP.

Name	Functionality
RemoveTag	Remove a tag
RemoveColor	Remove color information
ContentDiscard	Discard the data
ReduceImageSize	Reduce image size
ColorToGreyScale	Transcode to grey scale
ColorToBW	Transcode to black and white
JPEGToGIF	Transcode JPEG to GIF
GIFToJPEG	Transcode GIF to JPEG
BreakPage	Break page
FilterHtmlFrame	Filter out a frame
FilterHtmlTable	Filter out a table

The examples below deal with the content of Web pages. The first one filters images by removing all image tags from an HTML page and by discarding all image objects (`RemoveTag` function makes use of a HTML parser on a HTTP object of type `text/html`).

```
if $http.content_type contains "text/html" then
   RemoveTag "img";
endif
if $http.content_type contains "image" then
   ContentDiscard;
endif
```

The next example reduces the size of JPEG images by half if the original image is greater than 1 Kbyte.

```
if (($http.content_type = "image/gif") or
    ($http.content_type = "image/jpeg")) and
    ($http.content_length > 1000) then
       ReduceImage 0.5;
endif
```

4 Implementation of *GAG* and *GateScript*

4.1 *GAG* Prototype on Linux

We have implemented *GAG* on Linux (its first version was called ProAN [17]). Linux is a good candidate for such an active node because of its properties: packet

forwarding support, loadable kernel modules, and the ease of modifying the kernel behavior. The forwarding part of our architecture with the matching engine is implemented in the Linux kernel. Each active application is implemented as a user space process and may receive packets belonging to a flow defined by some packet properties such as source or destination address. An active application may dynamically install and uninstall packet filters in the matching engine. When installed, a packet filter passes matching packets to the application.

The matching engine uses Netfilter [16], the support for custom processing of packets in the kernel. It allows users to hook extended modules in the packet forwarding path and to pass packets of a flow to a process in the user space for further processing. After processing packets are re-injected into the kernel, however the process cannot inject newly created packets into kernel so that some processing such as packet duplication is impossible with standard Netfilter.

Another limitation of Netfilter is that only one process in the user space may receive packets from the kernel. IP Queue Multiplex Daemon (`ipqmpd`) [11] adds the possibility of passing packets from different flows to different user processes. It communicates with user processes using sockets or other IPC mechanisms. This is inefficient, because packets must re-enter the kernel before arriving in the destination user process.

To obtain better performance of *GAG*, we have modified Netfilter to pass different packet flows directly to the right user process without going through the multiplexer daemon. We use `iptable` to mark packets with the corresponding process ID (PID) of the active application. When the `ip_queue` module receives the packets, it detects and forwards them directly to the right process. We have also modified the `ip_queue` module to support more than three modes of operation (drop a packet, pass the kernel metadata of a packet to the user process, pass the metadata and the packet payload to the user process)—the standard `ip_queue` module always keeps a copy of a packet passed to a user space process. A module can only modify the payload of packets and it is not possible for a module to inject newly created packets into the kernel. With our modification, when a packet is passed to a user space process in this mode, it uses a new verdict value (`NF_INJECT`) to inject a new packet into the kernel. Our version of the modified `ip_queue` currently supports 40 queues in the `ip_queue` module.

4.2 *GateScript* Implementation

We have implemented *GateScript* in Java. A user space process implementing each active application contains the *GateScript* engine as well as PDU parsers and reassembler. A script program is compiled into an intermediate form interpreted by the *GateScript* engine. The compilation is done only once per each application activation. Protocol variables exist in the intermediate form, however their values become assigned when a packet arrives in the application.

Internally, *GateScript* makes use of a structure containing the set of variables corresponding to a PDU: the *PDU context*. It is a hashed table with all protocol-related variables obtained from the parsing of a PDU. When a protocol parser receives a PDU, it parses it and creates a PDU context. The *GateScript* engine

uses it when executing a program script and passes it to any invoked function, which can change the variable values or may add more variables if necessary (when developing functions, the programmer needs to carefully handle the PDU context).

We use Flavor [6] to describe the structure of bitstream oriented protocols such as IP, TCP, UDP, RTP, or X Window. The PDU description in Flavor is compiled to generate a C++ or a Java class, integrated with the *GateScript* engine to parse a bitstream, recognize the defined fields, and obtain their values. Table 2 presents the description of an IP packet containing a TCP segment in Flavor.

For text oriented protocols such as HTTP, FTP, SMTP, SNMP, RTSP, or SIP we generate parsers using JavaCC [12]. We describe a given protocol in a syntax description file proper to JavaCC. Table 3 presents the description of the HTTP protocol. It defines the structure of the HTTP PDUs and couples the parser and reassembler with the *GateScript* engine by means of the *PDU context*. The header attributes become available for scripts in variables whose names are HTTP attributes (because of compatibility problems with Java, we replace dash with underscore, for example, the `Content-Type` header attribute is represented by the `$http.content_type` variable).

5 Evaluation

We have experimented with *GateScript* by implementing active applications enhancing the behavior of several protocols: an active gateway for HTTP that scans the HTTP traffic on behalf of a user and performs customization (filtering out ad banners, reducing image size), a multiplexer of the X Window protocol able to replicate a window of a remote application on different X displays, a SIP gateway that performs user defined actions on SIP INVITE messages, an MPEG adaptation gateway that monitors the RTCP reports to detect degrading reception conditions and transcode MPEG to H.263, and a snooping wireless adaptation gateway that acts at the IP and TCP layers in a 802.11 WLAN cell to provide statistical QoS by limiting the rate of TCP flows through modification of the announced window size.

Although the best performance was not our primary goal, we wanted to obtain a first evaluation to see if the overhead of *GateScript* is not too prohibitive compared to standard gateways. Therefore we separated *GateScript* from *GAG* and evaluated them independently. We have measured the performance of a HTTP gateway programmed using *GateScript* on a 1.06 GHz Pentium III PC with 248 MB RAM running Windows XP and compared with the performance of Muffin [15], a public Java HTTP proxy. In this experiment, our gateway operated as a proxy without the packet matching kernel: all packets go through a user process running *GateScript* engine. Both tested tools are entirely developed in Java and executed with Java 2 SDK 1.4.1.

In the test, we have downloaded pages from a popular Web server through the gateways that processed HTTP Replies: each page has been analyzed and

Table2. TCP/IP PDU described in Flavor.

```
class TCP_IP {
unsigned int(4) ip.version;
unsigned int(4) ip.hdr_length;
unsigned int(8) ip.service_type;
unsigned int(16) ip.total_length;
unsigned int(16) ip.identification;
unsigned int(3)  ip.flags;
unsigned int(13) ip.fragment_offset;
unsigned int(8)  ip.ttl;
unsigned int(8)  ip.protocol;
unsigned int(16) ip.header_checksum;
unsigned int(32) ip.source_address;
unsigned int(32) ip.destination_address;
if (ip.hdr_length>5)
  { unsigned int(8) ip.options[(ip.hdr_length*4-20)]; }

unsigned int(16) tcp.source_port;
unsigned int(16) tcp.destination_port;
unsigned int(32) tcp.sequence_number;
unsigned int(32) tcp.ack_number;
unsigned int(4) tcp.data_offset;
unsigned int(6) tcp.reserved;
unsigned int(1) tcp.URG;
unsigned int(1) tcp.ACK;
unsigned int(1) tcp.PSH;
unsigned int(1) tcp.RST;
unsigned int(1) tcp.SYN;
unsigned int(1) tcp.FIN;
unsigned int(16) tcp.window;
unsigned int(16) tcp.TCP_Checksum;
unsigned int(16) tcp.urgent_pointer;
if (tcp.data_offset>5)
  { unsigned int(8) tcp.options[(tcp.data_offset-5)*4]; }

unsigned int(8) tcp.data[ip.total_length-(ip.hdr_length*4)
                                -(tcp.data_offset*4)]; };
```

Table3. HTTP PDU described in JavaCC.

```
options
{ USER_CHAR_STREAM = true; }

PARSER_BEGIN (HTTPResponseParser)
public class HTTPResponseParser
{ public Map PDUcontext; }
PARSER_END (HTTPResponseParser)

void HTTPParse(): {}
{{ PDUcontext = new HashMap(); }
Status_Line() <CRLF>
( Header() <CRLF> )*
<CRLF>
Message_Body()
}

void Status_Line() :
{ String version,reason_phrase;
int status_code; }
{ version = string()  <SPACE>
{ PDUcontext.put("version",version); }
status_code = number() <SPACE>
{ PDUcontext.put("status_code",
                 new Integer(status_code)); }
reason_phrase = String();
{ PDUcontext.put("reason_phrase", reason_phrase); }}


void Header():
{ String header,value; }
{ header = string() ":" value = string()
{ header = header.replace('-','_');
PDUcontext.put(header,value); }}

void Message_Body():
{ byte[] data; }
{ data = byte_array()
{ PDUcontext.put("content",data); }}
```

all images have been filtered out. Figure 2 compares the download delay for the *GateScript* gateway and Muffin in function of different page sizes. We can see that the overall performance in terms of delay remains comparable.

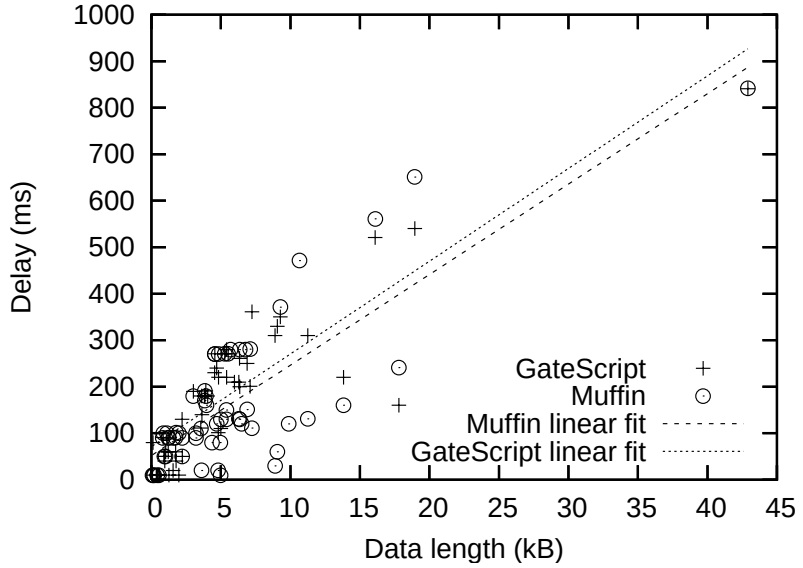

Fig. 2. Performance of Gatescript vs. Muffin, image elimination

To evaluate the *GAG* implementation on Linux, we have measured the performance of packet forwarding and passing them to the user space on a 800 MHz Pentium III PC with 128 MB RAM running Red Hat 7.2. Figure 3 presents the delay of packet forwarding in function of the packet size for two cases: in the first one, packets enters the kernel and they are just forwarded to the destination (no active application installed); in the second case, a packet filter is installed to intercept packets and pass them to an active application (active flow - active application installed). It does not perform any processing and just re-injects packets into the kernel for further forwarding. The difference between the two curves represents the overhead of passing a packet to the user space. These results show that when an active application does not install a packet filter, data flows do not incur any performance penalty. We can also see from the figure that in the second case the overhead has only impact on data flows on which active applications need to perform useful processing: the delay for a passive flow (the flow for which packets are not intercepted by its packet filter) stays small even if the packets of an active flow are processed by the associated active application.

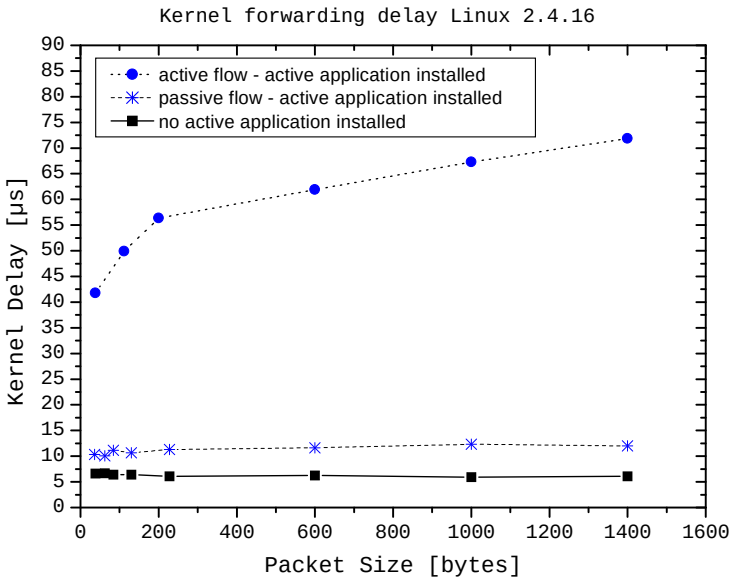

Fig. 3. Performance of passing packets to the user space

5.1 Limitations of the Prototype

At the moment only one script can access a packet. We have not dealt with multiple scripts processing the same packet yet—this requires solving the problem of the processing order, defining allowed operations on the packet, and eventual communication between scripts.

We currently use only one protocol parser per script. It is fairly easy to increase their number if they are of the same nature, e.g. two bit-oriented or two text-oriented protocols. However, coupling Flavor parsers with those generated by JavaCC is more difficult and needs more work.

At the current stage, GateScript does not automatically handle a PDU split over multiple packets, e.g. a HTTP Reply containing a large image. If really needed, it can be done by programming a function that keep state between two packet arrivals: it can store packets with fragments, reorder them if needed, and finally process the whole PDU.

6 Related Work

Research in active networking has brought in several platforms supporting active applications and services. Many of them use full-fledged programming languages such as Java ([8,18]), C ([5,21]), or TCL [1]. However, as said previously, we think

that a specialized scripting language with automatic parsing of PDU fields like *GateScript* provides a more flexible tool for programming active gateways. As to Java, we consider it as an excellent language for developing *GateScript* internal functions, but we do not need all its complexity to program active applications, in which for example, the programmer would have to deal with exceptions and all Java keywords.

There are several other specialized languages for active networking platforms. PLAN [10] and *GateScript* have different objectives: PLAN is a language for programming active packets while *GateScript* is used for programming active applications that process regular (passive) packets in a transparent way.

Netscript [23] is a connector-oriented language for composing active applications from smaller components called Netscript boxes. The main difference between Netscript and *GateScript* is that Netscript is suitable for composing extensible routers with dynamic protocol stacks, while *GateScript* is mainly used to customize a flow at a given protocol layer without cumbersome interpretation of the incoming data and encoding the outgoing data.

Unlike several existing platforms that require developing kernel modules or plugins [13,14], we place custom processing in the user space. Other platforms such as ALAN [8] or AS1 [1] have adopted a similar goal, but they provide support for active services working mostly at the application layer. The *GateScript* support for packet processing in the user space does not limit the scope of programmability to application layer protocols—it can deal with packets of any layer ranging from network to application.

Adaptation proxies have been extensively studied in the context of HTTP and content distribution. *CacheL* is a language that enables creating customizable caching policies based on different cache events and a set of predefined actions [3]. The *Open Pluggable Edge Services (OPES)* [20] IETF working group is defining an architecture that allows services to operate on application data when they transit across an intermediate node (a proxy or a surrogate server). In some sense, OPES devices (intermediaries supporting the OPES architecture) are programmable by means of a rule language that may depend on some protocol properties such as HTTP headers.

Several content adaptation proxies have been developed for image or video transcoding for wireless clients [1,7,9,4,22]. However, most of them are fixed in the sense that their functionalities cannot be dynamically extended nor customized—they are only configurable, but not programmable.

The programmable video gateway [19] uses a scripting language to program a video gateway. The focus here is on the video data only and not on the protocol data structures. By integrating a parser of a given protocol, *GateScript* can deal with data packets not only at the application layer.

7 Conclusion

In this paper, we have presented *GAG*, a generic active gateway that supports *GateScript*, a scripting language for easy programming of custom processing on data packets. Unlike other active networking platforms it is

- *generic and easy to use*: we automatically create a PDU parser and reassembler for the protocol that needs to be enhanced with custom processing, and provide useful functions to operate on the content of PDUs; in this way, the programmer may focus on PDU processing and not on cumbersome and error prone interpretation of incoming data packets.
- *reactive*: in addition to custom processing of packets, active applications are able to react to the state of the environment: they can receive events from monitors and test variables reflecting the state of the environment;
- *flexible*: *GateScript* allows processing at different protocol layers ranging from network to application levels.

GateScript makes the development of active applications fairly easy within the grasp of a user not familiar with expert network programming. Our examples show that even complex problems such as snooping TCP can be easily programmed in *GateScript*.

GateScript can be especially useful for creating personal communication gateways on mobile computers. In this case, we place the active gateway on a mobile host so that standard applications may benefit from network customization of flows entering the host. The user can easily specify the behavior of the gateway by injecting scripts into the *GateScript* engine. In this way, we can handle configuration modifications while the host changes the point of attachment to the global network. We plan to experiment with *GateScript* to develop such personal communication gateways.

We also need to get more insight into the performance of our prototype and its ability to handle an increasing number of flows, packet filters, and active applications.

8 Acknowledgments

This work has been partially supported by France Telecom R&D.

Appendix

```
GateScriptProgram  = Statements
Statements = ([static] Statement ‘‘;’’)*
Statement = AssignStatement
    | IfStatement
    | FunctionStatement
```

```
AssignStatement = "set" Variable  Expression
IfStatement = ‘‘if’’ Expression ‘‘then’’
Statements  [ElseStatement] ‘‘endif’’
ElseStatement = ‘‘else’’ Statements
OnEventStatement =‘‘onEvent’’ Expression ‘‘then’’
Statements ‘‘endEvent’’
FunctionStatement = FunctionName (Expression)*
Expression  = ConstantValue
       | ‘‘$‘‘Variable
              | Expression BinOp Expression
              | UnOp Expression
              | ‘‘[‘‘ FunctionStatement ‘‘]‘‘
       | ‘‘(‘‘ Expression ‘‘)’’
Variable     = Identifier
FunctionName = Identifier
Identifier      = Letter (Letter | Digit)*
ConstantValue   = Boolean | String | Integer |
                  Real | Character
BinOp        = ‘‘+’’ | ‘‘-’’ | ‘‘*’’ | ‘‘/’’
               | ‘‘<’’ | ‘‘<=’’ | ‘‘=’’ | ‘‘!=’’
               | ‘‘>’’ | ‘‘>=’’
               | ‘‘and’’ | ‘‘or’’ | ‘‘contains’’
UnOp         = ‘‘-’’ | ‘‘+’’ | ‘‘!’’
Boolean      = ‘‘true’’ | ‘‘false’’
String       = ‘‘"’’  (~[‘‘"’’,‘‘\‘‘,‘‘\n‘‘,
               ‘‘\r‘‘,"["]"])* ‘‘"’’
Integer      = Digit (Digit)*
Real         = Integer [Fraction] [Exponent]
Fraction     = ‘‘.’’ Integer
Exponent     = (‘‘e’’ | ‘‘E’’) [‘‘+’’ | ‘‘-’’]
               Integer
Digit      = ["0"-"9"]
Letter       = ["a"-"z", "A"-"Z"] | "_"
```

References

1. E. Amir, S. McCanne, and R. Katz. An Active Service Framework and its Application to Real-time Multimedia Transcoding. *ACM Communication Review*, 28(4):178–189, Sep. 1998.
2. H. Balakrishnan, S. Seshan, and R. H. Katz. Improving Reliable Transport and Handoff Performance in Cellular Wireless Networks. *ACM Wireless Networks*, 1(4), December 1995.
3. J. F. Barnes and R. Pandey. CacheL: Language Support for Customizable Caching Policies. In *the 4th International Web Caching Workshop*, San Diego, California, 1999.
4. S. Chandra, C.S. Ellis, and A. Vahdat. Multimedia Web Services for Mobile Clients Using Quality Aware Transcoding. In *the 2nd ACM International Workshop on Wireless and Mobile Multimedia (WoWMoM'99)*, Seattle, Washington, USA, August 1999.

5. D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers. *IEEE/ACM Transaction on Networking*, Feb. 2000.
6. A. Eleftheriadis and D. Hong. Flavor: A Language for Media Representation. In *the Fifth ACM International Conference on Multimedia*, Seattle, Washington, 1997.
7. A. Fox and E.A. Brewer. Reducing WWW Latency and Bandwidth Requirements by Real-Time Distillation. In *the 5th International WWW Conference*, Paris, France, May 1996.
8. M. Fry and A. Ghosh. Application Level Active Networking. *Computer Networks*, 1999.
9. R. Han et al. Dynamic Adaptation in an Image Transcoding Proxy for Mobile Web Browsing . *IEEE Personal Communications Magazine*, 5(6):8–17, December 1998.
10. M. Hicks et al. PLAN: A Programming Language for Active Networks. In *Proc. ICFP '98*, 1998.
11. `ipqmpd` - IP Queue Multiplex Daemon. `http://gnumonks.org/projects/`.
12. Java Compiler Compiler (JavaCC) - The Java Parser Generator. `http://www.webgain.com/products/java_cc/`.
13. R. Keller, L. Ruf, A. Guindehi, and B. Plattner. PromethOS: A Dynamically Extensible Router Architecture Supporting Explicit Routing. In *IWAN02 - Fourth Annual International Working Conference on Active Networks*, Zurich, Switzerland, December 4-6, 2002.
14. A. Kind, R. Pletka, and M. Waldvogel. The Role of Network Processors in Active Networks. In *IWAN03 - Fourth Annual International Working Conference on Active Networks*, Kyoto, Japan, Dec. 2003.
15. Muffin - a World Wide Web Filtering System. `http://muffin.doit.org/`.
16. The NETFILTER/IPTABLES project. `http://netfilter.samba.org` .
17. H-B. Nguyen and A. Duda. ProAN: an Active Node for Proactive Services in Pervasive Environments. In *The 2nd International Workshop on Active Network Technologies and Applications (ANTA 2003)*, Osaka, Japan, May 2003.
18. E. Nygren, S. Garland, and M. F. Kaashoek. PAN: A High-Performance Active Network Node Supporting Multiple Mobile Code Systems. In *The Second IEEE Conference on Open Architectures and Network Programming-OpenArch99*, New York, New York, March 1999.
19. W. T. Ooi, R. Renesse, and B. Smith. Design and Implementation of Programmable Media Gateways. In *the 10th International Workshop on Network and Operating System Support for Digital Audio and Video*, Chapel Hill, North Carolina, June 2000.
20. The Open Pluggable Edge Service (OPES). `http://www.ietf-opes.org` .
21. S. Schmid, T. Chart, M Sifalakis, and A. C. Scott. Flexible, Dynamic and Scalable Service Composition for Active Routers. In *IWAN02 - Fourth Annual International Working Conference on Active Networks*, Zurich, Switzerland, December 4-6, 2002.
22. J. Seitz, N. Davies, M. Ebner, and A. Friday. A CORBA-based Proxy Architecture for Mobile Multimedia Applications. In *MMNS'98 - 2nd IFIP/IEEE International Conference on Management of Multimedia Networks and Services*, Versailles, France, November 1998.
23. Y. Yemini and S. Silva. Towards Programmable Networks. In *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, L'Aquila, Italy, October 1996.