Gary J. Minden
Kenneth L. Calvert
Marcin Solarski
Miki Yamamoto (Eds.)

LNCS 3912

Active Networks

IFIP TC6 6th International Working Conference, IWAN 2004
Lawrence, KS, USA, October 2004
Revised Papers

ifip

Springer

Gary J. Minden Kenneth L. Calvert
Marcin Solarski Miki Yamamoto (Eds.)

Active Networks

IFIP TC6 6th International Working Conference, IWAN 2004
Lawrence, KS, USA, October 27-29, 2004
Revised Papers

Springer

Volume Editors

Gary J. Minden
University of Kansas
Information and Telecommunications Technology Center
2335 Irving Hill Road, Lawrence, KS 66045, USA
E-mail: gminden@ittc.ku.edu

Kenneth L. Calvert
University of Kentucky
Department of Computer Science, Laboratory for Advanced Networking
Hardymon Building, 301 Rose Street, Lexington, KY 40506-0495, USA
E-mail: calvert@netlab.uky.edu

Marcin Solarski
Deutsche Telekom A.G.Laboratories
Ernst-Reuter-Platz 7, 10587 Berlin, Germany

Miki Yamamoto
Kansai University
Department of Electrical Engineering and Computer Science
3-3-35 Yamate-cho, Suita, Osaka 564-8680, Japan
E-mail: yama-m@ipcku.kansai-u.ac.jp

Library of Congress Control Number: 2007923238

CR Subject Classification (1998): C.2, D.2, H.3.4-5, K.6, D.4.4, H.4.3

LNCS Sublibrary: SL 5 – Computer Communication Networks and Telecommunications

ISSN 0302-9743
ISBN-10 3-540-71499-5 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-71499-6 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12038831 06/3142 5 4 3 2 1 0

Preface

We are pleased to present to you the proceedings of the sixth Annual International Working Conference on Active Networks, which took place in October 2004 at The Information and Telecommunications Technology Center, The University of Kansas, USA. The proceedings of IWAN 2004 mark a transition between the funded active networking programs in Europe, Japan, and the United States and a strong, continued interest in the architectures of programmable networks.

The technical committee accepted 14 papers for presentation from 32 submitted papers. The papers are organized into sections on active network systems and architectures, security in active networking, active network applications, mobile active networks and active network management. Whereas the contributions on active network architectures and management build upon mature concepts devised in the previous years and are incremental follow-ups of the related research, the security considerations are of primary importance to the active networks practitioners. The papers on mobile applications of active networks, like TCP gateways between wireless and wireline networks, provide additional inspirations to the active network researchers.

Featured in the program were a keynote address by Jonathan M. Smith of DARPA and two invited papers, one by Takashi Egawa, Yoshiaki Kiriha, and Akira Arutaki on "Tackling the Complexity of Future Networks", and one by Bernhard Plattner and James Sterbenz, titled "Programmable Networks: Alternative Mechanisms and Design Choices". Based on the reviewer feedback, the authors of the paper "Dynamic Link Measurements Using Active Components", Dimitrios Pezaros, Manolis Sifalakis, Stefan Schmid and David Hutchison, all of Lancaster University, were awarded this year's Best Paper Award. During the two days of the conference there were several lively discussions, including one at the end of the first day on the scope and future of IWAN itself. The social event, barbecue at the Circle-S ranch, provided a most enjoyable venue for discussion and collegiality.

We would like to thank the members of the Program Committee for their excellent work in reviewing, selecting, and in some cases shepherding papers for the program. V. Rory Petty and F. "Ted" Weidling supported the Web site and conference organization.

We appreciate the work of all the authors, they are the core of this workshop and proceedings. The participation of all the attendees made an outstanding conference. Enjoy the fruits of all their labors. We trust you will find these proceedings interesting.

October 2004

Gary J. Minden
Ken Calvert
Marcin Solarski
Miki Yamamoto

Organization

IWAN 2004 was organized by The Information and Telecommunications Technology Center (ITTC) at The University of Kansas. We would like to acknowledge the support of our sponsors, The Information and Telecommunications Technology Center, The University of Kansas, and Hitachi Ltd., and we thank them for their contributions. Their support and the research presented in these proceedings continue to demonstrate international interest in active networking.

Executive Committee

General Chair	Gary J. Minden, The University of Kansas, USA
General Co-chairs	Tadanobu Okada, NTT, Japan Bernhard Plattner, ETH Zürich, Switzerland
Program Co-chairs	Marcin Solarski, Fraunhofer FOKUS, Germany Ken Calvert, The University of Kentucky, USA Miki Yamamoto, Osaka University, Japan

Technical Program Committee

Stephane Amarger
Bobby Bhattacharjee
Matthias Bossardt
Bob Braden
Torsten Braun
Marcus Brunner
Ken Calvert
Hermann DeMeer
Takashi Egawa
Ted Faber
Mike Fisher
Alex Galis
Anastasius Gavras
Jim Griffioen
Robert Haas
Toru Hasegawa
Michael Hicks
David Hutchison
Javed Kahn
Andreas Kind
Yoshiaki Kiriha
Akira Kurokawa
Laurent Lefevre
John Lockwood
Douglas Maughan
Gary Minden
Toshiaki Miyazaki
Sandy Murphy
Scott Nettles
Bernhard Plattner
Guy Pujolle
Lukas Ruf
Nadia Shalaby
Yuval Shavitt
Marcin Solarski
James Sterbenz
Christian Tschudin
Naoki Wakamiya
Marcel Waldvogel
Tilman Wolf
Miki Yamamoto
Krzysztof Zielinski
Martina Zitterbart

Sponsoring Institutions

The University of Kansas
The Information and Telecommunications Technology Center
Hitichai

Table of Contents

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).

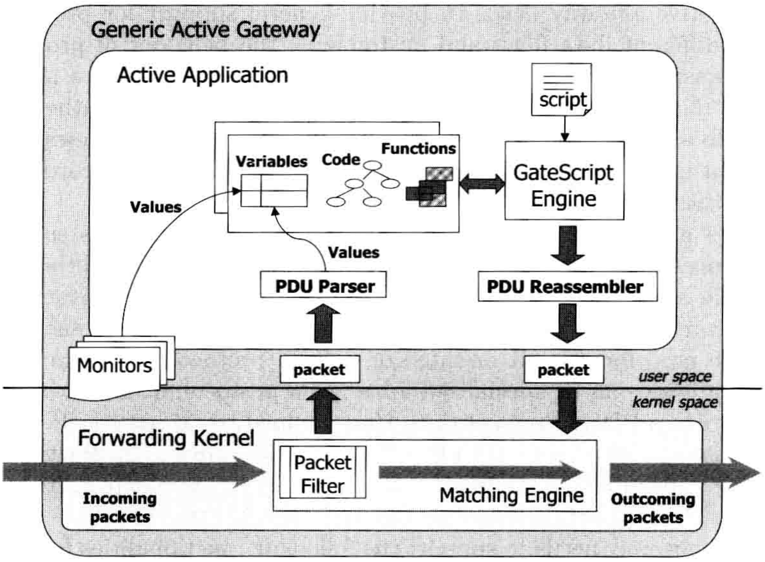

Fig. 1. Architecture of GAG

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.

Table 1. 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