Fast Intercept of a Passing Stream for High Performance Filter Appliances

Javed I. Khan & Yihua He

Media Communications and Networking Research Laboratory
Department of Computer Science,
Kent State University, 233 MSB, Kent, OH 44242
javed|yhe5@kent.edu

Abstract

Stream interception is becoming an important task in Internet appliances. In this paper we discuss a scheme that can facilitate random access into streams and provide serious performance boost to intercepting filter like appliances under various application scenarios. The scheme we discuss is based on dynamic stream indexing and can be potentially implemented using IPV6 extension header.

Keywords: *stream interception, filtering, network appliances, IPV6.*

1. Introduction

Stream interception is rapidly becoming a common task in Internet appliances. Beginning from cache, proxy, filters, firewalls and gateways, there are now host of new services including content adaptation, content personalization, location-aware data insertion, to security filters -- all are fundamentally stream interception machines requiring some form of intermediate access inside transiting traffic's content. A significant percent of the delivered Internet traffic is now 'touched'. However, if we look into the current protocol design, particularly the protocol packet structures we will see that it has little facility that enables efficient stream intercept.

Most real-life content, particularly high performance networked multimedia transport data carried over a network packet are multi-level hierarchically encapsulated (that means bags within suitcases). For example an H.261 video element may require access to 16 pre-dependent elements before the element of interest can be read. The situation is also difficult for recent tag-delimited content protocol standards (such as XML, HTML), where this dependency is formally infinite. For example, in annotation based web content transcoding [2], the transcoder has to make considerable effort in searching a certain value in the annotations (basically tags/metadata expressed by XML) at the application level before it can decide how the content should be transcoded.

The problem has major implication on the CPU cycle, memory size, and overall performance of any intercepting appliance system's architecture. The stream is the working data structure of these capsules. It is perhaps as salient to appliance's overall architecture as the design of disk scheduling algorithm or multilevel memory/cache organization is to the conventional machine architecture.

While, simple end-to-end applications can do away with marginal treatment of this issue, indeed, we believe right placement of protocol element inside data stream and some form of random access will be one of the most important factor for high performance stream data processing appliances.

In this research we focus on this new but important problem and demonstrate a novel content indexing scheme that can facilitate dynamic index based random access into streams and provide performance boost to intercepting filter like appliances. Though conceptually the mechanism can be implemented in layers above IP, we present an IPV6 [5] based protocol called *Embedded Data Indexing Protocol* (EDIP). It is an IPv6 extension header based content indexing mechanics, which defines the how a Content Provider (CP)'s serverlet can add special marks into the data stream, and how an intercepting Active Router (AR) can decode those marks from the data streams and gain pattern dependent random access into the elements of required data stream. An example service with EDIP header is show in Fig 1.

1.1 Few Related Work

Ma [6] suggested a content service network framework that extents the function of a traditional proxy to a content adaptation proxy. The Open Pluggable Edge Services (OPES) [9] working group built up a rule-based specification transmitting scenario [10][14] to express the service requests. A rule is made by description of a certain condition and a corresponding action, which will be executed by the service provider if the condition

meets. XML-like languages are often used in the description of rules. These languages include IRML[11] and OMML[15]. In that scenario, a service provider may download a set of "rules", and interpret and execute it by the rule processor. Here, rules are actually a kind of program languages with specific purpose. However, we noticed some limitations in such scenario. An obvious one is that, although the OPES model can be configured to "source-centric" or "client-centric"[10], there is not an easy way for the client to gain help from the source or vise versa. An alternative way to describe the "rules" is by a pair of tightly coupled program, distributed by a single authority to both the service source and the service client. Our ASDL[13] model is such an infrastructure which is considered "service centric", and the EDIP header can be used as a media carrying the helping information in this case.

The idea of putting information in IP level headers is not new, but little effort has been made in utilizing it in value-added services. S. Blake discussed about the differentiated services by adding marker field DS in IPv4 and IPv6 headers [1]. Packets marked by this field will receive a particular per-hop forwarding behavior on nodes along their path. It is a close approach as this paper's. However, they didn't investigate the possibility to add indexing information into IP headers and utilize it in value-added service to make random access of the data stream available. Spatscheck [7] and D. Maltz [8] have separately presented two TCP splicing mechanisms which would allow a filter (connected by two TCP links at two ends), to shed-off some TCP window maintenance functions, for passive filters by splicing the two TCP stream at two end-points.

In other extreme, Akamai [3] and ESI [4] use a full application level XML-like markup language to define web page components for dynamic assembly and delivery of web applications at the edge of Internet. IBM WebSphere[12] uses annotations (yet another application level XML-like markup language) to mark the content of the web and make it easier to be transcoded. The full application level maker searching and recognizing slow down the whole content process.

The technique we propose accelerates the actual filtering operation and applications, as much as it helps the networking layers. Also, the gain is not restricted for passive mode of operation. It uses network layer markup mechanisms to avoid decapsulation of non-essential application data (stream segments). Also, a key difference is that we include the case of cooperative application processing in the service model where server side help may also be available. One can think EDIP is another index mechanism at the IP level beneath the application level for faster marker recognizing. We could use it to accelerate the processing while not changing infrastructure already there.

2. In Route Application Service

First we explain the service model. In the service model a *content stream* from *content provider's server* (CP) flow to the *end-user* (EU). However it may also be processed in an ISP *application processing* (AP) server in between during transit. The end user initiates the content delivery by requesting content from the content provider via Internet. The Application Service Provider (ASP) modifies the content and adds value to the communication by application level intercept processing at strategically and/or topologically located AP servers. In special cases the CP and AP can be collocated in application service provider's AP.

A special case of AP intervention is the *passive filtering* service where AP server only monitors the stream without changing it. A further special case is the *stealth filters* where servers or end-users are not aware of the intercept service (and thus also not helping). If the content provider is also willing to help we call it *co-operative filtering* (for non-co-operative filtering some extra fast string matching operations are needed at the AP server).

The AP server additionally can provide "content cache". The cache can connect at either 'pre' or 'post' AP stage. Conceptually, caching is just another piped service that AP can provide. AP server can be configured to provide multiple services piped on a specific request/response stream-- caching can be one of them. The *piping sequence* is soft configurable. Complex application service can be composed from simpler services by *service piping*. The connection between EU, CP, and AP servers are provided by point-to-point separate TCP/IP or UDP connections.

3. EDIP Indexing Mechanism

The operation of application processing is expedited by two techniques. The first is pre-marking the content stream and allowing fast access into to the stream. Second is the selective decapsulation reencapsulation of only the pertinent data segments. Finally, we also define a language to express and carry the marks between the parties involved. The actual content intercept processing is performed by a program called *filter capsule*, which

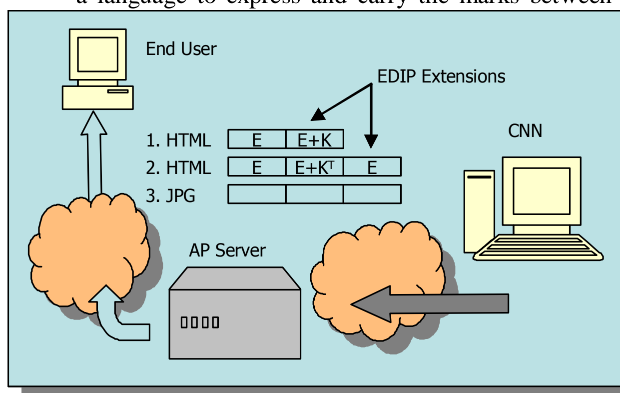

Fig 1: Example Service with EDIP Header

runs on AP server. The application service provider generally also sends a *marking servlets* to the CP server for marking of the content stream. Every Application Service Processing has a specified "*scope segment*" and a "*key segment*" in it. Generally a service is conditional. The data element that contains the condition or key is always intercepted and is decapsulated and delivered to

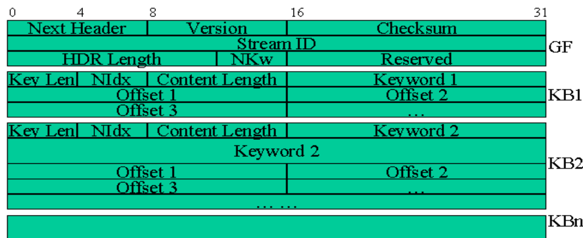

Chart 1: EDIP Header Format

the application capsule. The stream segment that is within the scope of an active key is intercepted and buffered. However, its decapsulation and delivery can

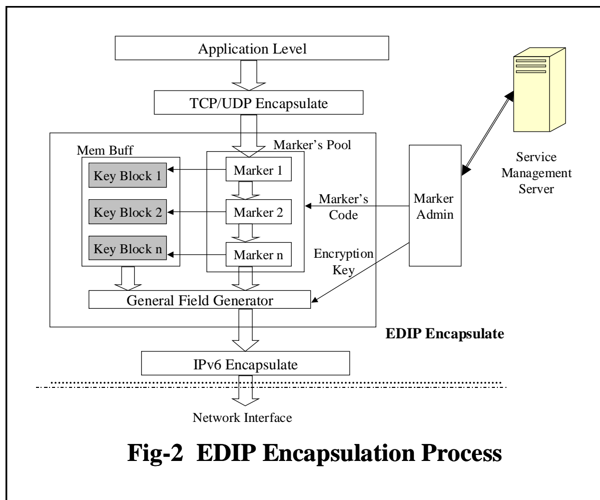

Fig-2 EDIP Encapsulation Process

be deferred based on the key evaluation result. If the evaluation is false, it is directly forwarded. Fig-1 shows the example service with EDIP header, and Fig-2 and Fig-3 are the schematics of the enhanced network layers that we have designed for the appliances machine.

3.1 EDIP Header Format

EDIP uses IPV6 extension header for content marking. It contains two parts: the General Field (GF) and Key Blocks (KB). The General Field (GF) identifies that it is an EDIP header, and contains general information in how to process the header. Each Key Block (KB) represents a keyword in this IP package, with positions of the keyword indexed by the offsets. Not every EDIP header has one or more KBs. Sometimes, an EDIP header may only have a GF, representing that the current IP package belongs to an indexed stream, while there is no key word appearance in this package. The total number of KBs that an EDIP can have is only limited to the maximum size of an IP package. Chart 1 shows the EDIP header format.

3.2 EDIP Encapsulation by Servelets

After capsulated by TCP/UDP, data stream can pass through multiple markers in the source's servelet pipe. Each marker program is associated with exactly one keyword and it examines the passing stream to see if there is any keyword appearance inside. If there are one or more appearances, the marker generates a KB containing the offset information about where the keyword is in the stream. Later, these KBs join the

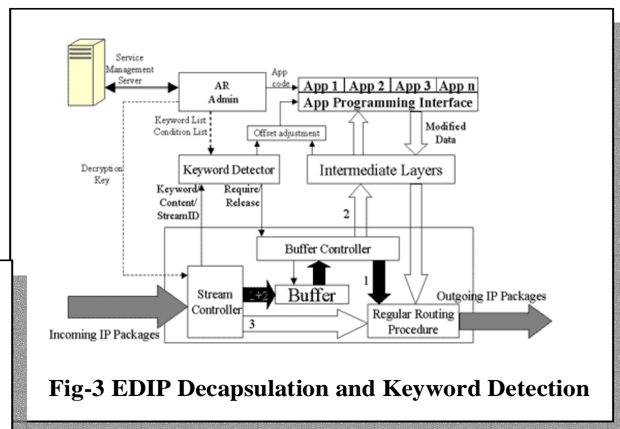

Fig-3 EDIP Decapsulation and Keyword Detection

original data stream in the GF generator, where a GF, as well as the KBs, will be added at the beginning of each package.The markers' codes are registered and distributed by SMS. Each CP's server running the markers will have a marker admin (MA) to maintain the markers. After MA receives markers deployment request from SMS and pass the authentications, MA will check if there is available resource (such as available slots in markers' pool, the size limit of a marker, etc) to deploy the marker. To enhance the security, MA may provide an encryption key to the GF generator, who may encipher the GF, and only authorized value-added service providers can decipher it.

3.3 Decapsulation and Services APIs

EDIP decapsulation and value-added services are executed in the ISPs intercept router, which sit on the edge of Internet backbone. There are several tasks that an intercepting router must do. (1) Differentiate the IP packages that need special processing from those normal IP packages. (2) Retrieve the offset information from the special-marked streams to the corresponding applications, which may use the information for value-added service. (3) Negotiate with SMS and maintain the service statistics. The main components include a stream controller, a keyword detector and a buffer controller. Fig-3 shows a possible architecture of a typical selective decapsulation system running on a router. Its main components and functions are described below:

Stream Controller: A stream controller's input is mixed IP packages, which may be IP packages with EDIP header, or just normal IP packages without EDIP header. A stream controller is supposed to forward those IP packages without EDIP in normal procedures, but store those with EDIP header into the Buffer for further actions. Further more, if the EDIP header contains any key blocks, the stream controller will decrypt it with corresponding decryption key from AR Admin, and send the keywords, contents and stream IDs to Keyword Detector.

Buffer Controller: The buffer controller maintains two lists --- a required_steamid_list and a release_streamid_list. Periodically, the buffer controller will check if there are any IP packages with the stream id listed in the two lists. Those in required list will be sent to application level and those in release list will be forwarded to their destinations. Every IP packages in the buffer has a timestamp. If timestamp expires, the IP package will be released.

Keyword Detector: The keyword detector is supposed to check if the keywords sent by stream controller are in the keyword list maintained by AR Admin. If not, the stream id will be added into release_streamid_list in the buffer controller. If yes, the stream id can be added to the required_streamid_list. Sometimes, Detector can do a little more. For example, each keyword entry can have a condition on the corresponding content. If a package's content matches the corresponding condition, its stream id will be added in the required_streamid_list in the buffer controller. If not, release it.

3.4 Application Processing:

The application is armed with a set of special services APIs to take advantage of the marking processing.

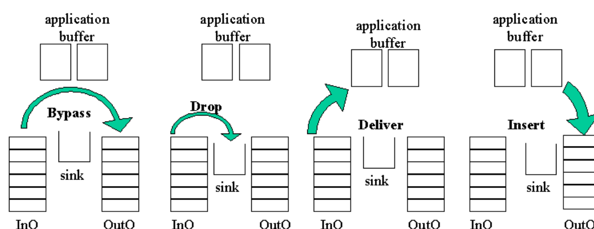

Fig 4: Application Processing Interfaces

These APIs can be viewed as two parts: (1) the administrative API subset, which is related to the start and stop of the service, and (2) the stream manipulate API subset, which is related to editing the coming stream. An example of these two subsets of APIs is illustrated below in table-1 and table-2. The application program can use the administrative API subset to activate and deactivate various marking and detection steps. It then can use the stream manipulation API set to edit, bypass, drop, or insert bytes with a sequence stream of incoming data.

Fig-4 explains the stream edit operations. Each of these operations is performed within the context of a paired incoming (InQ) and outgoing (OutQ) transport (TCP in this case) socket streams. The application creates sockets in usual way. The edit operation begins by associating an InQ with an OutQ. The bypass operation allows a specific number of application bytes to be moved from InQ to OutQ by the embedded layers (TCP, IP, and the router intercept in this case) directly. It can also request the enhanced TCP layer to drop specific segments of bytes. When the application needs the data it can request segments of stream to be delivered to the application buffer. It can also insert bytes into a stream from the application buffer.

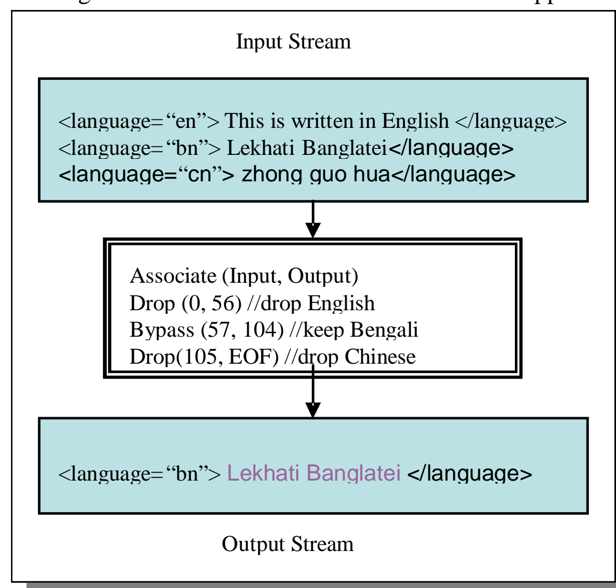

Fig5: Example of a content processing with stream-edit API

Fig-5 shows an example of a stream-edit capsule and its edit operations on a stream. In this example an incoming multilingual HTTP stream is converted into a single language document. The stream offsets are algebraically calculated from key indexes supplied by EDIP. Based on the indices, the filter uses drop and bypass calls to perform the edit operation.

API	Comment
ActivateMarker(IP, M_ID)	Start the marker (serverlet) at source side
DeactivateMarker(IP, M_ID)	Stop the marker (serverlet) at source side
ActivateEditor(IP, E_ID, labellist, range)	Start the editor at router's side
DeactivateEditor(IP, E_ID, range)	Stop the editor at router's side
ActivateTrap(E_ID, labellist)	Set the trapper in OS

Table –1 Administrative API Subset

API	Comment
Associate (inQ, outQ)	Associate two streams
GetOffset (label)	Get the offset of the label in the stream
Bypass(aid, a, b)	Forward bytes from a to b
Drop(aid, c, d)	Drop bytes c to d (into trash sink)
Deliver(aid, e,f, &*msgbuffer*)	Deliver bytes from e to f with *newcontent*
Insert (aid, *msgbuffer*)	Insert the messagebuffer content to the stream.

Table –2 Stream Edit API Subset

The data manipulate API subset can enable/disable the tracking of keys by activating/deactivating the marker/servelets and the intercept mechanism beneath. It can request for the next offset for a particular key. If the key test is successful (or unsuccessful), it can request (or release) delivery of the scope data. The AP capsules are also given a set of fast string search and protocol parsing routines (with potential hardware accelerators).

After the serverlet and the filter have been deployed, a common procedure generally takes place at the intercepting router's execution environment to conduct the service. Both the administrative API and the stream manipulate API are used in those procedures. A typical filter procedure is given below:

(1) Activate marker (in serverlet) at the source side. This step will activate the pattern detector, which will search some specific keywords or labels.

(2) Activate system trap in the intercept router's execution OS, telling the OS when some keyword in the labellist comes, wake the service up.

(3) Go to sleep

(4) When waken up by the OS, request to deliver the stream within the specified range to the application

(5) The application will use data manipulate APIs, such as getoffset(), bypass(), insert(), drop() and etc to modify the data stream if needed. An example of a content processing using stream edit API is show in Fig-5.

(6) Go to step (3) until the editor is deactivated.

The proposed mechanism accelerates the application level intercept process. The advantage is derived essentially by three principal sources -- (i) only the byte segments carrying 'keys' are unconditionally decapsulated; (ii) the byte segments carrying 'scope' are conditionally decapsulated only when the key conditions are true; (iii) rest of the bytes is never decapsulated.

This is also another source of run-time performance boost. Streams are marked by the servelet processes running at the content source. In cases, it is sometime possible to mark with direct content knowledge by the content generator without any string search. Otherwise, the marking can still be performed by sting search or parsing of the original content as preprocessing. It still therefore can drastically reduce the run time cost. To compare—current filters have to perform run-time full search and/or full parsing. The scheme however has cost. It is the extra data that will be needed by the EDIP markers. The actual saving therefore is the function of *key density*, and the *key success probability* in the stream. Though, apparently it may seem that high key density can offset the performance gain, but in practice always the EDIP key density can be controlled, by using a gross key in EDIP and then using application level processing to find the real keys. This is benefit of application level soft key definition ability. In practice, only a small part of data stream is generally modified. Consequently, the expensive part is way too inconsequential compared to the saving made by bypassing the costly decapsulation/

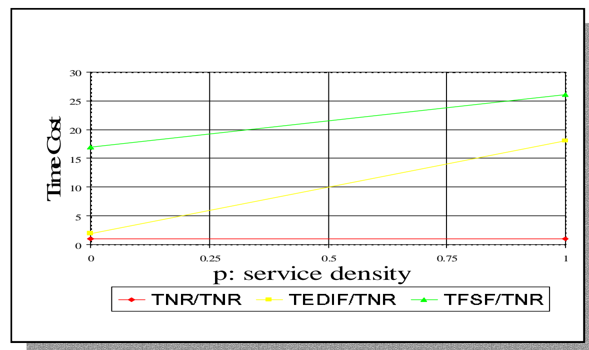

Fig-6 EDIP Speedup over Classical Filter

reencapsulation of the rest.

4. Performance Analysis

In the next section we provide the performance of the EDIF filtering. We compare it with simple Full Search Filtering (FSF) service. FSF represents the cost of performing the same service without the EDIF mechanism. For plotting we will normalize each of the costs with respect to the time it takes to process similar amount of data volume, without any service via a normal router (NR).

In the first experiment, plotted in Fig-6, we show the speedup. It plots the *relative speed cost* of both EDIF,

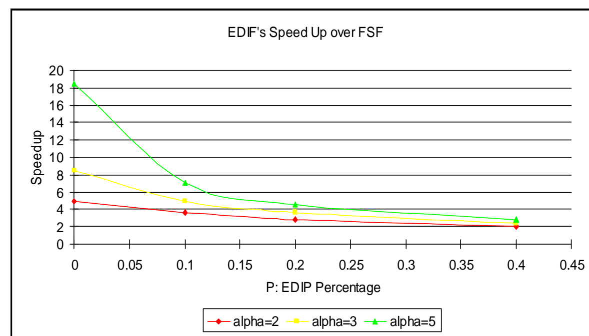

Fig-7 EDIP speedup over with interlayer speed differential

and FSF schemes. The cost of value-adding service is generally related to the amount of packets in the traffic those need the service. We call it *service density* (p). Fig-6 plots the relative speed cost for various service density. As can be seen, the EDIF incurred much smaller cost than FSF throughout. Particularly interesting is the points with a low (p). Here simple FSF incurred a cost about 16-17 times higher than that of a normal router. However, the EDIF performs almost as good as the normal router. This is because the marking mechanism allows EDIF to avoid decapsulations. In contrast the FSF has to decapsulated the entire stream whether there is a serviceable packet or not. Naturally, with the increased (p), the cost of service is increased in both the schemes. Notably, even at p=1, the EDIF mechanism could perform better. This is because the routine search can be performed at lower layer and upper layer will get index to the required data sections.

It is important to note that in the entire operation the computations are performed in three levels. These are (i) IP level (stream controller, routing, buffering, etc.), (ii) intermediate level (encapsulation, decapsulation, etc.) and (iii) application level (searching/indexed jump, filtering, etc.). The speed differential among these three levels in any filter architecture will significantly affect the overall performance of the scheme. The objective of our next experiment was to understand the potential impact of such architectural difference. In this experiment we assumed that each higher level is α times slower than its immediate lower level. Fig-7 now plots the EDIP scheme's relative speedup for three different speed differentials (α=2,3, and 5). As can be noted that advantage of EDIP increases with large alpha. The time saving is particularly dramatic if (p) remains small.

In next experiment we provide the space comparison for EDIP scheme. Here we assume that service density p=1 and keywords will appear at the frequency of *f* times/byte (assuming there is only one keyword per package, but it can appear multiple times in one packages). The result is shown in Fig 8. As evident the extra space cost is below 2% when the average package size is larger than 3k bytes, even if f=1/250, which is considered a very high frequency of keyword appearance. When average package size is over 10k bytes, the extra space cost by EDIP header is negligible.

5. Conclusions

Fast intercept of streamed data is a growing concern in networking. The application level embedded processing is rapidly increasing and can be a potential bottleneck in Internet traffic carriage. The network protocols and packet data structures have been designed mostly for end-to-end processing. A stream is a working data structure for network applications. However, little previous work exists which address this problem. In this paper we have presented a research, which looks into mechanisms that can provide random access in a stream. There are varieties of scenarios under which intermediate packet processing will be needed. In general the more the distributed components are cooperating the more benefits can be derived. However, particular scenarios may exclude specific co-operations. We believe a framework that can support multiple cooperative filtering scenarios will be valuable. To expedite the intercepting systems it may even be required to build special enhanced transport and network endpoints which are peer compatible, but has additional functionality for local end-points.

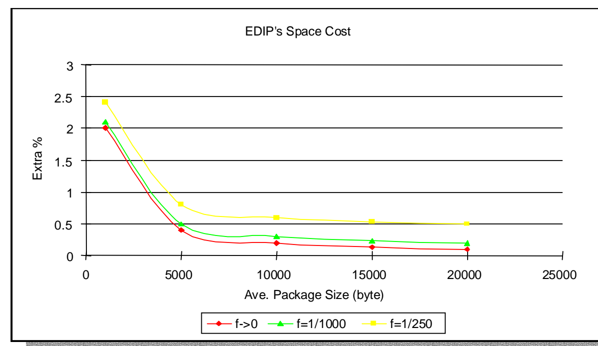

Fig-8 EDIP Space Overhead

The work is currently being funded by the DARPA Research Grant F30602-99-1-0515 under its Active Network initiative.

6. References

[1] S. Blake, D. Black and etc., RFC 2475 "A Architecture for Differentiated Services", 1998

[2] "Annotation-based web content transcoding", M. Hori, etc, The 9th WWW conference, 2000, available at: http://www9.org/w9cdrom/169/169.html

[3] Akamai, http://www.akamai.com

[4] ESI, http://www.esi.org

[5] S. Deering, R. Hinden, "Internet Protocol, version 6 specification", RFC 2460, 1998

[6] Wei-Ying Ma, Bo Shen and Jack Brassil, "Content Services Network: The Architecture and Protocols", Int. workshop on web caching and content distribution, June 2001.

[7] Oliver Spatscheck, J. S. Hansen, J. H. Hartman and L. Peterson; Optimizing TCP forwarder performance, IEEE/ACM Trans. Networking 8, 2, Apr. 2000, pp146 - 157.

[8] D. Maltz and E Bhagwat, "TCP splicing for application layer proxy performance," IBM, ftp://ftp.cs.cmu. edu/user/dmaltz/Doc/splice-perf-tr.ps, Mar. 1998.

[9] Open Pluggable Edge Service (OPES), http://www.ietf-opes.org

[10] "A Model for Open Pluggable Edge Services", G. Tomlinson, etc., draft-tomlinson-opes-model-00.txt

[11] "IRML: A Rule Specification Language for Intermediary Services", A. Beck, M. Hofmann, draft-beck-opes-irml-02.txt

[12] "Developing Web Applications for Pervasive Computing Devices", Steve Imes, IBM webserver studio document, Jan 2001

[13] "Ubiquitous Internet Application Services on Sharable Infrastructure", Javed I. Khan and Yihua He, technical report, available at: http://bristi.facnet.mcs.kent. edu/~javed/medianet/ techreports/TR2002-03-02-asp-KH.pdf

[14] "OPES Architecture for Rule Processing and Service Execution", Lily Yang, Marcus Hofmann, draft-yang-opes-rule-processing-service-execution-00.txt

[15] "OMML: OPES Meta-data Markup Language", Christian Maciocco, Marcus Hofmann, draft-maciocco-opes-omml-00.txt