Design of a SIP Outbound Edge Proxy (EPSIP)

Sergio Lembo

Dept. of Communications and Networking
Helsinki University of Technology (TKK)
P.O. Box 3000, FI-02015 TKK, Finland

Jani Heikkinen, Sasu Tarkoma

Dept. of Computer Science and Eng.
Helsinki University of Technology (TKK)
P.O. Box 5400, FI-02015 TKK, Finland

Abstract

This paper presents the design of a SIP Outbound Edge Proxy that follows the requirements stated in IETF Internet-Draft SIP Outbound. The paper presents a ready-to-implement design, thus bridging the gap between the requirements in the draft and an actual and feasible implementation of these requirements in a real system.

I. INTRODUCTION

SIP (Session Initiation Protocol) is a signaling protocol designed to relate users (User Agents) on an IP network and allow them to establish multimedia communication sessions such as voice and video. Without special considerations SIP User Agents behind a Firewall or Network Address Translator (NAT) are unable to receive incoming SIP requests due to the presence of these network elements. Among several proposals to solve this issue we focus here on a particular solution provided by an IETF draft: *Internet-Draft SIP Outbound* [1]. SIP Outbound enables incoming SIP requests to a User Agent behind a Firewall or Network Address Translator (NAT) with a mechanism that requires a particular kind of SIP proxy, namely, SIP Edge Proxy. This document presents a design of such SIP Edge Proxy conforming to SIP Outbound draft. The design has been named EPSIP (Edge Proxy SIP).

The structure of this paper is as follows. Section II presents a brief introduction to SIP Outbound. Section III presents the scope of the design of a SIP Edge Proxy. Section IV describes the guidelines followed during the design of the Edge Proxy. Section V presents details about the architecture at transport level of the proxy. Section VI discusses flows, nonflows and transport protocols. Section VII gives more details about transport and application layers design. Finally conclusions are presented on Section VIII.

II. BRIEF OVERVIEW OF SIP OUTBOUND DRAFT AND FUNCTIONALITY OF A SIP OUTBOUND EDGE PROXY

SIP Outbound address scenarios where SIP Registrars/Proxies can not form a connection toward an UA (User Agent) due to, for example, the presence of a NAT or Firewall between the UA and a Registrar/Proxy. In this section SIP Outbound is introduced through a particular example for the scenario depicted in Figure 1 and explained from a point of view centered at the Edge Proxy functionality. The scenario shown in Figure 1 consists of a UAC (User Agent Client), SIP Outbound Edge Proxy (EPSIP) and a SIP Registrar/Authoritative-Proxy.

In Figure 1 UAC is a SIP UAC that implements SIP Outbound as presented in SIP Outbound draft [1]. Edge Proxy, is a proxy defined in the draft as *" any proxy that is located topologically between the registering User Agent and the Authoritative Proxy"*. An Authoritative Proxy is a proxy defined as *"a proxy that handles non-REGISTER requests for*

a specific Addressof-Record (AOR), performs the logical Location Server lookup described in RFC 3261, and forwards those requests to specific Contact URIs".

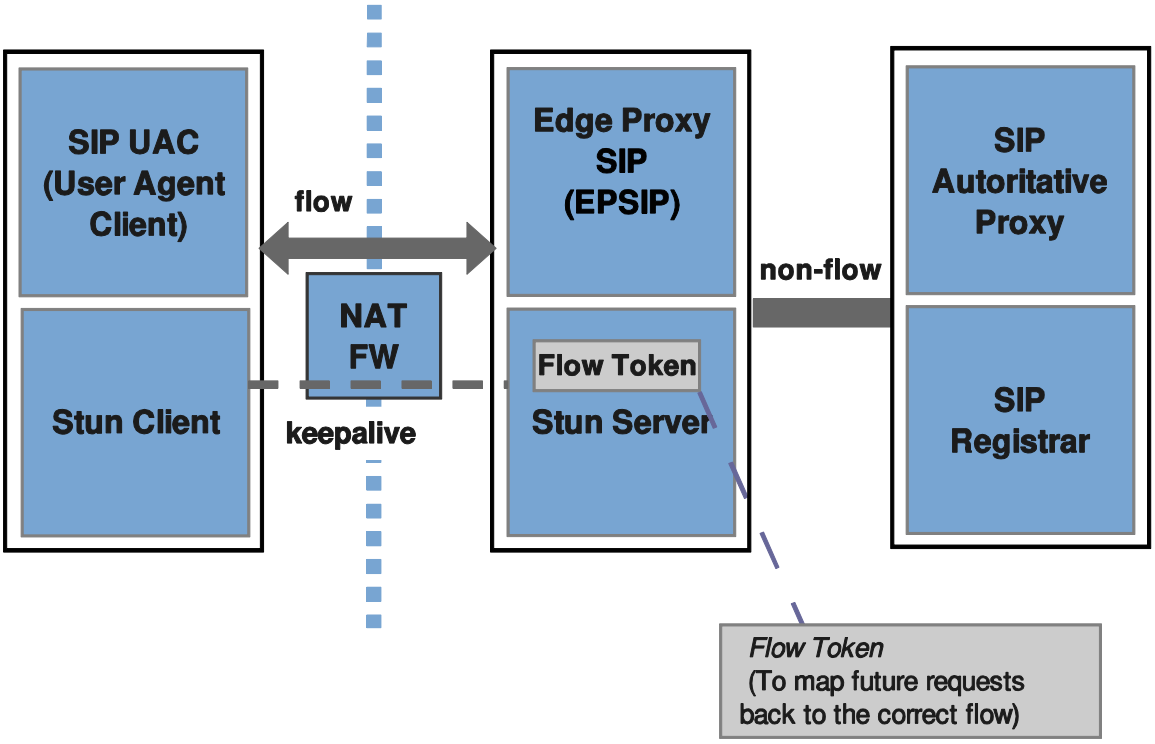

Fig. 1. Example scenario: UAC, Edge Proxy, Authoritative Proxy and Registrar.

SIP Outbound draft proposes to form network "flows" between the UAC and an Edge Proxy when an UAC registers sending SIP REGISTER requests. Where "flow" is considered as a TCP connection in the case of TCP transport protocol, or as a bidirectional stream of datagrams in case of UDP transport protocol. The main idea is that when the Edge Proxy intends to forward a request to the UAC, the request is sent to the UAC by using a previously established flow. More than one flow can be established by the UAC to the Edge Proxy to avoid possible intervals of time where the UAC could not be reached if a flow fails. Multiple flows are established by sending multiple SIP REGISTER requests to the Registrar with different registration-id (reg-id) identifiers.

The Edge Proxy must be ready to forward a request toward the UAC at any point in time; therefore the flows must be "alive" all the time. A flow is kept alive by means of a keepalive mechanism that basically consists of the UAC sending a keepalive message over the flow (i.e. over the same flow used for SIP messages) toward the Edge Proxy. Proposed keepalive mechanisms in the case of TCP transport are *TCP keepalive* and *CRLF keepalive*. The keepalive mechanism proposed in the case UDP transport is *STUN-keepalive* and it is implemented by using a limited version of a STUN server working on the same IP address and port number used for SIP [2].

When a Edge Proxy supporting SIP Outbound receives a registration request from the UAC, and when it is the first SIP node after the UAC, the Edge Proxy adds a flow identifier "flow-token" in the Path URI and an 'ob' parameter into the Path header field value; and then it forwards the registration to the Registrar. The flow-token identifier is unique in the network

and it is generated in the Edge Proxy by any suitable algorithm. SIP Outbound draft [1] provides an example algorithm that maps local and remote IP addresses and port numbers, and the transport protocol used by the flow, into a HMAC using a 20-octet crypto random key and MAC-SHA1-80 algorithm [3].

The flow-token added to the REGISTER request by the Edge Proxy will be stored in the Registrar and included in future SIP requests toward the UAC, forwarded by the Edge Proxy. When the Edge Proxy receives a request it applies normal routing procedures; except when the Edge Proxy is the host in the topmost Route header and the Route header contains a flow-token. In this case the flow-token is compared with the source of the request, and it is determined if the request is "incoming" (toward the UAC in our example) or "outgoing" (from the UAC in our example). In the "outgoing" case the Edge Proxy continues processing the request as a 'traditional' SIP proxy (a SIP proxy as defined in RFC 3261 [4]). In the "incoming" case the Edge Proxy forwards the request to the UAC over the flow stated by the flow-token.

III. SCOPE OF THE DESIGN

The design of EPSIP (Edge Proxy SIP) comprises mainly the design of a working model of SIP Outbound for UNIX systems, suitable to handle several transport protocols and be relatively more than a proof of concept; in fact it was designed to serve as a basic prototype for real production scenarios.

EPSIP implementation focuses to fulfill SIP Outbound draft [1]. In the design, the behavior as SIP proxy is not considered in the sense as a proxy is described in RFC 3261 [4], but limited to the necessary and sufficient functionality required by an Edge Proxy as described in SIP Outbound draft.

Our main contribution in the design of EPSIP comprises the idea to handle flows by processes, mapping flow-tokens to processes through local sockets, and logic related to the behavior required by an Edge Proxy as defined in the SIP Outbound draft. Design and logic are explained in the following sections.

IV. GUIDELINES TOWARD THE DESIGN OF A SIP OUTBOUND EDGE PROXY (EPSIP)

EPSIP (Edge Proxy SIP) was designed according to the following guidelines.

a) The proxy should be based on a stateless SIP proxy.

b) The architecture must be capable to maintain TCP connections in time.

c) TCP connections (flows) must be managed with a concurrent approach.

d) Messages to be forwarded over a TCP connection (flow) must reach the corresponding flow by a suitable mechanism that maps the content in the flow-token to the module in charge of the TCP connection.

These guidelines are explained below.

a) *The proxy should be based on a stateless SIP proxy.* A SIP proxy can be implemented either as a stateful or stateless proxy. We decided to adopt a stateless SIP proxy by considering the following issues. First, SIP Outbound suggests a stateless approach to map incoming SIP requests to a flow by using a flow-token; therefore we considered hat keeping SIP transactions may be not necessary. A SIP transaction could be useful for example in a case when it is used to associate a mapping to the flow, but since the draft does not encourage the use of storage, there

is not particular reason to choose a stateful approach. Second, a stateless proxy is easy to implement and thus may reduce significantly the implementation time when considering elaborated SIP stacks implementations.

b) *The architecture must be capable to maintain TCP connections in time.* A TCP connection represents a flow that is created once with the arrival of a SIP REGISTER request and from that point must remain in time. When we point out that the connection must remain in time we want to state that this connection will be always used for the established flow. The connection will expire after certain time if it is not refreshed by a keepalive mechanism. To keep a connection in time means in fact that the proxy is keeping a state, nevertheless this state is at transport level and not at application (SIP) level.

c) *TCP connections (flows) must be managed with a concurrent approach.* Our intention is to provide a design that can be used as a basic model for real production scenarios and not be merely a simple proof of concept. Then, for example, the management of TCP connections (that remain in time) can not be effectively implemented with an iterative method. The design must be able to handle the connections in a concurrent way and avoid serial processing that can produce delays and locking entirely the proxy. In order to handle TCP connections in a concurrent way we propose to associate a connection to a process or thread that will be responsible to handle the particular connection. The main idea is to have a main process or thread that will originate child-processes or sub-threads, and then every one of these processes or threads will be responsible to handle a particular TCP connection. The processes or threads will work concurrently and handle arriving messages over the flows that these have in charge. In this sense we introduce the concept that the proxy will have one process or thread running per established flow.

d) Messages must reach the corresponding flow by a mechanism that maps the content of the flow-token to a connection. As it was proposed to have a concurrent implementation with one process or thread per TCP flow, these processes or threads must be reached appropriately when it comes the time to forward an arriving message over a flow. In other words, the content of the flow-token must be mapped to some reference that will make the message reach the appropriate process or thread in charge of the flow.

V. TRANSPORT LAYER ARCHITECTURE DESIGN OF A SIP OUTBOUND EDGE PROXY (EPSIP)

Our design lies in the basic guidelines mentioned in the previous section. As it was mentioned, one of our goals is to use concurrency in our design. Possible ways to implement concurrency are using either multiple processes or multiple threads.

The design presented here is based on implementing concurrency in the proxy by using multiple processes. Even when an approach using multiple processes is less efficient (in execution time and system resources) than an approach using multiple threads, the multiple processes approach was chosen for several reasons. These reasons are, first, better personal and general knowledge with the multiple-process approaches; second, the easiness to debug code between multiple processes during an implementation phase; third, the easiness to monitor the behavior of the processes during a development stage; and fourth, due that some SIP stacks are thread-safe, thus requiring a suitable redesign like using additional locking and signaling mechanisms.

The design at transport level using multiple-processes is based on the traditional approach of using a main process and forking child processes [5][6]. The main process will create child-processes every time a message arrives. Each process will be in charge to handle an incoming message and terminate or remain in time according with the transport protocol used by the flow and type of SIP message.

Figure 2 shows the proxy architecture at process level. In this figure we differentiate three kinds of processes:

1. Main process
2. Child processes to handle connections for TCP transport protocol.
3. Child processes to handle datagrams for UDP transport protocol.

These processes are explained in the sub-sections below.

A. Main process (1)

The main process (1) will be in charge of listening in certain port (e.g. 5060) for incoming messages over connectionless protocols (UDP) as well as connection-oriented transport protocols without an existent connection (TCP). The main idea is that every time a message arrives the main process will create a child process, and subsequently the child process will handle the incoming message. In this way the main process will return to listening state for new incoming messages and the child process will handle independently all the processing load. With this approach we introduce concurrency in our design.

Types of child processes are divided into two categories. The child process created by the main process (1) will depend on the transport protocol from where the message arrived. If the message arrived over TCP transport protocol the main process will create a process type (2). If the message arrived over UDP transport protocol the main process will create a process type (3). These types of processes are described below.

B. Child processes for TCP (2)

Messages arriving over a connection-oriented protocol without a previous connection will be handled by the main process (1). When such message arrives, the main process will create a child process that will be in charge to keep the connection. Thus the main process continues listening for new events, while the new child process handles the message and remains in time holding the established connection and waiting for messages over the connection (flow). The connection represents an outbound-flow for TCP transport protocol. In this way a flow is associated to a dedicated process that is in charge to handle exclusively messages arriving over this flow.

Incoming messages over a TCP flow will be handled directly by a process type (2). An incoming message over the flow can be either a SIP message or a *CRLF keepalive*. If the message is a SIP message, it is updated appropriately at application level and then forwarded over a non-flow. If the message is a keepalive we refresh a countdown timer inside the process and answer the *CRLF keepalive*. If the countdown timer is not refreshed in a suitable time by a keepalive the process will terminate, thus removing the SIP Outbound flow.

We define ’non-flows’ to normal SIP connections or bidirectional streams of datagrams that are not related to SIP Outbound. Messages arriving over a non-flow by a new formed TCP connection will also have a dedicated process. This process will forward the SIP message and after that automatically terminate since it will not receive any keepalives. In this

sense the proxy design is not limited to SIP Outbound but it can be used as a general SIP proxy.

On the system several processes type (1), (2) and (3) may coexist and need to communicate each other in order to forward messages arrived at one process and required to be delivered by other process that has the correct flow. For example, a message can arrive to the main process (1) or to a process type (2) and need to be forwarded over a flow kept by other process type (2). Then processes of type (2) will need to have a suitable inter-process communication with other processes working in the proxy. Inter-process communication can be either shared memory or local sockets for example. We for example implemented successfully the proposed design using inter-process communication with local sockets.

Messages can arrive to one process and then required to be delivered to a different one because some decisions will arise according to the content of the flow-token in a SIP message. Some examples of decisions are like if the message is a SIP request and if it must be forwarded over a flow (incoming case) or over a non-flow (outgoing case).

In Figure 2 we show that a process type (2) can have four kinds of inputs and associated outputs.

1. SIP messages arriving over a flow are delivered over a non-flow.
2. *CRLF keepalives* are answered over the flow.
3. A message arriving over a local socket is forwarded over the flow associated to this process.
4. A message over non-flow by a new formed TCP connection is forwarded over a local socket (TCP case) to the corresponding process handling the flow, or directly (UDP case) to destination.

In the design it is proposed to keep the file-descriptor of a connection within the child process. This approach provides suitable associations of file-descriptors (connections) to flowtokens and helps to manage several open file-descriptors in the proxy.

C. Child processes for UDP (3)

Messages arriving over a connectionless protocol will be received by the main process (1). When such message arrives the main process will create a child process that will be in charge to handle the message. Thus the main process continues listening for new events, while the new child handles the message and then terminates.

Arriving messages can be either over a non-flow or over a flow. The destination to where the message must be forwarded is determined by the flow-token information. The destination can be over a UDP or TCP flow. If the destination is to a UDP flow, the message can be forwarded directly using the file-descriptor for UDP and the destination available in the flowtoken (IP address and port). If the destination is to a TCP flow the content in the flow token must be mapped to the appropriate flow. As mentioned above, the flow-token can be mapped to a flow by some inter-process communication to the process keeping the flow; for example, we used local sockets for this purpose. Processes type (2) listen in a local socket with a name equivalent to the information stored in the flow token. In this way an incoming message directed to a flow is delivered to the corresponding process via a local socket by using the information available in the flow token. Once the message arrives to the process type (2) it is forwarded over the flow.

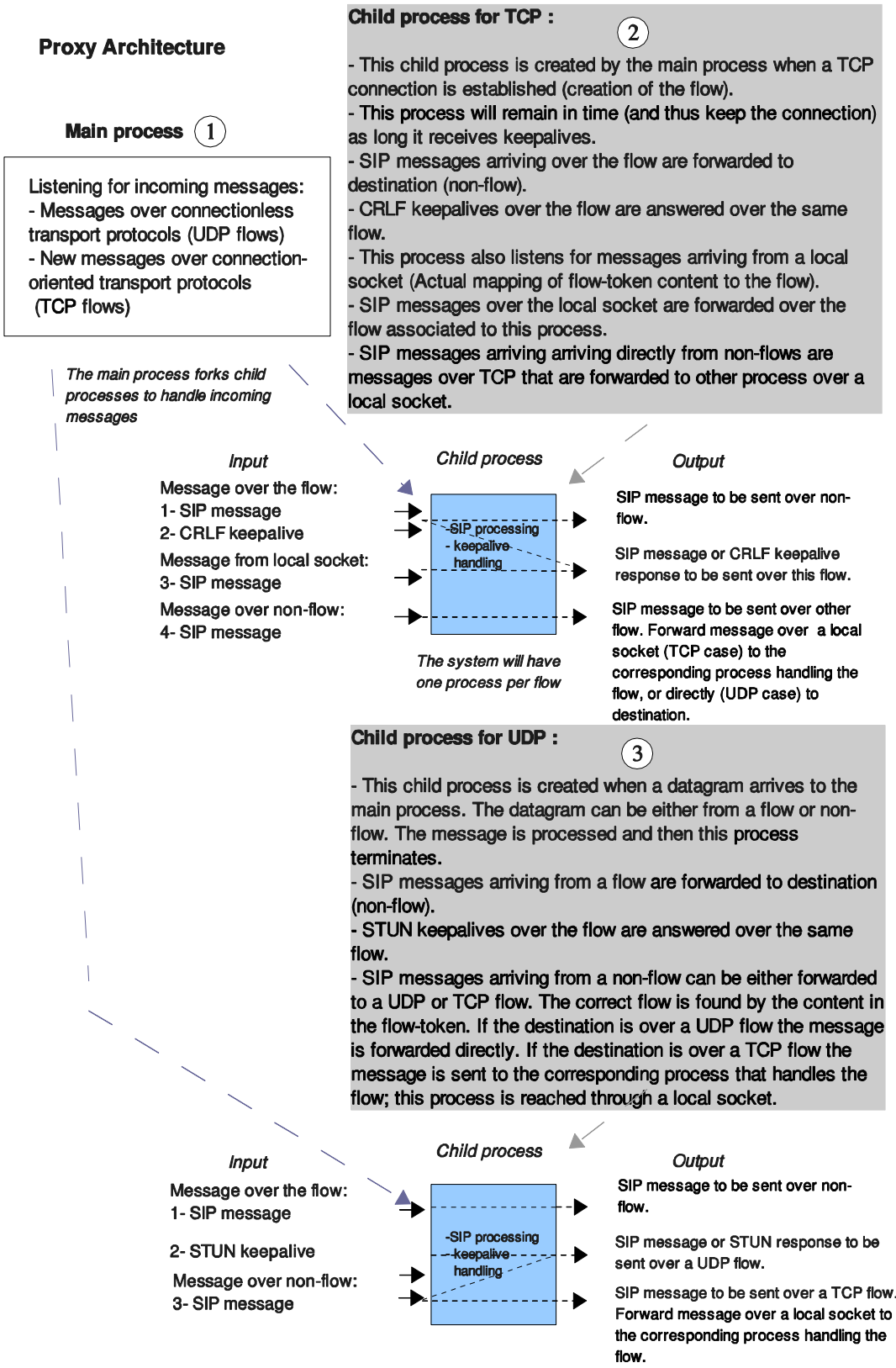

Fig. 2. Proxy architecture at transport layer level

In Figure 2 we show that a process type (3) can have three kinds of inputs and associated outputs.

1. SIP messages arriving over the flow are delivered over a non-flow.

2. *STUN keepalives* are answered over the flow.
3. A message over non-flow is processed and the suitable destination is determined. In case of destination over a TCP flow the SIP message is delivered to a process, associated to a flow, through a local socket. In case of destination over a UDP flow the SIP message is delivered directly over the UDP flow.

VI. FLOWS, NON FLOWS AND TRANSPORT PROTOCOLS

In this section we distinguish flows and non-flows in the edge proxy. Figure 3 shows the Edge Proxy along with flows and non-flows. EPSIP proxy was designed to work with flows over the connection-oriented protocol TCP and over the connectionless protocol UDP.

We define 'non-flows' to normal SIP connections or bidirectional streams of datagrams that are not related to SIP Outbound.

In the proposed design traffic over non-flows is considered to be only over UDP . Messages arriving from TCP/UDP flows are forwarded over a non-flow using UDP protocol by default. TCP protocol support to forward messages over non-flows is not considered in the design. The main reason to use UDP for non-flows is just a decision based on two facts. First, SIP is widely used with UDP. UDP is mainly used because it is difficult to build SIP proxies that can maintain a very large number of active TCP connections, [1] (section 14). Second, there is not any defined criterion in SIP to switch from one protocol to other, except the one mentioned below. For example, sending a SIP message over a flow and then forwarding it using other protocol over a non-flow is more a configuration issue that a general imposed rule. The only exception that imposes a definite criterion to switch from one protocol to other is found in RFC 3261 [4]. RFC 3261 states that when a SIP message is too large for UDP (exceeding the maximum transmission unit, MTU) a UA sending a SIP message may need to use TCP protocol. EPSIP does not support currently this criterion. EPSIP supports at some extent non-flows using TCP but this feature was not extensively considered.

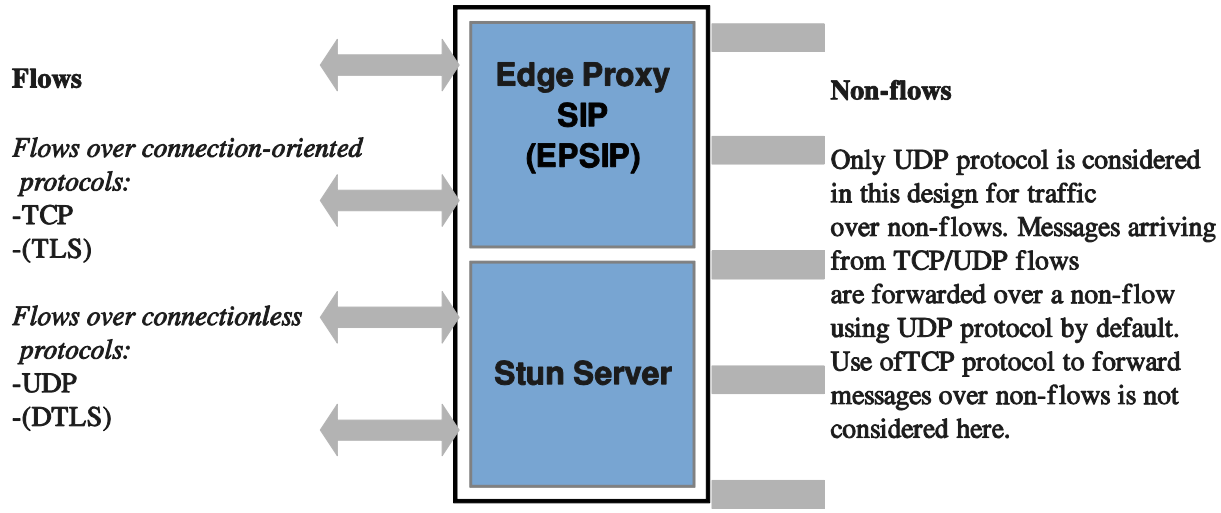

Fig. 3. Flows and non-flows supported by the Edge Proxy.

VII. EDGE PROXY SIP, DETAILS OF TRANSPORT AND APPLICATION LAYERS

Figure 4 shows the details of architecture and design of the Edge Proxy SIP (EPSIP). The figure is divided into two main areas, Transport Layer and Application Layer.

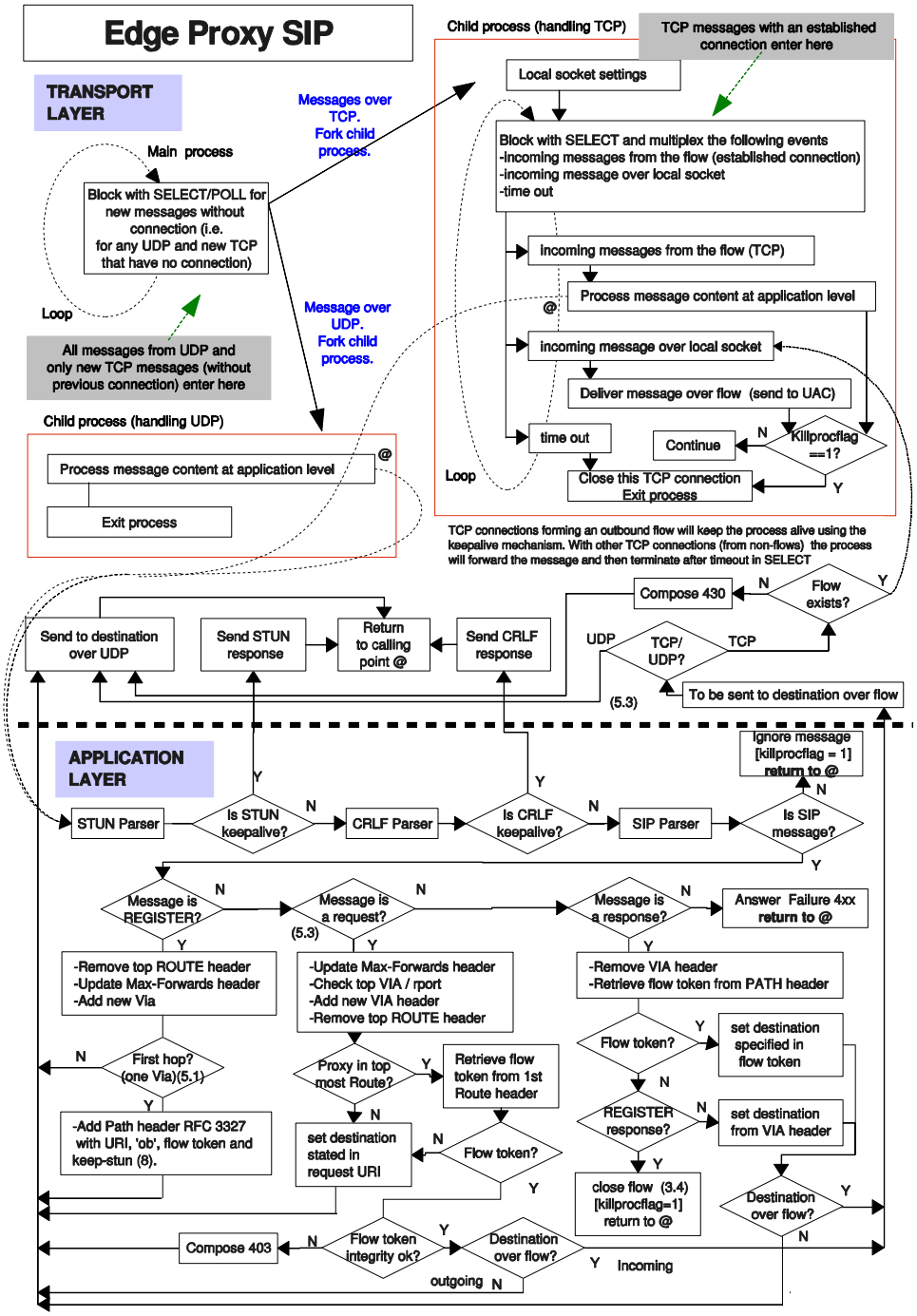

Fig. 4. EPSIP, Details of implementation

Transport Layer is based on the diagram shown in Figure 2. On Figure 4 the transport layer shows a Main Process, one Child Process for TCP and one Child Process for UDP. Actually two child processes are shown but the design follows the 'one child process per flow' concept explained in Section V. Rectangular boxes in the flow chart indicate actions and rhombuses indicate decisions.

The application layer shows the details behind the processing of a new incoming message. First the message is identified as keepalive or SIP message and then the corresponding actions shown on the figure are taken. The logic related to SIP Outbound handling is mainly divided

in three cases, namely, SIP REGISTER messages, other SIP request messages and SIP responses. SIP processing shown on the figure aims basically to focus on SIP Outbound message handling; details of SIP Outbound message processing are detailed on SIP Outbound draft [1]. Numbers between parentheses indicate the section in the draft related to the actions taken or decisions evaluated. The logic shown on the figure can be understood in detail by following the explanations in SIP Outbound draft.

Additional information related to the software implementation of EPSIP and testing scenarios are reported in [7] and [8]. Furthermore [6] presents additional information considered during the development of EPSIP.

VIII. CONCLUSION

We designed a SIP Outbound Edge Proxy and introduced the concept of using a dedicated process per flow. Also we proposed a method to map a flow-token to a TCP connection by relating a file-descriptor to a process and a process to a local socket with a name equivalent to the information stored in the flow token. At the same time the design implements a multiple-process concurrent approach to handle independently multiple flows. Furthermore we composed a flow chart that resumes in one figure (Fig.4) the design of the proxy and SIP Outbound logic [1].

The proxy is suitable not only to be used as a SIP-Outbound Edge Proxy, but also as a multi-transport-protocol SIP proxy with the advantage of offering a design that contemplates the use of transport oriented protocols.

The design was implemented in a real proxy (EPSIP) and we successfully verified its behavior in a test scenario.

REFERENCES

[1] Jennings, C., Mahy, R., “Managing Client Initiated Connections in the Session Initiation Protocol (SIP),” *Internet-Draft draft-ietf-sip-outbound-10*, IETF, Jul 2007.

[2] Rosenberg, J., “Simple Traversal Underneath Network Address Translators (NAT) (STUN)”, *draft-ietf-behaverfc3489bis-05*, October 2006.

[3] Krawczyk, H., Bellare, M., and R. Canetti, “HMAC: Keyed-Hashing for Message Authentication”, *RFC 2104*, February 1997.

[4] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A.,Peterson, J., Sparks, R., Handley, M., and E. Schooler, “SIP: Session Initiation Protocol”, *RFC 3261*, June 2002.

[5] Stevens, W., R., “UNIX Network Programming - Networking APIs: Sockets and XTI - Volume 1”, *Prentice Hall*, 1997.

[6] Lembo, S., Heikkinen, J.,“Enabling Session Initiation in the Presence of Middleboxes”, *The 3rd Student Workshop at the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Columbia University, NY, USA, December 10.-13., 2007.

[7] Lembo, S., “SIP Outbound Implementation Status Report - June 2007 - Rev.01”, *WeSAHMI project*, Helsinki University of Technology, June 2007.

[8] Lembo, S., “Implementing a SIP Outbound Edge Proxy”, *T-110. 6100 Special Assignment in Datacommunications*, Helsinki University of Technology, November 2007.