A Programmable Message Classification Engine for Session Initiation Protocol (SIP)

Arup Acharya, Xiping Wang, and Charles Wright

IBM T.J. Watson Research Center
19 Skyline Drive, Hawthorne, NY, 10532
{arup,xiping,cpwright}@us.ibm.com

ABSTRACT

Session Initiation Protocol (SIP) has begun to be widely deployed for multiple services such as VoIP, Instant Messaging and Presence. Each of these services uses different SIP messages, and depending on the value of a service, e.g. revenue, the associated messages may need to be prioritized accordingly. Even within the same service, different messages may be assigned different priorities. In this paper, we present the design and implementation of a *programmable* classification engine for SIP messages in the Linux kernel. This design uses a novel algorithm that in addition to classifying messages can extract and maintain state information across multiple messages. We apply the classifier for overload control using operator-specified rules for categorizing messages and associated actions, augmented with a protocol-level understanding of SIP message structure. When faced with loads beyond their capacity (e.g., during catastrophic situations and major network outages), SIP servers must drop messages. It is therefore desirable that the server process high-value messages in preference to lower-value messages. We evaluated our in-kernel classifier implementation with an open source SIP server (SER) for such an overload scenario. The workload consists of a mix of call setup and call handoff messages, and the classifier is programmed with rules that prioritize handoffs over call setups. We show that, while SER can process about 40K messages/sec (in a FIFO manner), our classifier can examine and prioritize 105K messages/sec during overload. With the classifier operating at peak throughput, SER's processing rate drops to 31.6K messages/sec, but all of the available high-value messages are processed.

Categories and Subject Descriptors

C.2.2 [**Computer Communication Networks**]: Network Protocols – *SIP*. C.2.3 [**Computer Communication Networks**]: Network Operations. D.4.4 [**Operating Systems**]: Communications Management – *Buffering.*

General Terms

Performance, Design, Experimentation, Reliability

Keywords

SIP, Overload Control, Programmable Classification.

1. INTRODUCTION

Session Initiation Protocol (SIP) is a control plane for establishing, manipulating, and terminating multimedia sessions with one or more participants. SIP is media agnostic and can establish voice, text, video, and other types of sessions. SIP has already gained widespread acceptance and deployment among wire line service providers for introducing new services such as VoIP, within the enterprise for Instant Messaging and collaboration, and for push-to-talk service amongst mobile carriers. Service providers ranging from cable companies to mobile providers are looking to deploy the SIP-based IP Multimedia Subsystem (IMS) as a common platform for deploying new services and applications [2].

In this study, we examine the problem of how to classify SIP messages before they are processed by a SIP server. Today, SIP servers process messages in a first-in-first-out manner. This does not lend itself to prioritizing messages before they are processed at the server. We design an efficient algorithm that takes as input a set of user-defined rules, and morphs them into suitable data structures that enable fast matching of rules against the input message stream. The rules specify both how to identify specific subsets of messages, based on a combination of message header values including complex functions such as set membership as well operations on the state amassed at the classifier from previous messages, and the actions to be executed on the matching packets. Since the classifier is driven by these user-specified rules, it can be programmed to suit specific goals ranging from overload control to denial-of-service prevention for SIP servers.

We showcase overload control as a defining example for our classifier in this paper. Given the variety of usage contexts for a SIP server (e.g. Voice-over-IP, Instant Messaging, Presence, etc.), it is not surprising that each service provides a different value to the operator (e.g., revenue or customer satisfaction). Moreover, different types of messages within a service can also provide different amounts of value. Thus, our classifier-based solution for overload control will aim to maximize the value of the messages processed by the server. SIP servers can become overloaded despite being provisioned correctly. During overload, only some requests can be handled and the rest are dropped to decrease the server load and bring the load down to maximum server capacity. Rather than dropping requests randomly or in a FIFO fashion, our goal is to prioritize requests in order to maximize value for an operator. Additionally, each server operator may have a different notion of value attached to a specific type of request. We demonstrate how our classifier prioritizes messages according to operator-specified metrics, so that under overload conditions, revenue is maximized by servicing the higher-value requests first.

2. SIP BACKGROUND

A SIP infrastructure consists of *user agents* and a number of *SIP servers*, such as registration servers, location servers and SIP proxies deployed across a network. A *user agent* is a SIP endpoint that controls session setup and media transfer. RFC 3261 [14] describes the SIP protocol in detail.

All SIP messages are *requests* or *responses*. For example, INVITE is a request, whereas "180 Ringing" and "200 OK" are responses. A SIP message consists of a set of headers and values, all specified as strings, with a syntax similar to HTTP but much richer in variety, usage and semantics. For example, a header may occur multiple times, have list of strings as its value, and a number of sub-headers, called parameters. In the following example from [14] Alice invites Bob to begin a dialog:

```
INVITE sip:bob@biloxi.com SIP/2.0
Via: SIP/2.0/UDP pc.atl.com;branch=z9h
Max-Forwards: 70
To: Bob <sip:bob@biloxi.com>
From: Alice <sip:alice@atl.com>;tag=192
Call-ID: a84b4c76e66710@pc.atl.com
CSeq: 314159 INVITE
Contact: <sip:alice@pc.atl.com>
Content-Type: application/sdp
Content-Length: 142
(Alice's SDP not shown)
```

SIP messages are routed through SIP proxies to setup sessions between user agents. All requests (such as an INVITE) are routed by the proxy to the appropriate destination user agent based on the destination SIP URI included in the message. A session is setup between two user agents through an INVITE request, an OK response and an ACK to the response. Call setup is followed by media exchange using RTP. The session is torn down through an exchange of BYE and OK messages.

SIP separates signaling from the media – signaling messages are carried via SIP, whereas media is typically carried as RTP over UDP [17]. The body of a session setup message (e.g., INVITE) contains Session Description Protocol (SDP) [8] encoded parameters that the clients use to transmit media in an end-to-end fashion. Any of these parameters (e.g., IP address) can be changed during an ongoing session via a re-INVITE message, which is identical to an INVITE except that it occurs within an existing session. The re-INVITE message is used most often in mobile networks to support call handoff due to user mobility (and subsequent change of endpoint addresses).

SIP can operate over multiple transport protocols such as UDP, TCP, or SCTP. UDP is most prevalent today, but TCP usage is expected to grow down the road. Additionally, when using TCP, SIP can use SSL (secure sockets layer) for security and encryption. It may also use IPSec underneath any of the transport protocols as well.

3. MOTIVATION

Overload is an inevitable condition for servers. Flash crowds, emergencies, and denial-of-service attacks can all initiate loads that exceed a server's resources. Therefore, servers must be designed with overload in mind. Given that a server can not handle all of the requests it receives, it is desirable for it to handle the requests that produce the most value. For example, "911" emergency calls should take precedence over other calls; text or picture messages may generate more revenue than local calls; and dropped calls are more frustrating for users than "system busy" messages. Furthermore, each operator may have different policies and values associated with each type of message.

Our solution is to leverage the rich header information contained within SIP messages to classify the incoming stream of messages according to operator-defined rules; and then based on the classification deliver the highest priority messages to the server first. We achieve this with a novel SIP message classification algorithm described in Section 3.

A key motivation behind the design of our classification engine, which is an implementation of the above algorithm, is that it must be programmable with a set of user-specified rules. In this paper, we use the classifier to provide overload control. The need for programmability arises from our earlier observation that SIP can be used to support multiple services / applications and each provider may offer different services and assign a different set of values (e.g., revenue metrics) to them. Clearly then, the rules for overload control that aims to maximize value for an operator under overload, must be different, and rather than create a specialized engine for each operator, it is eminently better to program a common engine with different rules.

Because our classification engine is programmable, it can be used in multiple contexts besides overload control. For example, the classifier can be used as a SIP-aware load balancer in front of a SIP server farm that provides session affinity. It could also potentially be used to prevent denial-of-service attacks by programming it with rules that drop undesirable messages. We are currently studying the multitude of scenarios and architectures where a fast, efficient classification engine would be useful. In this paper, we provide a brief sketch of two additional scenarios, a passive network monitor for SIP and a session-aware dispatcher for a SIP server farm.

Another key point of our classifier design is that it is independent of the underlying transport protocol. As mentioned earlier, SIP can operate over multiple transport protocols: our classification engine operates on SIP messages and assumes that the transport layer connections have been terminated and provide a single FIFO stream of messages as input to the classification engine. However, depending on the transport protocol used, additional transport-layer mechanisms will be needed. We discuss some of these issues briefly in Section 3.2. In this paper, we assume that UDP is transport protocol used which requires no additional (transport-layer) mechanisms in order to apply our classification engine for overload control.

We would also like to point out that the classification algorithm is independent of queuing policy. The end-result of the classification process is to place an incoming message in one of multiple categories. In case of overload control, the categories are realized as queues, which may be serviced using one of many possible queuing schemes such as weighted round-robin or priority schemes. Thus our design decouples the classification algorithm from the queuing mechanism/policy used.

It is worthwhile also mentioning here that a SIP server would normally parse a SIP message before processing it. Thus, a key goal for our classifier design is *not* to duplicate server functionality, and instead extract only the needed information (to enable rule matching) from a subset of the message headers, as will be described in detail in Section 3. This is especially relevant for use in overload control for two reasons: (1) the classifier should take few resources from the SIP server and yet provide a disproportionately higher return, (2) when messages need to be dropped, it is better to drop them earlier in the processing path. The second point ties in with the positioning of the classification engine: as will be discussed in Section 3.3, for performance the classifier is best positioned as an in-kernel module.

We expect the usage of the classifier not *require* any modification to the SIP server application (proxy, redirect server, presence server etc), i.e. the classifier is self-contained with its own rules. However, we also expect configurations where a SIP server would cooperate with the classification engine by programming rules and/or input to the classifier, such as routing policies. An example of such cooperation is in using the classifier for VoIP denial-of-service (DoS) prevention, where the SIP server may use its own methods to detect onset of DoS attacks, and thereupon, insert rules in the classifier to detect and drop undesirable messages before they are processed by the server.

Lastly, it should be noted that the classification-based overload control scheme is triggered only at the onset of overload, and not during normal operation. Since overload detection is non-trivial in itself, a detailed evaluation of detection schemes is outside this scope of this paper [10].

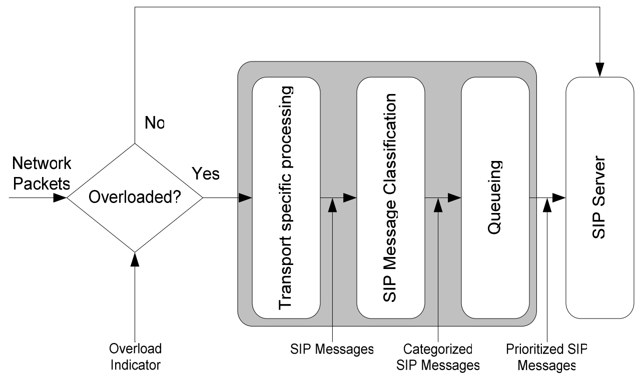

Figure 1: Classifier Usage for Overload Control

3.1 Target Network Scenarios

We present three scenarios for our classification engine: (a) overload control for a SIP server servicing many clients, (b) overload control for a SIP server interconnected with another SIP server, and (c) as a dispatcher for a SIP server farm. Figure 1 shows the classifier deployed for overload control. In (a), each client has a separate transport connection to a SIP server such as a proxy. The case for overload in this scenario arises when a proxy is supporting a large number of users, such as a VoIP service provider like Vonage, and clients come back up at roughly the same time after a regional loss of network connectivity (and thus requiring the clients to re-register). In (b), SIP servers in the core of a service provider network, for example, will receive requests from SIP servers in other administrative domains and/or the same domain, and this request volume is much higher than an access server which supports only user-agents. For a given server-to-server interconnection, all messages are sent over a common transport connection. In both of these cases, our classification engine would be collocated on each server, and programmed with a set of rules to provide overload control (maximizing revenue). Additionally, servers in (a) or (b) may be organized as a server farm front-ended by a SIP-aware dispatcher, which distributes requests among the servers while maintaining session affinity. In this scenario (c), the dispatcher can also be realized through the same classification engine, but programmed with a different set of rules (to ensure for example, the INVITE, 200 OK and ACK messages constituting a session setup are sent to the same server). The key point to be made here that as the classification engine is programmable; it can be used in multiple ways with appropriate configurations. This paper focuses on using the classification engine for overload control.

3.2 Impact of transport layer

As mentioned earlier, SIP can use multiple transport protocols and optionally security support. When SIP is used over UDP, each SIP message is completely contained within a UDP packet [14] and messages from multiple hosts arrive on the same socket, i.e. the classifier can pick up messages from this single socket. In case of TCP, data is delivered to the SIP proxy through a socket interface as a byte-stream. The overload protection mechanism thus needs to be interjected between the TCP implementation and in-kernel socket data structures, so that the byte-stream coming out of the TCP connection can be recognized as a series of messages which are then acted upon by the classifier (and re-ordered). Additionally, when TCP is used, each connection to the proxy results in a separate socket data structure. The classification of messages needs to be done *across* multiple connections. Connections encrypted using SSL must be terminated by a component that is interposed between the user agent and the classifier, as the classifier must inspect unencrypted SIP headers to make its decisions. For IPSec connections, no additional components are needed because IPSec is implemented in the kernel.

3.3 User-level vs. in-kernel design

A classification engine can be executed in user-space or the kernel. In either case, it should be noted that the classification *algorithm* remains exactly the same. When applied to overload control, a kernel-level implementation is preferable, because messages are dropped to reduce load. Clearly, dropping messages needs to happen early in processing path to minimize the amount of processing (CPU, I/O, etc.) resources spent on a message that will ultimately be dropped. For this reason, an in-kernel realization of the classification algorithm can provide higher performance gains than integrating an overload control module within a SIP server (in user-space). Additionally, a kernel-level implementation is more flexible and portable than a user-level implementation, because it can be applied to multiple unmodified SIP servers.

4. CLASSIFICATION ALGORITHM

The input to our classification algorithm is a set of rules, expressed as a conjunction of conditions. Our classification algorithm has a static and a runtime component. The static component consists of rule parsing and creating several tables and bitmaps that allow the runtime portion to operate efficiently. Our algorithm uses three tables: a *header* table, a *header value* table and a *condition* table that store the required message headers, values of those headers and the conditions to evaluate, respectively. We express a list of rules as bitmaps, where each bit represents a condition that must be true for the rule to match. Each rule has associated actions, one of which is a priority for the packet.

The runtime component consists of extracting only those headers (and their values) from a SIP message that are present in the header table, evaluating the conditions in the condition tables, storing the results of the condition evaluations in a bitmap, and then comparing that bitmap with each rule. When a rule matches, a set of actions is applied. We extract only the necessary header values through a single scan of the message, i.e. we parse only the relevant headers and extract only the necessary sub-fields.

A key feature of our algorithm is user-defined data types (such as associative arrays); which allows users to maintain and update customized state as part of rule actions.

In the remainder of this section, we describe our algorithm in detail. Section 4.1 describes message headers and Section 4.2 describes data types. Classification rules are described in Section 4.3, and their grammar in Section 4.4. The static and run-time phases of our algorithm are described in sections 4.6 and 4.7, respectively. We illustrate the algorithm through an example in section 4.8.

4.1 Message Header Types and Specification

The header table consists of a list of SIP message headers, which we have classified according to three types:

- **Simple headers** exist as such in SIP message, e.g. From, Via, and Call-ID.
- **Pseudo-Headers** do not appear as such in a SIP message, but allow us to refer to certain strings or characteristics of the message. For example, Method is a pseudo-header that we define to represent the type of the SIP message such as an INVITE.
- **Derived Headers** are constructed from one or more headers. Derived headers may either be "sub-headers" or tuples of headers.

A sub-header is a string of the form X.Y where X represents a simple header, and Y represents a parameter for header X (e.g., the "tag" parameter of the From header), or a token that we defined to represent values of interest (e.g., "From.URI" represents the URI in a From header).

A tuple of headers may consist of simple, pseudo, or derived headers. For example, a SIP dialog consists of the tag parameter values of the From and To headers, and the Call-ID header value:

```
Dialog = {From.tag,To.tag,Call-ID}
```

Here, From.tag and To.tag are sub-headers, and Call-ID is a simple header. Individual elements of a user-defined derived header are indicated by dotted notation, such as Dialog.From.tag or Dialog.Call-ID.

4.2 Data Types

In conjunction with user-defined derived headers, we also allow the user to specify complex data types such as structures and complex data variables such as associative arrays (hash tables), pointers and scalars. The basic data types are string and integer. Any time a user defines a derived header like Dialog, a type of the same name is also implicitly created. A structure is a data type consisting of a collection of data types. For example:

```
Struct Session = {Dialog D, String State}
```

The element "State" stores the state of a dialog which could be "*established*" or "*shutdown*". To differentiate variables, we prefix each variable with "$", "*", or "%" for scalars, pointers, and associative arrays, respectively. Associative arrays must be defined in terms of a structure. The first element of the structure is the key, and the remaining elements are the values. A list can be created by using an associative array containing keys, but not values. Pointers can only reference elements within an associative array.

All variables are assumed to be global in scope unless explicitly specified to be of local scope. A variable with a global scope exists for the lifetime of the classification process, i.e. it retains its existence across messages and can be modified as a result of classifying individual messages. Global variables are typically used to maintain state that is dynamically generated and modified by the classifier (e.g. an associative array of dialog-ids for ongoing SIP sessions). In contrast, a locally scoped variable retains its value only within the context of a specific rule execution. Moreover, multiple instances of a local variable can be concurrently instantiated (e.g., when running on a multi-processor system the classifier processes packets concurrently). Local variables such as pointers are used to extract an element of a globally scoped list that matches with some set of header values in the message currently under classification.

For example, to maintain a count and an array of session information, three variables could be used:

```
Int: $ActiveSessionCount
Session: %ActiveSessions
Local Session: *CurrentSession
```

An integer suffices for the count. The `ActiveSessions` array uses the previously-defined Session structure, as does the local reference (`CurrentSession`).

The classifier maintains a global variable table (GVT) and a per-message local variable table (LVT). Each entry in the GVT or LVT can store a basic type (string, integer), tuple, or reference. The run-time classifier elements store only indices to the GVT or LVT (much like compiled code references variables by memory locations rather than names).

4.3 Rules

A rule consists of a conjunction (AND) of conditions resulting in an action, along with a priority for each rule. For example:

```
C1 AND C2 AND C3 • A1, Priority
```

Disjunctions (OR) do not need to be supported since disjunction of conditions can be expressed as separate rules, without loss of generality. Rules are applied sequentially, until a matching rule is found. If no match is found, the packet is given the lowest possible priority.

Conditions are of the form "Header op Literal." The header may be a simple header, pseudo-header, or a derived header. The operator can be ==, *subset*, *superset*, or *belongs-to*. The *belongs-to* operator also supports an optional assignment to a pointer. We also support negation for ==, subset, superset, and belongs-to (without assignment). String equality can only be used for headers that have a single value. For multi-valued headers such as Record-Route, Via, or various authorization headers, the subset and superset operators are used. For example, the condition, "Via subset {host1, host2}" matches messages that traverse only "host1," "host2," or both. Set equality is expressed as the conjunction of subset and superset.

The belongs-to operator locates headers in the classifier's state (i.e. associative arrays). For example, "Dialog-ID belongs-to %ActiveSessions" expresses the condition that Dialog-ID is a key in the associative array ActiveSessions. This operator returns a "true" value by returning a pointer to the element in the list that matches the Header; it returns a "false" value if no match is found. Thus, it serves a dual-use of evaluating the Boolean value of a condition and, in addition, returning a pointer value. Thus, we support a special assignment operator, =, that may prefix a belongs-to condition. For example, "*CurrentSession = (Dialog-ID belongs-to %ActiveSessions)" assigns the found item to the *CurrentSession pointer.

4.4 Actions

The actions that an implementation of the classifier provides necessarily includes the ability to color messages with a category (e.g., the priority for overload control), but our classification algorithm does not define what the set of actions is. We have chosen a general representation for our actions, an action table which is a series of actions represented as three-address code. Each action consists of a type, left-hand side, right-hand side, and a next pointer. The type of the action defines the instruction. For example, our in-kernel implementation defines priority assignment; variable assignment; arithmetic operations such as addition, multiplication, division, and modulus; tuple allocation, assignment, and extraction; variable assignment; array insertion and deletion, and more. The left-hand side of an action is a variable which may have side effects (e.g., for addition the left-hand-side serves as both an addend and where the result is stored). As not all instructions affect variables, the left-hand-side may not be specified (e.g., priority assignment does not alter any variable). The right hand-side is used as input and may be an immediate variable (e.g., the constant 1), a state variable (e.g., $ActiveSessionCount), or a header, sub-header, or derived header (e.g., From.tag). For example, the action "ADD $ActiveSessionCount 1" adds one to the value of $ActiveSessionCount. Each rule has a pointer to an action that is executed when the rule is matched, and each action has a next pointer to another entry in the table. To prevent loops only backward pointers are allowed and the first entry (i.e. 0) terminates the action. For clarity, the rule compiler should allow actions to be specified using a richer syntax (e.g., "$A = $B + $C"), but should produce a series of simpler actions (e.g., "ASSIGN $A $B", "ADD $A $C").

4.5 Rules Set Specification Syntax

A complete set of rules begins with type definitions, a list of user-defined headers, variable declarations, and finally an ordered list of rules and actions. The BNF grammar of our rule language is shown in Figure 2. We use Italics for grammar symbols, bold characters for string literals, and roman type for alphanumeric strings (e.g., identifiers). The starting symbol is "RuleSet", which is made up of type declarations, variable declarations, and one or more rules.

4.6 The Static Phase of the Algorithm

1. Extract a set C of unique conditions from the rule set.

2. From the set C, extract a set H of unique headers, which may be pseudo-headers, simple headers or derived headers. For each derived header in H, recursively include the list of headers that comprise it. For example, the Dialog derived header defined in Section 4.2 consists of From.tag, To.tag, and Call-ID. The derived headers From.tag and To.tag would recursively include "From" and "To" in the set H.

3. Create a *Header Table*, with a row for each header in H of the format <*Header, Header-Type, Indices, fn*>. Header is the string representation, such as "From", or "From.tag", "Dialog", or "Method." Header-Type refers to simple, derived or a pseudo-header. For derived headers, there is a corresponding ordered list of indices referring to the headers comprising it, e.g. Dialog would refer to the indices for "From.tag". "To.tag", "Call-ID". For pseudo-headers and derived headers, the element *fn* refers to a function that can extract the value of the header from the message (for pseudo-headers) or its component simple and derived headers (for derived headers). For example, the function pointer for Dialog encodes the logic to create a dialog ID from the list of indices. The header table entries are ordered so that the components of a derived header are computed before the derived header.

4. Associated with the *Header Table* is another table, *Header Value Table*, which for every header (index) in the Header Table will eventually hold a value in the Header Value Table, e.g. the "From" header in the Header Table contains "sip:alice@atl.com" in the associated entry for the *Header Value Table*. These values are populated during run-time, i.e. when a message is being classified. For pseudo-headers and derived headers, the associated function *fn* stores result in the corresponding index in the *Header Value Table*. Each row also contains a type which can be a string, list of strings, tuple, integer, or NULL.

5. Create a *Condition Table*, with a row for condition from the set C consisting of <*operator, header, literal, Assignment-variable*>. The aim is to efficiently represent conditions by storing pointers to header values of a SIP message under classification. At run time, the requisite header values can be referenced in constant time for efficient evaluation of these conditions. In general, *header* is an index to a header table

```
RuleSet = TypeDeclaration* VarDeclaration* Rule+
TypeDeclaration = (UserHeader|Structure)
UserHeader = TypeName = { (Header|TypeName) (,(Header|TypeName) ) * }
Structure = Struct TypeName = { TypeName FieldName (, TypeName FieldName) * }
VarDeclaration = (Local)? TypeName: Kind VarName (, Kind VarName) *
Kind = $|%|*
Rule = Condition (AND Condition) *• Action (, Action) *
Condition = Header (==|!=) String) | Header (subset|superset) {String (, String) *} |
     Header belongs-to %List | VarName = (Header belongs-to %List)
Action = (Assignment | Function | Priority) +
Assignment = (VarName = Value | VarName.FieldName = Value)
```

Figure 2: Classifier Ruleset Grammar

(in the examples below we use X), and the *literal* is a fixed operand that the *header* is compared to. The *operator* is one of the operators defined earlier:

a) For string (in)equality operators, *literal* refers to the literal string that is being compared to a specific SIP message header value, which is specified by an index (*header)* in the Header-Table (HT). The fourth element of the row is unused. An example of this type of entry is *< ==, X, "Alice", NULL >* representing the condition *"From == 'Alice"*.

b) When the operation is *belongs-to*, *literal* refers to a list. For example, the condition with assignment, "*S1=(Dialog belongs-to %L1*)"*, will be represented by a row in the condition table as *<belongs-to, X, %L1, *S1>*.

c) For *subset* and *superset* operators, *header* is a list-valued message header such as "Via", while *literal* is a list of values. For example, a condition such as "Via *subset* {proxy1, proxy2}" is represented as *<subset, X, {proxy1, proxy2}, NULL>*.

6. Bit vector representation: The set of conditions C is efficiently represented as a bit-vector (*Condition bit-vector*), where bit *i* refers to the *i*th condition in the Condition-Table, and will be set 1 iff that condition is true for a message being classified. Additionally, for each rule, create a *Rule bit-vector* where the *i*th bit is 1 iff the rule specification includes the *i*th condition.

4.7 Run-time: classification actions per-message

1. For each header in the header table, return a pointer to the header value if it exists in the message. As there are a limited number of pre-defined SIP headers, we use a simple switch statement. The advantage of doing this is (a) while the entire message needs to be *scanned*, the entire message is not *parsed.* The end-result of this step is to populate each entry in the header value table with a pointer to the corresponding header in the message. For pseudo-headers and derived headers, we perform limited parsing and transformation of the extracted simple headers and store the result in the corresponding index in the header value table.
2. Walk through each row (condition) in the *Condition table* and set the corresponding bit in the *Condition bit-vector* to 1 if the condition is true or 0 if the condition is false. As was indicated earlier, our data structures were carefully designed to make evaluating the condition at run-time efficient: each entry in the *Condition Table* has pointers to the condition's operands (i.e. a literal and a header).
3. Next, we compare the *Condition bit-vector* (CBV) to each rule bit-vector. A rule matches the SIP message under classification, if and only if (R & CBV) == R, where R is the rule bit vector for that rule and "&" is the bitwise *and* operation. In words, if the i^{th} bit is 1 in the rule it must also be one in the CBV. Because the rule-bit-vectors are sorted according to priority, the matching process is stopped after the first match.

4.8 Data Structure Example

We now sketch an example using the following rules:

- `Method=="INVITE" AND To.tag==NULL→High`
- `Method=="INVITE" AND From.URI=="sip:carol" → Medium`
- `From.URI == "sip:alice@atl.com" →Low`

The set of headers H for this rule set is initially "Method", To.tag", and "From.tag." Because "To.tag" and "From.tag" are derived from "To" and "From", respectively, the "To" and "From" headers are added to H forming the set "Method", "To", "From", "To.tag," and "From.tag." The unshaded portion of the following table represents the static portion of the header table.

Index	Parent	Header	Value
0		Method	INVITE
1		To	Bob <sip:bob@biloxi.com>
2		From	Alice <sip:alice@atl.com>;tag=192
3	1	To.tag	NULL
4	2	From.URI	sip:alice@atl.com

The conditions are Method == "INVITE", To.tag == NULL, From.URI == "sip:carol", and From.URI == "sip:alice@atl.com". The unshaded portion of the table below represents the static portion of the condition table.

Index	Op	Header	Literal	Value
0	==	0 (Method)	INVITE	1
1	==	3 (To.tag)	NULL	1
2	==	4 (From.URI)	sip:carol	0
3	==	4 (From.URI)	sip:alice@atl.com	1

Note that duplicate headers and conditions in the rule set are only expressed in the header and condition tables once. Using the condition table, the rules can be expressed as a bitmap. For example, Method == "INVITE" AND To.tag == NULL is expressed as 1100, with each bit corresponding to an index in the condition table. Similarly, Method == "INVITE" AND From.URI == "sip:carol" and From.URI == "sip:alice@atl.com" are expressed as 1010 and 1001, respectively.

The shaded columns represent the run time state for the message, using the sample message from Section 2. The value column in the header table is the header value table, and the values in the condition table represent the condition bit vector, 1101. The condition vector is compared to each of the rules in turn. The first rule matches because 1101 & 1100 == 1100, so processing may stop. The second rule does not match, because bit 3 is not set (1101 & 1010 != 1010). The third rule matches (1101 & 1001 == 1001), but is not executed because the first rule already matched.

5. OVERLOAD CONTROL IMPLEMENTATION

We have developed a prototype classifier for overload control that consists of three components: (1) the core of the classifier is a Linux kernel module responsible for parsing and classifying messages, (2) a user-level rule parser, and (3) a kernel patch that provides an extensible priority queue for UDP sockets. The kernel module is 3,425 lines of code, provides support for parsing all defined SIP headers using a switch statement and associative arrays using linear hashing [12], all of the types, operators, and actions described in the previous section, and several additional actions. The user-level rule parser is 2,121 lines of C code that parses a set of rules and compiles them into a header table, condition table, rule list, and three-address-code action set. Our kernel priority queuing extension adds 626 lines of kernel code that adds a new socket option (SO_QDISC) which allows servers to specify which classification rule set should be used. When a packet is received over the network it is classified using our kernel module, and then inserted into one of n queues. When the server reads from the socket, higher priority messages are returned first. Additionally, if sufficient room is not available in the socket's buffer, lower priority messages are dropped in favor of higher-priority messages.

5.1 Testbed / Workload Used

We ran our classifier and SIP Express Router (SER) 0.9.6 on a dual 3.0 GHz Xeon with 4.5GB of RAM. We used two identical 1.7 GHz Pentium IVs with 512MB of RAM to send and receive the messages. All machines were connected via a 1Gbps Ethernet network.

The workload we used to evaluate our classifier is reflective of what would be typically used by a mobile service provider during call overload, which is to reduce the number of dropped calls due to handoffs in preference to new call setups. Call setup is modeled via SIP INVITE messages; while a call handoff is modeled via re-INVITE messages. The re-INVITE message is structurally same as an INVITE but with a non-empty value of the "tag" parameter in the To header. We evaluated our classifier by sending a sequence of SIP messages to it that consists of INVITEs and re-INVITEs from one SIP client to another through SER with and without our classifier. We configured our classifier with the following set of rules that (a) differentiate re-INVITE vs. INVITE messages using the To.tag field and (b) prioritized re-INVITE messages over INVITE messages :

```
10:Method=="INVITE" && To.tag==NULL -> Color 1
20:Method=="INVITE" && To.tag!=NULL -> Color 0
30:ReqResp != NULL -> Color 1
```

Rule 10 matches INVITE messages without a To.tag field (i.e., the INVITE messages) and assigns them to color 1 (the highest priority is zero). Rule 20 matches INVITE messages with a To.tag, and assigns them priority 0 (i.e. the highest priority). Rule 30 matches all other SIP messages and assigns them to the same priority as INVITE.

When overload occurs, some messages are dropped. We record the number of messages of each type received. We compare the total number of messages received as well as the number of INVITEs and re-INVITES received. We measured our classifier using 10,000-120,000 messages/sec (MPS) in increments of 10,000 MPS, which demonstrates its behavior both with and without overload.

5.2 Observed results

Figure 3 shows the results our experiment with a mix of 75% INVITEs and 25% re-INVITEs (i.e. a handoff-ratio of 25%). We selected these ratios as in practice call handoffs happen less often than call setups. As can be seen in the figure, before overload is reached our classification engine has no impact on the number of messages processed by SER. After overload, throughput decreases by 8.8% for 40,000 MPS to between 18.7-23.3% for 60,000-120,000 MPS, because classifier must process all incoming messages, and we observed that the classifier was able to handle 104,891 MPS at its peak, more than 2.6 times as many as SER could handle at its peak; at the same time SER was processing 31,616 MPS.

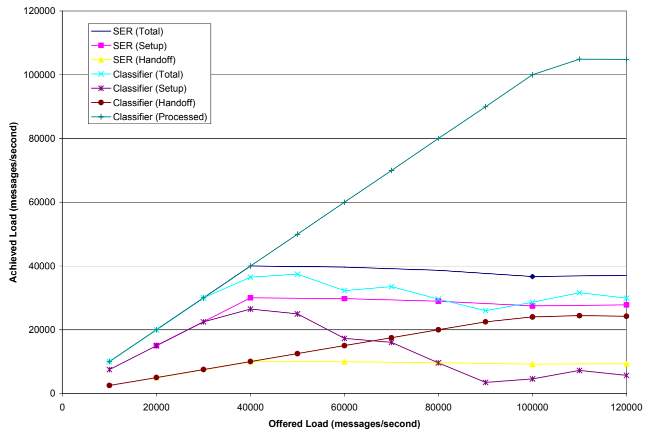

Figure 3: 75% INVITEs, 25% re-INVITEs

Because the classifier was able to process so many more messages than SER alone, it was able to select the high-value messages in this workload for processing, increasing the number of hand-offs processed by 50.9% for 60,000 MPS to 160.2% for 120,000 MPS. Of course this comes at a cost, a corresponding number of setup messages can no longer processed, and thus the call setup throughput decreased between 11.7% and 79.5%.

The performance of the classifier is dependent on the incoming stream of messages; therefore we also ran the same experiment using varying handoff ratios and recorded the maximum number of call handoffs that were processed. The results are shown in Figure 4, the region between the peak handoff capacity of SER and the peak handoff capacity of our classifier is the additional operating region provided by the classifier. As can clearly be seen, the peak number of handoffs scales linearly with the ratio for SER, as SER processes randomly selected messages. The classifier is able to select the high-priority handoffs from the message stream, and thus processes all of the handoffs until the SER itself is saturated at 40,000 MPS, at which point no more capacity is available to process additional hand-off messages.

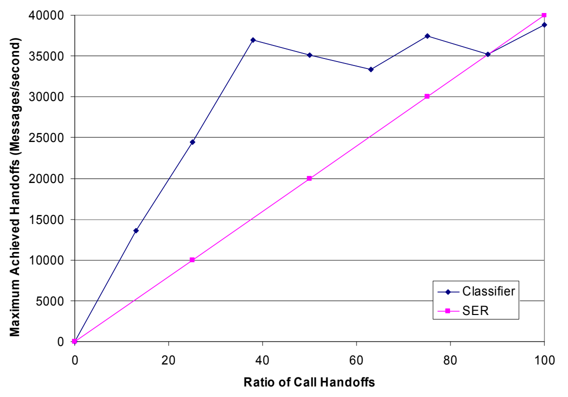

Figure 4: Maximum handoff operating region, as a function of the input stream.

The classifier can provide no benefit at ratios of either 0% or 100%, because there are no high-priority messages to select or low priority messages to discard, respectively. As expected, the classifier is most effective when the offered load is higher than the server's capacity, but number of offered high-priority messages is below the servers capacity (in this case a handoff ratio of 37.5%).

6. ADDITIONAL USE CASES

As mentioned before, the programmability of the classifier through rule-sets makes it applicable to multiple scenarios. We have described one such scenario, namely overload control and demonstrated the performance gains achievable with the classifier. In this section, we briefly outline a two more use cases for the classifier, and describe a partial rule-sets to program the classifier for these scenarios.

6.1 Passive SIP Network Monitoring

Applying the classification algorithm to an input stream of SIP messages can also be useful to simply count the number of messages that fall into a given class and report the number through a network monitoring infrastructure. To prototype a passive SIP network monitor we developed a packet capture program using libpcap that sends packets to our kernel-level classifier [19]. The classifier, then runs the packet through our classification algorithm, assigns the packet a color, and updates a per-color counter which is exported through a Linux `/proc` file system interface. Using the classifier for monitoring is attractive because it receives copies of the packets, and thus does not affect existing infrastructure. There are several options for deployment, the two simplest ones are to: (1) deploy the classifier on the same machine as a SIP server, and (2) deploy the classifier on another machine using switch-based port mirroring to copy all traffic to the classifier. The advantage of the first approach is that no infrastructure changes are required. The advantage of the second approach is that the classifier is completely passive (and thus does not affect the performance of the SIP server), and also that a single classifier instance can monitor more than one SIP server.

For example, we have developed a rule set that extracts basic SIP properties such as the number of requests, successful responses, and failed responses from the SIP message stream. By graphing these rates an operator can quickly get a sense of the health of a SIP server or network. Alternatively, automatic alerts can be triggered based on various conditions worthy of attention (e.g., the ratio of failures to requests). The following rule set divides SIP messages into general classes of interest (for clarity, we use symbolic names for the colors):

```
10: Status.ToInt > 199 &&
      Status.ToInt < 300 -> Color OK
20: Status.ToInt > 299 -> Color Error
30: Status.ToInt < 200 &&
      Status.ToInt > 99 -> Color Provisional
40: Method == "ACK" -> Color Control
50: NOT Method == NULL -> Color Request
60: ReqResp == NULL -> Color NonSIP
70: NOT ReqResp == NULL -> Color OtherSIP
```

As can be seen, these seven rules classify each message into one of seven possible categories: Request, OK (for successful responses), Error (for failure responses), Provisional (for provisional responses), Control (for the ACK method), OtherSIP (an unused catch-all category for SIP messages that are not previously matched), and Non-SIP (for messages that could not be parsed). These seven categories are generic and can apply to any SIP server, but provider or server specific rules are also possible. For example, when authentication is enabled the 401 and 407 error codes are expected for roughly half of all transactions (as the first attempt does not contain authentication information; and is thus a failure). Two simple rules can separate out the 401 and 407 responses from the remainder of the errors:

```
15: Status.ToInt == 401 -> Color NoAuth
15: Status.ToInt == 407 -> Color NoAuth
```

Clearly, more complicated examples are possible and by leveraging the ability of the classifier to store state even more metrics of interest can be added (e.g., the number of retransmissions, concurrent calls, and more).

6.2 SIP-Aware Dispatcher

In many configurations, it is necessary for functional correctness to dispatch all messages that belong to the same transaction to a common server (e.g., messages that comprise the INVITE/OK/ACK three-way handshake for a call setup). Moreover, for correct accounting, it may be necessary to dispatch all messages related to the same call to the same server. This use-case consists of a set of SIP servers that is front-ended by a dispatcher, as shown in Figure 5. A SIP-unaware dispatcher is not suitable, because for proxy-proxy interconnections many sessions will operate over a single connection. We construct a SIP-aware dispatcher using our classifier to determine which server a message should be sent to. After the message is tagged with a server by the classifier, the existing dispatcher framework can forward it as normal.

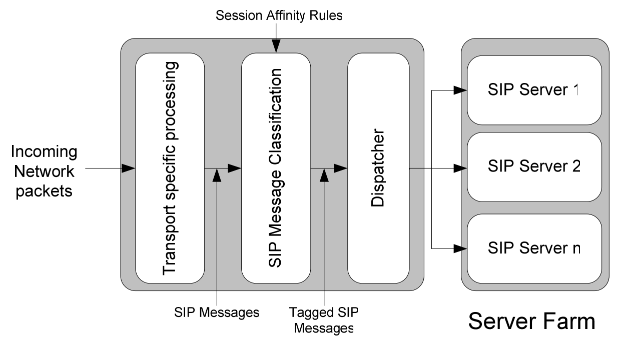

Figure 5 : SIP-Aware Dispatcher

This is accomplished by programming the classifier to create state for the first message in a call, based on the Call-ID message header. The call-ID header value is required to be unique for each call and is the same across all messages in a session [14]. This state is represented by the *Session* structure in the rules below, and includes the specific server id to which the messages will be forwarded. All succeeding messages for this session (e.g. RINGING, OK, BYE), are dispatched to the same server, thereby providing session affinity. The key aspect of the classifier that is being leveraged is the ability to dynamically maintain classification state through an in-kernel associative array.

The following annotated rule set demonstrates how state can be used. First, a session structure is defined and an associative array named %ActiveSessions is created with ID as the key. Two local variables are also required: (1) $NewSession, which is a temporary entry that can be inserted into %ActiveSessions, and (2) *CurrentSession which is used as a pointer into the %ActiveSessions Array.

```
Struct Session = {String ID, Int Server,
    Int Expire}
Global Session: %ActiveSessions
Local Session: $NewSession, *CurrentSession
```

We also define three integers: (1) $MyServer which is the server the current packet should be sent to, (2) $CurrentServer is a round-robin counter to evenly distribute the load across servers, and (3) $nServers is the number of servers in the server farm.

```
Local Int: $MyServer
Global Int: $CurrentServer, $nServers
```

The Init function is used to initialize variables (e.g., setting the number of servers to three), and create a kernel thread to remove array entries that are no longer needed.

```
Init -> $CurrentServer = 0, $nServers = 3,
    ExpiryThread(%ActiveSessions, Expire)
```

Rule 10 matches calls that have an entry in the %ActiveSessions array, and its action sets the Expire element of the entry to the current time plus 15 minutes (900 seconds). Finally, the message is colored with the Server element of the entry, so that the dispatcher knows to use the same server as the previous messages in this call.

```
10:*CurrentSession = Call-ID belongs-to
    %ActiveSessions ->
    *CurrentSession.Expire = Now() + 900,
    Color *CurrentSession.Server
```

Rule 20 matches calls that do not have an entry in the %ActiveSessions array. First, a server is selected and stored in MyServer using a round-robin algorithm. Next, a new session structure is allocated and inserted into the %ActiveSessions array, using the Call-ID as the key. Finally, the $MyServer variable is used to color the packet, thus indicating which server the dispatcher should use.

```
20: NOT Call-ID belongs-to %ActiveSessions ->
    $MyServer = $CurrentServer++ % $nServers,
    $NewSession = (Call-ID, $MyServer, Now() +
    900),
    Insert(%ActiveSessions, $NewSession),
    Color $MyServer
```

Rather than using expiration only, it is possible to modify the rules such that messages which indicate the end of a session trigger state deletion. For example, the following rule removes entries when a final response is received for a BYE transaction:

```
30: Response >= 200 && CSeq.Method == "BYE"
    && *CurrentSession = Call-ID belongs-to
    %ActiveSessions ->
    Remove(%ActiveSessions,*CurrentSession),
    Color *CurrentSession->Server
```

Of course, a timer to purge old state is still required as some transactions may never complete, and additional rules will be needed for corner cases (e.g., dropped messages).

7. FUTURE PLANS

Programmability of the classifier lends itself for use in multiple scenarios. Our current work revolves around developing support for such scenarios, including the corresponding rule sets, e.g. denial-of-service protection for VoIP. This includes identifying scenarios where the classifier and SIP server work cooperatively. Additional systems-level work includes incorporating support for TCP and SSL connections. We are also investigating better methods of detecting overload [10], and specializing the classifier for specific SIP servers such as a presence server, which receive a narrower class of SIP messages, but with richer information in the payload (such as the Presence Information Document). We believe this work opens up a rich set of possibilities to enhance SIP server performance.

8. RELATED WORK

Related work primarily includes IP packet classification, HTTP header inspection in web-proxies, and parsing of SIP messages within SIP proxies/servers such as SIP Express Router (SER) [18]. We are not aware of any earlier work on SIP message header classification per se. SIP proxies and libraries, use efficient parsing techniques such as lazy parsing which include parsing *up to* a required header and/or incremental parsing. However, in our case, the classification engine needs to extract a (small) subset of the header values and thus extracting a programmatically defined subset of headers is more efficient. Additionally, extraction of information from the SIP message is only one aspect of our algorithm. We share a similar bit-vector representation for rules as in [11]; however, unlike [11], we operate on string-based header value pairs, with no predetermined ordering of headers, and our basis for creating rules by extracting a common set of conditions from the rules is conceptually different from creating numeric ranges of interest (e.g. port numbers, IP addresses). SIP messages are syntactically similar to HTTP headers; however the diversity and semantics of SIP headers are much larger than HTTP. Web proxies typically use the content URL in a HTTP request for forwarding it to the right web server; since the number of headers in HTTP requests is small, they usually simply inspect all headers.

A key feature of our classifier is its ability to create, manage and update state across multiple messages in a very general fashion. None of the related work cited support this feature as there was no need for it. However, the notions of transactions and sessions are integral to SIP, so this feature is an essential requirement for SIP classification.

There has been recent related work on overload control of SIP servers. The solution proposed in [13] uses queue-length thresholds within a SIP proxy to determine congestion, and during congestion, it separates INVITE messages and returns a 503 "service unavailable". Current discussion in the IETF [9][16] centers on creating an overload control framework and adopting appropriate new headers to convey additional information beyond a simple 503 response. The approach taken in [5] leverages the fact that throughput of a SIP proxy is higher when processing requests in a transaction-stateless manner, and thus their solution consists of handling a subset of requests statelessly during the onset of congestion, thus trying to avoid overload. In contrast to the aforementioned work, our focus is on creating a mechanism in the form of a programmable engine that enables user-defined policies to be executed efficiently, without modifications to the SIP server.

9. CONCLUSION

In this paper we have presented an algorithm for efficiently classifying SIP messages using a *programmable* set of rules, and applied it for overload control. Our algorithm consists of a static phase and a run-time phase. In the static phase, we define a header table that is a list of attributes to extract from the message and a condition table. These tables eliminate redundancy that is often found in classification rule sets. At run time, classification consists of directed parsing to extract only the relevant headers from the message, evaluating each unique condition, and efficient rule matching using bit vector representation for rules. We implemented an *in-kernel* Linux prototype of the algorithm and programmed the classifier prototype with rules to prioritize handoff messages over call setup messages. Our detailed performance evaluation shows that the in-kernel classification engine is able to process more than twice as many messages than the application-level SIP server, thus significantly extending the operating capacity of the server for high-value messages in a *transparent* manner.

10. ACKNOWLEDGMENTS

We would like to thank John Tracey and Erich Nahum for insightful discussions, and the anonymous ANCS reviewers for useful comments.

11. REFERENCES

[1] A. Acharya, N. Banerjee, B. Sengupta, X. Wang, and C. P. Wright. *Real-time monitoring of SIP infrastructure using message classification.* In Proceedings 3rd annual ACM workshop on Mining Network Data (MineNet 2007). June 2007. 45-50.

[2] G. Camarillo and Miguel-Angel Garcia-Martin. *The 3G IP Multimedia Subsystem (IMS) : Merging the Internet and Cellular worlds*. John Wiley and Sons, 2004.

[3] B. Campbell et al. *The Message Session Relay Protocol.* SIMPLE Working Group Internet-Draft, Jan 2004.

[4] B. Campbell, editor. *Session Initiation Protocol (SIP) Extension for Instant Messaging*. RFC 3428.IETF, Dec 2002.

[5] M. Cortes, J. O. Esteban and H. Jun. *Towards Stateless Core: Improving SIP Proxy Scalability.* IEEE Globecom 2006.

[6] S. Donovan, The SIP INFO Method. RFC 2976. IETF, Oct 2000.

[7] Pankaj Gupta and Nick Mckeown, *Algorithms for packet classification*. IEEE Network, Mar/Apr 2001.

[8] M. Handley and V. Jacobson, *SDP: Session Description Protocol*, RFC 2327, IETF Apr 1998.

[9] V. Hilt and I. Widjaja. *Hop-by-Hop Overload Control for the Session Initiation Protocol (SIP).* IETF Internet-Draft, June 2006.

[10] S. Kasera, J. Pinheiro, C. Loader, T. LaPorta, M. Karaul and A. Hari. *Robust Multiclass Signaling Overload Control*. In Proceedings of the 13th IEEE International Conference on Network Protocols (ICNP'05). November 2005. 246-2

[11] T.V. Lakshman and D. Stiliadis, *High-Speed Policy Based Packet Forwarding using Efficient Multi-Dimensional Range Matching*,', Proceedings of ACM SIGCOMM'98

[12] P. Larson. *Dynamic Hash Tables.* Communications of the ACM 31:4, April 1988. 446-457.

[13] M. Ohta, *Overload control in a SIP Signaling Network*. Transactions on Engineering, Computing and Technology V12, March 2006.

[14] J. Rosenberg et al. *SIP: Session Initiation Protocol*. RFC 3261. IETF, June 2002.

[15] J. Rosenberg, The Session Initiation Protocol (SIP) UPDATE Method, RFC 3311. IETF, Sept 2002

[16] J. Rosenberg. *Requirements for Management of Overload in the Session Initiation Protocol.* IETF Internet-Draft, October 2006.

[17] H. Schulzrinne et al. *RTP: A Transport Protocol for Real-Time Applications*. RFC 1889.IETF, Jan 1996.

[18] SIP Express Router. http://www.iptel.org/ser/

[19] TCPDUMP public repository. http://www.tcpdump.org/

[20] G. Varghese, *Network Algorithmics,: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufman, 2005.