Session Initiation Proposal
Investigation Working Group
Internet-Draft
Expires: March 29, 2004

V. Hilt
Bell Labs/Lucent Technologies
J. Rosenberg
dynamicsoft

September 29, 2003

Supporting Intermediary Session Policies in SIP
draft-hilt-sipping-session-policy-00

Status of this Memo

Copyright Notice

Abstract

Proxy servers play a central role as an intermediary in the establishment of sessions in the Session Initiation Protocol (SIP). In that role, they define and impact policies on call routing, rendezvous, and other call features. However, there is no standard means by which network elements can have any influence on session policies, such as the codecs that are to be used. As such, ad-hoc and non-conformant techniques have been deployed to allow for such session policy mechanisms. In this document, we discuss a complete and standards-based mechanism for session policies.

Table of Contents

1. Introduction 3
2. Framework for Dynamic Policies 5
2.1 Request/Response/ACK-based Framework 6
2.1.1 Constructing the INVITE/UPDATE Request 6
2.1.2 Proxy Processing of Requests 6
2.1.3 Processing Requests and Generating Responses 8
2.1.4 Proxy Processing of Responses 9
2.1.5 Processing Responses and Generating ACKs 10
2.1.6 Processing ACKs 10
2.1.7 Applying Dynamic Policies 10
2.2 Response/ACK-based Framework 11
2.2.1 Creating the INVITE Response 11
2.2.2 Proxy Processing Responses 12
2.2.3 Processing Responses and Generating ACKs 12
2.2.4 Proxy Processing of ACKs/PRACKs 12
2.2.5 Processing ACKs/PRACKs 12
2.3 "Media-Interface" header usage 12
2.4 "Media-Filter" header usage 13
2.5 "Reverse-Media-Filter" header usage 14
3. Dynamic Policy Packages 15
3.1 Media Interface Object 15
3.2 Media Filter Object 15
4. Framework for Static Policies 16
4.1 Static Policies using REGISTER 17
4.1.1 Generating the REGISTER Request 17
4.1.2 Proxy Processing of Requests 18
4.1.3 Processing Requests 19
4.1.4 Applying Static Policies 19
4.2 Static Policies using the Config Framework and XCAP 20
4.2.1 Discovering Policy Servers 21
4.2.2 Subscribing to Static Policies 21
4.2.3 Creating Notifications and Policy Objects 22
4.2.4 Retrieving and Applying Static Policies 23
5. Example Policy Package: Network-based Codec Selection . . . 24
5.1 Dynamic Codec Selection 24
5.1.1 Media Interface Object 24
5.1.2 Media Filter Object 25
5.2 Static Codec Selection 26
6. Security Considerations 27
7. IANA Considerations 28
8. Syntax 29
8.1 Header Fields 29
References 30
Authors' Addresses 31
Intellectual Property and Copyright Statements 32

1. Introduction

The Session Initiation Protocol (SIP) [9] was designed to support establishment and maintenance of end-to-end sessions. Proxy servers provide call routing, authentication and authorization, mobility, and other signaling services that are independent of the session. Effectively, proxies provide signaling policy enforcement. However, numerous scenarios have arisen which require the involvement of proxies in some aspect of the session policy. One scenario is in the traversal of a firewall or NAT. The midcom group has defined a framework for control of firewalls and NATs (generically, middleboxes) [10]. In this model, a midcom agent, typically a proxy server, interacts with the middlebox to open and close media pinholes, obtain NAT bindings, and so on. In this role as a midcom agent, the proxy will need to examine and possibly modify the session description in the body of the SIP message. This modification is to achieve a specific policy objective: to force the media to route through an intermediary.

In another application, SIP is used in a wireless network. The network provider has limited resources for media traffic. During periods of high activity, the provider would like to restrict codec usage on the network to lower rate codecs. In existing approaches, this is frequently accomplished by having the proxies examine the SDP [1] in the body and remove the higher rate codecs or reject the call and require the UA to start over with a different set of codecs.

In yet a third application, SIP is used in a network that has gateways which support a single codec type (say, G.729). When communicating with a partner network that uses gateways with a different codec (say, G.723), the network modifies the SDP to route the session through a converter that changes the G.729 to G.723.

The desire to impact aspects of the session inevitably occurs in domains where the administrator of the SIP domain is also the owner and administrator of an IP network over which it is known that the sessions will traverse. This includes enterprises, Internet access providers, and in some cases, backbone providers. Typical session policies established in such domains influence NAT/firewall traversal or control bandwidth usage by selecting low-rate codecs. The desire to impact aspects of sessions may also occur in domains where services are provided that require the inclusion of a media intermediary such as transcoding or call recording.

Since SIP is the protocol by which the details of these sessions are negotiated, it is natural for providers to wish to impose their session policies through some kind of SIP means. To date, this has been accomplished through SDP editing, a process where proxies dig

into the bodies of SIP messages, and modify them in order to impose their policies. However, this SIP editing technique has many drawbacks. A discussion of these drawbacks can be found in [7].

Our solution is to introduce a framework that allows intermediary elements to request media-level policy operations from user agents. This framework satisfies the requirements listed in [7]. Section 2 introduces a framework for requesting dynamic policies during the establishment or modification of a session. Section 3 discusses the creation of policy packages for this framework. Section 4 introduces two alternative frameworks for static policies. Section 5 gives an example for the use of the dynamic and static framework to select codecs. Section 6 discusses Security and Section 7 IANA considerations. Section 8 describes the syntax of SIP extensions defined in this document.

2. Framework for Dynamic Policies

This framework for dynamic policies enables proxy servers to request session policies from UAs. A session policy may impact aspects of a session description, it may request a UA to perform steps that are outside of the SIP protocol (e.g. contact a NAT/firewall) or expose information about the session that is being set up or modified to a proxy. The syntax and semantics of a specific session policy is not part of this framework and needs to be defined in a separate session policy package. An example for a policy package for codec selection is given in Section 5.

Dynamic session policies may change from call to call. They need to be set up during the establishment or modification of a session. This requires two basic steps: first, UAs need to be able to expose aspects of a session description to proxies and, second, proxies need to be able to request session policies which may be based on the information exposed. In this framework, UAs create Media Interface Objects (MIOs), which describe an aspect of the session being set up or modified. For example, a UA might create an MIO for each of the IP addresses and ports of each media stream, and an MIO for the set of codecs in each stream. Proxies can request policies via Media Filter Objects (MFOs). An MFO describes a set of rules, the UA is requested to execute on a certain media aspect. Each proxy can create MFOs independently. MIOs and MFOs are only useful in conjunction with a session description and must travel in the same SIP message (e.g. in a INVITE request and a 200 OK response).

Session policies can be set up separately for media streams in each direction. The general scheme for requesting policies for media streams in a direction is as follows:

1. The receiver of a media stream creates MIOs (describing relevant media stream aspects) and inserts them into the SIP message, that also carries the corresponding session description.

2. Proxies inspect those MIOs insert MFOs (containing the policy) into the SIP message.

3. Once the message receives the sender of the media stream, it analyzes the session description and the MFOs and decides whether it wants to accept or reject the the requested policies. It applies the accepted policies.

4. The accepted policies are conveyed back to the receiver of a media stream.

The format of MIOs and MFOs is policy specific and needs to be

defined in a policy package (see Section 3).

2.1 Request/Response/ACK-based Framework

Proxies can request policies in INVITE and UPDATE [4] transactions, in which the session description offer is carried in the request and an answer is carried in the response. The basic message flow is depicted in Figure 1.

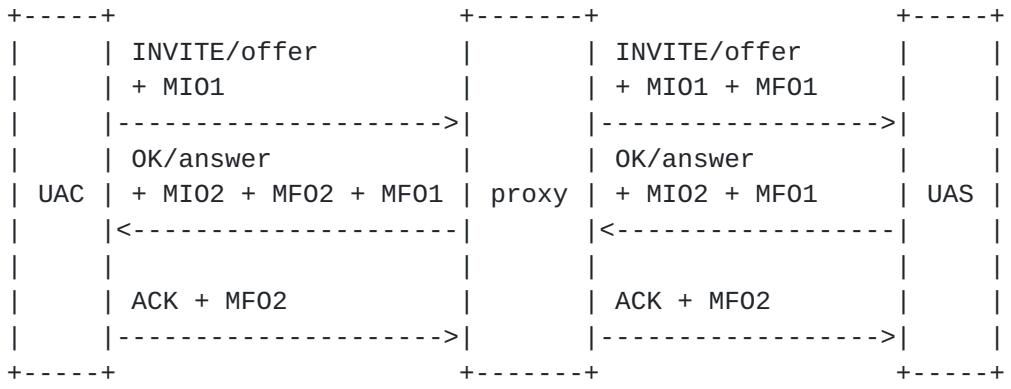

Figure 1

2.1.1 Constructing the INVITE/UPDATE Request

The UAC composes an INVITE or UPDATE request as usual. In addition to the session description, it creates MIOs for those aspects of the session, it wishes to permit the network to examine. For example, if the UAC wants to allow the network to examine the media codecs, it would insert MIOs representing these codecs. The UAC SHOULD expose as much information as possible in MIOs.

Since the MIOs are meant to be inspected by proxies, and since they are provided to enable a SIP feature (proxy insertion of session policy), the MIOs are carried as SIP headers (see Section 2.3).

A UAC that supports this framework MUST insert a SIP Supported header with the option tag "policy".

2.1.2 Proxy Processing of Requests

As the request traverses proxies, the proxies can insert Media Filter Objects (MFOs). MFOs contain the policies, the proxy wants to

request. A proxy can generate MFOs in response to information contained in a specific MIO in the request. These MFOs represent "diffs" that the proxy wants to apply to the MIO. For example, if an MIO contains an IP address and port for receiving an audio stream, a proxy can insert an MFO which changes that address and port to that of a media intermediary. A proxy may inspect MFOs that have been inserted by previous proxies to determine, which policies are already requested. However, MFOs created by a proxy MUST represent the differences to the original MIO and MUST NOT depend on MFOs inserted by previous proxies. A proxy can also generate MFOs independent of the MIOs contained in the request. Such an MFO describes a general policy applicable to the current session. For example, an MFO could contain a list of audio codecs that are allowed in the current session.

The proxy does not modify the MIO - that is fundamental. By specifying the requested modifications in MFOs rather than directly modifying MIOs and the session description, we enable an explicit consent and knowledge model. The UAs can know exactly, which policies where requested against the session.

The session description contained in an INVITE/UPDATE request describes media streams transmitted from UAS to UAC. Consequently, MFOs inserted into an INVITE/UPDATE request MUST contain policies for media streams transmitted in this direction.

A proxy MAY only insert MFOs (or other policy related headers) into the INVITE/UPDATE request, if the UAC has indicated its support for policies by including a Supported header with the value "policy" into the request. If no such Supported header was present and the proxy insists on the use of policies, it MAY return a 421 (Extension Required) response. However, this behavior is NOT RECOMMENDED as it generally breaks interoperability.

A proxy MAY insert a Require header with the option tag "policy" if it wants to make sure that the request fails in case the UAS does not support session policies. A proxy MUST insert a policy Require header if it has marked some policies as required in the MFO (see Section 2.4) and wants the request to fail if these policies are not accepted by the UA. However, not all session policies will be mandatory. Policies could be optional, in which case none of the inserted MFOs would contain a required policy and a policy Require header would not be inserted.

If an MIO contained in the request is not acceptable to the proxy, it MAY insert an MFO indicating the failure or it MAY reject the request by returning a 488 (Not Acceptable Here) response. This enables a proxy to inform the UAC that the information in the MIO is not

acceptable under the current policies or that information required by the current policy was not exposed in an MIO. For example, a proxy, which wants to route a media stream through a firewall, would not accept MIOs containing no information about the transport address. The failure MFO SHOULD explain the reason, why the MIO was not acceptable. Similarly, the 488 response SHOULD include a Warning header field value explaining why the request was rejected. The proxy SHOULD copy the MFOs that caused the problems from the request into the 488 response. This allows the UAC to know exactly why the request has failed and if it can attempt to retry with different MIOs.

[TBD: define warning codes and texts.]

To achieve backwards compatibility with devices that do not support policies, the proxy MUST NOT return a 488 response to requests that do not include a Supported header with the value "policy". A proxy may only reject requests if the UAC has indicated its support for policies and knows how to correct the problem and re-try the request. Rejecting a request is a quick way for the proxy to inform a policy-enabled UAC about policy related problems. It prevents that the request is forwarded to the UAS, which would reject it because of an included failure MFO. Returning a 488 response MUST NOT be used to enforce a policy. Such an enforcement would not be effective since it can be circumvented by a UAC, for example by creating fake MIOs. Using a failure MFO instead of a 488 response to signal a problem has the advantage that both endpoints become aware of the INVITE/UPDATE request and the reason why it failed.

In addition to adding an MFO, a proxy MAY generate an MFO-Reason header. This header contains the domain name of the proxy and explains the reasoning behind the session policy. The end device may present this text string to a human when querying whether the requested policies should be accepted or not.

[TBD: define the format to this header.]

A proxy that supports forking of requests, MAY generate a different set of MFOs for each target the request is sent to.

2.1.3 Processing Requests and Generating Responses

When the INVITE/UPDATE request reaches the UAS, the UAS will know exactly what the UAC indicated in MIOs, and which policies have been requested by intermediate domains. The UAS decides if it wants to accept some or all of these policies. If it decides to reject a policy that is marked as required or if the message contains a failure MFO, the UAS MUST reject the request with a 488 (Not Acceptable Here) response. This response SHOULD include a Warning

header field value explaining, why the policies were not acceptable and a copy of the declined MFOs or the failure MFO.

If all required (and possibly some optional) policies are acceptable to the UAS, it will eventually generate a response which contains a session description answer. If both user agents support reliable provisional responses [8], it is RECOMMENDED that the UAS returns the answer in a reliable provisional response. Using a reliable provisional response has the advantage that the UAC has the chance to reject policies before the session is established.

The UAS then inserts its own set of MIOs for its side of the session into the response. It MUST copy all MFOs it has accepted (required and optional) from the request into the response. The copied MFOs are purely informational, for the benefit of the proxy and the UAC. They inform proxies which policies have been accepted. They also ensure that proxies cannot establish policies without having the UAC become aware of them. The copied MFOs are end-to-end, and not meant for modification by proxies. They MAY be protected by end-to-end security mechanisms.

A UAS MAY only apply this extension to INVITE/UPDATE requests, that contain a Supported header with value "policy". If a UAS applies this extension, it MUST insert a Require header with the value "policy" into the response created. A Supported header with the value "policy" MUST be included in every response to an INVITE/UPDATE request.

2.1.4 Proxy Processing of Responses

If the response contains a Require header with the value "policy", the proxy knows that the UAC and the UAS support the use of session policies and that it may apply this extension. The proxy can determine which policies have been accepted by the UAS by examining the list of MFOs, the UAS has copied into the response.

The proxy can insert MFOs containing policies for media streams transmitted from UAC to UAS into the response to an INVITE request. These MFOs are created and formatted identically to those inserted into the request. If the MIOs contained in the response are not acceptable to a proxy, it may insert a failure MFO.

A proxy could also insert MFOs into the response to an UPDATE request. However, these MFOs would not be copied back to the UAS since UACs do not PRACK or ACK UPDATE responses. Thus, the proxy would not be informed which policies have been accepted and the UAS would not become aware of these policies. Such a behavior violates the requirement that both UAs need to know the set of policies requested along the call path and that the proxy needs to be informed

about accepted policies. It is therefore NOT RECOMMENDED.

[Question: would it make sense to send an additional message from UAC to UAS which carries the MFOs inserted into the response, e.g. a SUBSCRIBE/NOTIFY or INFO?]

2.1.5 Processing Responses and Generating ACKs

After receiving a 1xx or 2xx response, the UAC examines if a Requires header with the value "policy" is present and if the response contains MFOs. If so, it can either reject or accept the policies. If it accepts all policies marked required, the UAC MUST copy the MFOs, that were accepted, into the PRACK or ACK. These MFOs are informational to the proxy and the UAS. They may be protected by end-to-end integrity mechanisms. Due to forking of requests in proxies, the UAC may receive multiple responses from different UASs for one request, which may contain different policies. If the response did not contain a policy Requires header, the UAC must ignore all policy related information in the response (e.g. MFOs).

If the UAC decides to reject some of the required policies or if the response contained a failure MFO, the UAC should terminate the dialog associated with this response. If the UAS has responded with a 2xx response, the UAC must send an ACK and then terminate the dialog with a BYE. If the UAS has responded with a reliable provisional response, the UAC can terminate the dialog without fully establishing it by generating a CANCEL (after sending a PRACK, of course). The UAC does not copy the MFOs from the request into the PRACK or ACK. Instead, the declined MFOs SHOULD be copied into the BYE or CANCEL requests together with a Reason header [2] explaining why the policies were rejected.

[TBD: need to define reason code, phrases etc.]

If the UAC receives a 488 response and the reason explains that existing or missing MIOs caused the rejection, the UAC MAY try to correct the problem (e.g. by adding an additional MIO) and re-send the request.

2.1.6 Processing ACKs

If the MFOs contained in a PRACK or ACK message are not acceptable to the UAS, it may decline them by terminating the dialog with a CANCEL or BYE. The CANCEL or BYE SHOULD contain a copy of the declined MFOs and a Reason header [2] explaining why these policies were rejected.

2.1.7 Applying Dynamic Policies

If both UAs have accepted the policies, they MUST apply them to the media streams they generate. This may involve, for example, sending media to an intermediary indicated in an MFO. Since the user agents know about the full set of intermediaries, they have many options in the event of a failure (detected through an ICMP error, for example). The endpoint can try to send the media to the next intermediary on the path. Or, if the MFO specifies the intermediaries as a FQDN instead of an IP address, the endpoint can attempt to use DNS to find an alternative, and begin routing media through that.

2.2 Response/ACK-based Framework

Proxies may also request policies in INVITE transactions, which carry a session description offer in the response and an answer in the following ACK request. The basic message flow is depicted in Figure 2.

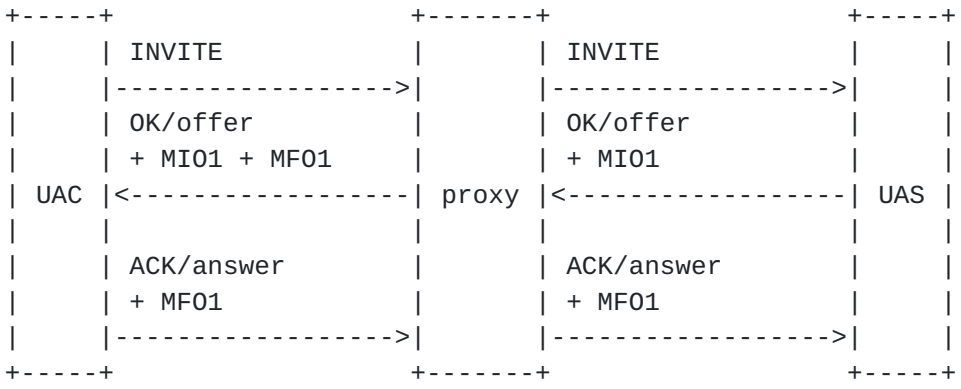

Figure 2

2.2.1 Creating the INVITE Response

The UAS creates the response as usual. It applies this extension to the response, if the request containes a Supported header with the value "policy". The UAS MUST insert a Require header with the value "policy" and SHOULD insert all MIOs it can create for its side of the session description. A Supported header with the value "policy" MUST be included in every response to an INVITE/UPDATE request.

It is RECOMMENDED that the UAS generates a reliable provisional response [8] if supported by both UAs.

2.2.2 Proxy Processing Responses

The proxy MAY add MFOs to responses that contain a Requires header with the value "policy". If an MIO contained in the response is not acceptable for the proxy, it MAY insert a failure MFO.

2.2.3 Processing Responses and Generating ACKs

The UAC may or may not accept the policies contained in the response. If it accepts all required policies, it MUST copy the accepted MFOs into the PRACK or ACK. It may protect these MFOs with end-to-end integrity mechanisms. If it declines at least one of the required policies or if the response contained a failure MFO, the UAC does not copy these MFOs into the PRACK or ACK and SHOULD terminate the dialog associated with this response.

2.2.4 Proxy Processing of ACKs/PRACKs

The proxy could insert MFOs into the PRACK or ACK. However, these MFOs would not be copied back to the UAC, which would violate the requirement that both UAs and the proxy should know the set of policies used in a session. This behavior is therefore NOT RECOMMENDED.

2.2.5 Processing ACKs/PRACKs

If the MFOs contained in a PRACK or ACK message are not acceptable to the UAS, it may decline them by terminating the dialog.

2.3 "Media-Interface" header usage

The Media-Interface header value contains Media Interface Objects (MIOs) created by a UA. The structure and semantics of MIOs needs to be defined in a policy package. However, the following general rules apply to Media-Interface header values:

The Media-Interface header value MUST consist of the policy package name, under which the MIO was created.

The Media-Interface header MAY contain a signature parameter which enables proxies to verify the identity of the UA and the integrity of the MIOs.

A UA creates a separate Media-Interface header value for each policy package it supports. A policy package MAY require the creation of multiple Media-Interface headers with different MIOs. The UAC SHOULD create MIOs for all policy packages it supports. MIOs SHOULD contain as much information about the session as possible.

In the following example, the UA supports the packages foo and bar. It exposes data1 and data2 for package foo and data3 for package bar in MIOs.

```
Media-Interface: foo;foo_param1=data1;foo_param2=data2,
bar;bar_param=data3
```

2.4 "Media-Filter" header usage

Media-Filter headers serve as a container for Media Filter Objects (MFOs). Each MFO is contained in a separate Media-Filter header value. Media-Filter header values implement a stack, which enables each proxy on the way to "push" its MFOs on top of the set of existing MFOs. The Media-Filter headers implement one single stack, which contains the MFOs for all packages. If a proxy wants to insert an MFO, it inserts the respective Media-Filter header value before the topmost Media-Filter header value.

A UA, which receives a SIP message containing MFOs, processes them one after another by popping them from the stack.

The structure and semantics of MFOs needs to be defined in a policy package. However, the following general rules apply to Media-Filter header values:

The Media-Filter header value MUST consist of the policy package name, under which the MFO was created.

The following general parameters are defined for Media-Filter headers. They provide basic information about the MFO to UAs even if they don't support the policy package used.

- Domain. The domain parameter carries the identity of the domain, which requested the policy. It MUST be present in each MFO.

- Consequences (cns). The consequences parameter is be used by the proxy to indicate the consequences of rejecting the policy to the UA. This parameter also enables a UA to determine if the acceptance of a policy is mandatory for establishing the session or not. The consequences parameter contains a consequences code, which has a "required" and an "optional" range. An MFO SHOULD contain a consequences code. An MFO is optional if the consequences parameter is not present.

- Signature. A MFO MAY contain a signature, generated by the domain that inserted the MFO. This allows the endpoints to verify the identities of the domains, which have requested session policy,

and the integrity of those policies.

[TBD: define consequence codes.]

A failure MFO is a special MFO, which indicates that the session is not acceptable to the proxy. A failure MFO is an MFO with consequence code 999. Additional package specific parameters MAY be present in a failure MFOs.

[TBD: define reason codes and texts for failure MFOs.]

In the following example, the proxy in domain example1.com has requested policies for package foo and the proxy in domain example2.com has requested policies for the packages foo and bar.

```
Media-Filter: foo;domain=example2.com;cns=100;foo_param=data1,
bar;domain=example2.com;cns=300;bar_param=data1,
foo;domain=example1.com;foo_param=data2,
```

2.5 "Reverse-Media-Filter" header usage

The Reverse-Media-Filter header is used to convey the MFOs, a UA has accepted, back to the peer UA. A Reverse-Media-Filter header contains a copy of the accepted MFOs and has the same structure as the Media-Filter header.

3. Dynamic Policy Packages

This section describes aspects that need to be considered when dynamic policy packages are defined.

3.1 Media Interface Object

This section MUST be present in a policy package. It defines the structure of Media Interface Objects used within this package.

A policy package MUST describe the semantics of an MIO. It MUST describe how proxies are supposed to interpret the information contained in an MIO.

3.2 Media Filter Object

This section MUST be present in a policy package. It defines the structure of Media Filter Objects used within this package.

Media Filter Objects (MFOs) may define the differences to an existing MIO. However, it is very important that MFOs don't just define a diff to an MIO, in the Unix sense. This is because it is important that the endpoints understand the semantics of a requested policy, not just the syntactical change that is needed to affect that policy. A MFO may also define a general policy which is independent of an MIO.

A policy package MUST describe exactly how a UA is supposed to apply the policy contained in an MFO. In particular, the policy package MUST describe how the information in the MFO is applied to the session description and if additional steps need to be taken when accepting the policy. This process MUST enable a UA to determine the consequences of accepting the policy before actually executing the necessary steps.

4. Framework for Static Policies

In contrast to dynamic policies, which can be defined on a call-by-call basis, static policies remain stable for a longer period of time, typically in the range of hours or days. In principle, static policies could be set up using the dynamic framework. However, establishing the same policies over and over again in every call is expensive, causing the continuous transmission of the same information during call setup, and possibly adding to call setup latencies. In general, static policies provide a way of conveying information to the UAC that is useful for setting up a call in the current environment. For example, a static policy could list the codecs that are currently allowed in the network or it may specify the MIOs the UAC is supposed to include in an INVITE/UPDATE request. In another example, the UAC has to traverse a NAT and is informed of the TURN [5] relay it should contact in advance of a call via a static policy.

Requesting static instead of dynamic policies is most beneficial for network providers, which are involved in many sessions a UA establishes. The following two types of network providers will most likely have an interest in requesting static policies:

- The Home Domain Provider is responsible for providing SIP service to a SIP user. Typically, this is the domain present in the URI in the address-of-record of a registration. The home domain provider may maintain user preferences or subscriptions to services, which involve static policies. For example, a user may have subscribed to a networked call recording service. The respective static policy makes sure, that all voice streams are routed through the recording intermediary.

- The Access Network Provider is responsible for providing IP service to a SIP agent. This may be the same provider as the home domain provider. However, they may be different in scenarios where a user roams in a foreign network or obtains SIP services and IP connectivity from different providers. Access Network Providers are often interested in static policies, which influence the traffic in their networks such as restricting the use of high bandwidth codecs.

This framework for static policies allows network providers to convey static policies to UAs. It does not define the structure or semantics of static policies. Static policies need to be defined in policy packages. An example for a static policy package for codec selection is discussed in Section 5.

This document proposes two different frameworks for static policies,

which both have their advantages and disadvantages. However, it is expected that one of the frameworks will become obsolete as this draft evolves.

4.1 Static Policies using REGISTER

This framework uses the REGISTER message to convey static policies to a UA. REGISTER messages are created at the time a device registers at a network, which is also the time static policies need to be exchanged. This message traverses the access network domain and the home domain, which are the domains typically interested in requesting static policies. An advantage of exchanging policy information in conjunction with a REGISTER message is the low overhead. No extra messages need to be created and clients do not need to implement additional protocols. The drawbacks are the tight coupling between registrations and static policies. Since registrations and policies are conveyed in the same message, proxies and registrars also need to be policy servers. Policies and registrations need to be refreshed in the same interval. The basic call flow in this framework is depicted in Figure 5.

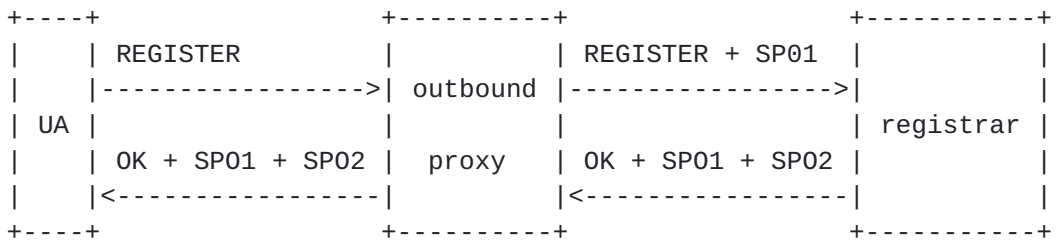

Figure 5

4.1.1 Generating the REGISTER Request

A UAC which supports this framework MUST insert a Supported header with the option tag "stat_policy".

To allow the access network provider to request static policies, the UA SHOULD attempt to discover an outbound proxy, for example by using the methods described in the SIP Framework for User Agent Configuration [3]. If an outbound proxy is available, the UAC SHOULD include it in the route set used for the REGISTER request.

4.1.2 Proxy Processing of Requests

Proxies may insert Static Policy Objects (SPOs) into a REGISTER request. A proxy MAY only insert SPOs if the REGISTER request contains a Supported header with the option tag "stat_policy". If no such header is present, the proxy MAY try to request the desired policies using the dynamic framework. It MUST NOT reject the request if the "stat_policy" Supported header value is not present.

An SPO represents a static policy, the UA is requested to apply to the sessions it establishes. An SPO can apply to all sessions established by the UA. However, it may also affect only a subset of these sessions. The scope of a static policy MUST be defined in a policy package. The policy package may either have global scope or define a scope attribute that is populated by proxies as needed. Possible scopes are:

- Sessions for a certain address of record (i.e. sessions created for a certain local user). This is useful if an end device supports multiple identities and, for example, only a subset of them has subscribed to a service requiring policies.

- Sessions to a certain remote URI. For example, a policy for NAT traversal might only apply to sessions to or from external addresses.

- Outgoing/incoming sessions only. A static policy may apply only to sessions initiated by the local/the remote UA.

- A certain media stream. This enables the specification of policies on a stream-by-stream basis. For example, a policy for audio codec selection only applies to audio streams.

- Media streams in the incoming or outgoing direction. This enables independent policies for the media streams in each direction.

SPOs are represented in a SIP header. The structure of such a header needs to be defined in policy packages. The SPO MUST contain the identity of the domain, which requested the policy. It MAY also contain a signature allowing the UA to verify the identity of that domain and the integrity of the SPO.

In addition to an SPO, a proxy MAY generate an SPO-Reason header. This header contains the domain name of the proxy requesting the policy and explains the reasoning behind the session policy. The end device may present this text string to a human when querying whether the requested policies should be accepted or not.

[TBD: define the format to this header.]

Static policies will usually be changed by the provider from time to time. This requires that the UA is able to refresh its view on static policies. In the REGISTER-based framework, this is done by periodically refreshing policies together with registrations. If a proxy wants to influence the refresh interval, it needs to determine the expiration intervals of all contacts in the REGISTER request as described in Section 10.3 of [9]. It MUST use the shortest of the determined expiration intervals as the expiration interval for inserted SPOs. If this interval is too long, the proxy MAY shorten it by changing the respective values in the REGISTER request (either the "expires" parameter value of the respective Contact header fields or the Expires header field value). If no expiration interval is given in the request, the proxy MAY insert an Expires header field with the desired value. This procedure makes sure that the UA generates the next REGISTER request at least at the time SPOs need to be refreshed.

A proxy SHOULD insert SPOs, which scope is a certain address of record, into the REGISTER request for that address. SPOs, that are not tied to a certain address of record, MAY be inserted into every REGISTER request. However, if a device creates multiple REGISTER requests for different addresses of record, a proxy SHOULD insert these generic SPOs only into the REGISTER requests of one address (typically the first encountered by a proxy). This avoids the retransmission of these SPOs in every REGISTER request. A proxy must make sure that these SPOs are inserted into different REGISTER requests in case the address used expires or is removed.

4.1.3 Processing Requests

The REGISTER request eventually reaches the registrar, which creates a response. If the request contains a Supported Header with the option tag "stat_policy", the registrar MAY insert SPOs representing its static policies into the response. If the request contained SPOs inserted by proxies on the way, the registrar must copy these SPOs from the request into the response.

The registrar must follow the same procedures as a proxy when creating SPOs (see Section 4.1.2).

4.1.4 Applying Static Policies

At the time the response reaches the UAC, it contains all static policies that have been requested by proxies and the registrar. The UAC can decide to accept or reject these policies. Since no session is established at this point, the UAC does not need to inform the proxies or registrar about its decision.

The UA MUST apply the accepted policies to new sessions it is establishing. For example, if the policy lists the audio codecs allowed in a wireless network, the UA includes only those audio codecs in the session description offers and answers it creates. The UA does not explicitly confirm the acceptance of a static policy in an INVITE or UPDATE message. The proxy might be able to determine the use of static policies by examining the actions of the UA (e.g. contacting a TURN relay) or the information it exposes in an MIO. The proxy may also have mechanisms in place to enforce its static policies.

A provider may have static and dynamic policies in place. Since dynamic policies are requested during session setup, they automatically override a static policy. In fact, a provider may use dynamic policies to quickly apply the change of a static policy, without waiting until all UAs have refreshed their static policies. Dynamic policies may also be used for clients that do not support static policies.

A UA, which has received an updated set of static policies in a REGISTER response, MAY apply them to existing sessions for example by issuing a re-INVITE request.

4.2 Static Policies using the Config Framework and XCAP

Static policies influence the way a UA sets up a session. In this respect, static policies can be regarded as device configuration information and the mechanisms for conveying configuration information to devices can be re-used for requesting static policies. However, an important difference between configuration information and static policies is that configuration information is usually applied in any case whereas a UA can decide whether or not it wants to accept static policies.

This document describes the use of the Framework for SIP User Agent Configuration [3] and The Extensible Markup Language Configuration Access Protocol (XCAP) [6] to deliver static policies to a UA. The SIP Framework for User Agent Configuration [3] enables a UA to discover configuration servers and retrieve a URL to configuration data. It also enables a configuration server to notify clients if new or updated configuration information is available. XCAP on the other hand provides the means to compose HTTP URLs, which point to components in configuration documents stored in XML format on a HTTP server. It allows clients to access these components on a fine grained basis.

The major advantage of these configuration mechanisms is that they decouple requesting static policies from other tasks such as

registering. Static policies may be provided by any entity in the network and not only by those involved in the registration process. Also, static policies may be updated at any time, independent of refreshing registrations. A drawback of this approach is the overhead needed for transmitting extra messages and the implementation overhead for providing the additional protocols in UAs and policy servers.

4.2.1 Discovering Policy Servers

The SIP Framework for User Agent Configuration [3] defines a SUBSCRIBE/NOTIFY-based mechanism, which enables UAs to subscribe to static policy information. Before being able to receive notifications about the availability of static policies, the UA must discover the relevant policy servers.

The first group of policy servers relevant for a UA are the home policy servers, i.e. servers responsible for the home domains of registered users. The URIs of these servers are derived by taking the host component of each registered address of record and adding "policy" as "userinfo" component to this address. For example, if an address of record is sip:bob@example.com, the UA would use the URI sip:policy@example.com to contact the policy server. Using "policy" as the "userinfo" component enables proxies to route the request to a policy server.

The second group of policy servers a UA is supposed to contact are access network policy servers. A UA SHOULD discover the URIs of these policy servers by using the procedures described in [3]. To distinguish policy from other device configuration servers, the UA MUST use the term "policy" wherever [3] requests the use of "sipuaconfig" when generating the URI.

Finally, UA may also have manually configured URIs to policy servers.

4.2.2 Subscribing to Static Policies

A UA supporting static policies MUST send a SUBSCRIBE request to the discovered policy servers. It generates the SUBSCRIBE request as described in [3].

The To header field of a SUBSCRIBE request MUST be populated with the SIP URI of the policy server. The UA uses the From header field to indicate on behalf of whom it is subscribing to static policies. The UA SHOULD subscribe each registered user to all manually configured servers and all access network servers. To do so, the UA sends a separate SUBSCRIBE request for each registered address of record (which is inserted into the From header field) to every of the above

policy servers. This way, all of these servers will learn about the users the UA has registered and may provide different static policies for them. A home policy server will typically not be able to provide static policies for users not registered in its domain. Therefore, a UA SHOULD only send a SUBSCRIBE for the respective address of record to a home policy server.

The UA SHOULD subscribe to all manually configured policy servers and to all discovered home policy servers. It SHOULD subscribe to access network servers until the first successful response is received.

UAs MUST include the event package name "policy" in the Event header.

4.2.3 Creating Notifications and Policy Objects

The policy server generates NOTIFY messages as described in [3]. In particular, it notifies the subscribers of any changes in static policies. The policy server does not insert policy objects into the body of a NOTIFY. Instead, it includes URLs pointing to the policy objects on a server. This saves bandwidth and enables a server to insert all current policies in a NOTIFY instead of tracking the policies that are new to a UA. The UA can then decide which policy objects it wants to retrieve.

The structure of a particular policy object needs to be defined in a policy package. The policy objects defined for this framework are based on XCAP [6]. As such, a policy package defines an XCAP application usage specification. This specification defines the XML schema and the semantics of policy documents, which represent the desired static policy objects.

Policy servers will frequently maintain multiple static policy documents. For example, they may maintain a document describing general policies and multiple user-specific documents, which describe policies for particular users. A policy server MUST insert URLs to all relevant policy documents into a NOTIFY. For example, a NOTIFY generated for user bob@example.com could contain URLs to the generic policies applicable in domain example.com and the specific policies of user bob@example.com. The NOTIFY MUST contain URLs to all relevant policy documents even if they have not been changed since the transmission of the previous NOTIFY. Each URL MUST have an associated Content-ID entity header, which SHOULD change every time the referred policy document changes. This enables clients to determine if they have the latest version of the policy without having to download and compare the documents.

Static policy objects are created by applying the procedures discussed in Section 4.1.2. They MUST be stored in the policy

document tree on an XCAP server. The XCAP naming conventions for the construction of URLs MUST be applied. In particular, global policy objects MUST be stored in the "global" document sub-tree whereas user specific policy objects MUST be stored in the "users" sub-tree.

A policy server MUST ensure that all URLs, it is inserting into a NOTIFY, refer to policy objects that are actually accessible in the XCAP document tree. This is in particular important if a policy server creates policy objects on the fly. For example, a new policy object might be generated when a new user requests policies for the first time. A policy server MUST NOT delay the transmission of a NOTIFY just because a relevant policy object is not yet available on the XCAP server. Instead, it SHOULD not refer to the new object in the current NOTIFY and create an additional NOTIFY as soon as the policy object becomes available on the XCAP server.

The policy server MAY use XCAP to upload policy objects to the XCAP server.

4.2.4 Retrieving and Applying Static Policies

After receiving a NOTIFY, the UA MUST determine if any of the URLs are pointing to a policy document, that is new or has changed since it was last downloaded. The UA SHOULD retrieve new or updated policy documents as soon as possible. After having retrieved a policy document, the UA can decide if it wants to accept the policies or not. Since no session is established at this point, the UAC does not need to inform the policy server about its decision.

The UA must follow the procedures for applying static policies discussed in Section 4.1.4.

The XCAP server MUST only allow read access for UAs to policy documents. Policies are used to request a certain behavior from a UA. The UA can decide if it wants to accept these policies or not but it can not modify them. In this respect, policy documents differ from device configuration data, which typically can be edited by the device. The XCAP server MUST determine if a client has authorization to read a resource. The default behavior is that the client of user X can read the policies under the "global" and the "users/X" document tree.

5. Example Policy Package: Network-based Codec Selection

5.1 Dynamic Codec Selection

This dynamic policy package enables a proxy to influence the codecs that are used within a session. The UAs are enabled to expose the codecs they support in MIOs. The MFOs created by the proxy contain the list of codecs allowed in the domain. The package is currently defined based on session descriptions in SDP [1] format. However, its is not restricted to SDP and can be used with other session description formats respectively.

The name of this package is "codec". This package name is carried in the Media-Interface, the Media-Filter and the Reverse-Media-Filter header as defined in this specification.

5.1.1 Media Interface Object

A codec MIO describes the codecs that are supported by the UA creating the MIO.

This policy package defines a media type parameter for codec MIOs (in addition to the general parameters for MIOs).

The parameter name consists of the media type, for which this MIO provides a policy. If used with a SDP session description, it MUST have the same value as the media name attribute in the media description (m=) of the corresponding SDP announcement. Typical values are "audio", "video", "application" and "data".

The value of this parameter consists of a media stream id and one or more codec formats. The media stream id provides an identifier for a media stream. It MUST have a value that is unique within the scope of the session description. The media stream id MUST be present in each codec MIO and it MUST NOT be zero. The codec format describes the codecs allowed for this media type. The format of the value is specific to each media type and has the same structure as the SDP rtpmap parameter. A UA SHOULD list all codecs is has listed for the media stream in the corresponding session description. All elements of the parameter value are concatenated with a "+" symbol.

An example for a Media-Interface header containing a codec MIO is

```
Media-Interface: codec;audio=7736ai+pcmu/8000/1+pcma/8000/1+
eg711u/8000/1;video=hha9s8sd0+h261/90000
```

This header specifies two media streams, an audio and a video stream. The available audio codecs are pcmu, pcma, and eg711u. The only video

codec supported is h261.

A proxy would create the following SDP announcement template from this MIO:

```
m=audio <port> RTP/AVP 0 8 <no1>
a=rtpmap:0 pcmu/8000/1
a=rtpmap:8 pcma/8000/1
a=rtpmap:<no1> eg711u/8000/1
m=video <port> RTP/AVP 31
a=rtpmap:31 h261/90000
```

5.1.2 Media Filter Object

A codec MFO describes the list of codecs that are allowed under this session policy.

In addition to the general header parameters, this policy package defines a media type parameter, which is structured exactly as the media type parameter in codec MIOs. The semantics of this parameter is as follows:

The media stream id MUST refer to a media stream contained in an MIO or contain the value zero. If the media stream id refers to a media stream in an MIO, the codec policy applies only to the referred media stream. If the media stream id is zero, the policy apply to all streams of the respective media type. A proxy MAY insert multiple media type parameters with different media stream id's for the same media type, if it wants to define different policies for different streams of the same type.

The media format element MUST list all codecs that are allowed under the current policy. It MAY contain codecs that are not listed in a respective MIO.

[TBD: Define consequence codes.]

An example for a Media-Filter header containing a codec MFO is

```
Media-Filter: codec;domain=example1.com;
audio=0+pcmu/8000/1+eg711u/8000/1,
codec;domain=example2.com;cns=100;
audio=0+eg711u/8000/1;video=0
```

This header contains two MFOs, one inserted by proxy example1.com and one by example2.com. The policy of domain example1.com is that the set of allowed audio codecs is limited to pcmu and eg711u.

Consequences for UAs rejecting this policy are not defined, which also indicates that this policy is optional. Domain example1.com has no policy for video codecs. The policy of domain example2.com is that only audio codec eg711u and no video can be used. Consequence of rejecting this policy is code 100, which indicates that the policy is mandatory. All policies apply to audio and video streams in general and are not bound to a stream listed in the MIO.

A UA would create the following SDP filter from these MFOs:

```
m=audio <port> RTP/AVP <no1>
a=rtpmap:<no1> eg711u/8000/1
m=video <port> RTP/AVP
```

A UA, that accepts this policy, removes all audio and video codecs that are not listed in the SDP filter.

5.2 Static Codec Selection

[TBD: Give an example for static policies.]

6. Security Considerations

[TBD.]

7. IANA Considerations

[TBD.]

8. Syntax

This section describes the syntax extensions required for session policies.

8.1 Header Fields

This table expands on tables 2 and 3 in SIP [9] and on table 1 and table 2 in Reliability of Provisional Responses in SIP [8].

Header field	where	proxy	ACK	BYE	CAN	INV	OPT	REG	PRACK
Media-Interface		r	o	-	-	o	-	-	o
Media-Filter		a	o	-	-	o	-	-	o
Reverse-Media-Filter	r		-	-	-	o	-	-	-
Reverse-Media-Filter			o	-	-	-	-	-	o

References

[1] Handley, M. and V. Jacobson, "SDP: Session Description Protocol", RFC 2327, April 1998.

[2] Oran, D., Schulzrinne, H. and G. Camarillo, "The Reason Header Field for the Session Initiation Protocol", draft-ietf-sip-reason-01 (work in progress), May 2002.

[3] Petrie, D., "A Framework for SIP User Agent Configuration", draft-ietf-sipping-config-framework-00 (work in progress), March 2003.

[4] Rosenberg, J., "The Session Initiation Protocol (SIP) UPDATE Method", RFC 3311, October 2002.

[5] Rosenberg, J., "Traversal Using Relay NAT (TURN)", draft-rosenberg-midcom-turn-01 (work in progress), March 2003.

[6] Rosenberg, J., "The Extensible Markup Language (XML) Configuration Access Protocol (XCAP)", draft-rosenberg-simple-xcap-00 (work in progress), May 2003.

[7] Rosenberg, J., "Requirements for Session Policy for the Session Initiation Protocol (SIP)", draft-ietf-sipping-session-policy-req-00 (work in progress), June 2003.

[8] Rosenberg, J. and H. Schulzrinne, "Reliability of Provisional Responses in Session Initiation Protocol (SIP)", RFC 3262, June 2002.

[9] 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.

[10] Srisuresh, P., Kuthan, J., Rosenberg, J., Molitor, A. and A. Rayhan, "Middlebox communication architecture and framework", RFC 3303, August 2002.

Authors' Addresses

Volker Hilt
Bell Labs/Lucent Technologies
101 Crawfords Corner Rd
Holmdel, NJ 07733
USA

EMail: volkerh@bell-labs.com

Jonathan Rosenberg
dynamicsoft
72 Eagle Rock Avenue
East Hanover, NJ 07936
USA

EMail: jdrosen@dynamicsoft.com

Intellectual Property Statement

Full Copyright Statement

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.