Network Working Group
Request for Comments: 3507
Category: Informational

J. Elson
A. Cerpa
UCLA

April 2003

# Internet Content Adaptation Protocol (ICAP)

## Status of this Memo

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

## Copyright Notice



## IESG Note

The Open Pluggable Services (OPES) working group has been chartered to produce a standards track protocol specification for a protocol intended to perform the same of functions as ICAP. However, since ICAP is already in widespread use the IESG believes it is appropriate to document existing usage by publishing the ICAP specification as an informational document. The IESG also notes that ICAP was developed before the publication of RFC 3238 and therefore does not address the architectural and policy issues described in that document.

## Abstract

ICAP, the Internet Content Adaption Protocol, is a protocol aimed at providing simple object-based content vectoring for HTTP services. ICAP is, in essence, a lightweight protocol for executing a "remote procedure call" on HTTP messages. It allows ICAP clients to pass HTTP messages to ICAP servers for some sort of transformation or other processing ("adaptation"). The server executes its transformation service on messages and sends back responses to the client, usually with modified messages. Typically, the adapted messages are either HTTP requests or HTTP responses.

Table of Contents

## 1. Introduction

As the Internet grows, so does the need for scalable Internet services. Popular web servers are asked to deliver content to hundreds of millions of users connected at ever-increasing bandwidths. The model of centralized, monolithic servers that are responsible for all aspects of every client's request seems to be reaching the end of its useful life.

To keep up with the growth in the number of clients, there has been a move towards architectures that scale better through the use of replication, distribution, and caching. On the content provider side, replication and load-balancing techniques allow the burden of client requests to be spread out over a myriad of servers. Content providers have also begun to deploy geographically diverse content distribution networks that bring origin-servers closer to the "edge" of the network where clients are attached. These networks of distributed origin-servers or "surrogates" allow the content provider to distribute their content whilst retaining control over the integrity of that content. The distributed nature of this type of deployment and the proximity of a given surrogate to the end-user enables the content provider to offer additional services to a user which might be based, for example, on geography where this would have been difficult with a single, centralized service.

ICAP, the Internet Content Adaption Protocol, is a protocol aimed at providing simple object-based content vectoring for HTTP services. ICAP is, in essence, a lightweight protocol for executing a "remote procedure call" on HTTP messages. It allows ICAP clients to pass HTTP messages to ICAP servers for some sort of transformation or other processing ("adaptation"). The server executes its transformation service on messages and sends back responses to the client, usually with modified messages. The adapted messages may be either HTTP requests or HTTP responses. Though transformations may be possible on other non-HTTP content, they are beyond the scope of this document.

This type of Remote Procedure Call (RPC) is useful in a number of ways. For example:

- Simple transformations of content can be performed near the edge of the network instead of requiring an updated copy of an object from an origin server. For example, a content provider might want to provide a popular web page with a different advertisement every time the page is viewed. Currently, content providers implement this policy by marking such pages as non-cachable and tracking user cookies. This imposes additional load on the origin server and the network. In our architecture, the page could be cached once near the edges of the network. These edge caches can then use an ICAP call to a nearby ad-insertion server every time the page is served to a client.

  Other such transformations by edge servers are possible, either with cooperation from the content provider (as in a content distribution network), or as a value-added service provided by a client's network provider (as in a surrogate). Examples of these kinds of transformations are translation of web pages to different human languages or to different formats that are appropriate for special physical devices (e.g., PDA-based or cell-phone-based browsers).

- Surrogates or origin servers can avoid performing expensive operations by shipping the work off to other servers instead. This helps distribute load across multiple machines. For example, consider a user attempting to download an executable program via a surrogate (e.g., a caching proxy). The surrogate, acting as an ICAP client, can ask an external server to check the executable for viruses before accepting it into its cache.

- Firewalls or surrogates can act as ICAP clients and send outgoing requests to a service that checks to make sure the URI in the request is allowed (for example, in a system that allows parental control of web content viewed by children). In this case, it is a *request* that is being adapted, not an object returned by a response.

In all of these examples, ICAP is helping to reduce or distribute the load on origin servers, surrogates, or the network itself. In some cases, ICAP facilitates transformations near the edge of the network, allowing greater cachability of the underlying content. In other examples, devices such as origin servers or surrogates are able to reduce their load by distributing expensive operations onto other machines. In all cases, ICAP has also created a standard interface for content adaptation to allow greater flexibility in content distribution or the addition of value added services in surrogates.

There are two major components in our architecture:

1. Transaction semantics -- "How do I ask for adaptation?"

2. Control of policy -- "When am I supposed to ask for adaptation, what kind of adaptation do I ask for, and from where?"

Currently, ICAP defines only the transaction semantics. For example, this document specifies how to send an HTTP message from an ICAP client to an ICAP server, specify the URI of the ICAP resource requested along with other resource-specific parameters, and receive the adapted message.

Although a necessary building-block, this wire-protocol defined by ICAP is of limited use without the second part: an accompanying application framework in which it operates. The more difficult policy issue is beyond the scope of the current ICAP protocol, but is planned in future work.

In initial implementations, we expect that implementation-specific manual configuration will be used to define policy. This includes the rules for recognizing messages that require adaptation, the URIs of available adaptation resources, and so on. For ICAP clients and servers to interoperate, the exact method used to define policy need not be consistent across implementations, as long as the policy itself is consistent.

IMPORTANT:
Note that at this time, in the absence of a policy-framework, it is strongly RECOMMENDED that transformations SHOULD only be performed on messages with the explicit consent of either the content-provider or the user (or both). Deployment of transformation services without the consent of either leads to, at best, unpredictable results. For more discussion of these issues, see Section 7.

Once the full extent of the typical policy decisions are more fully understood through experience with these initial implementations, later follow-ons to this architecture may define an additional policy control protocol. This future protocol may allow a standard policy definition interface complementary to the ICAP transaction interface defined here.

## 2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14, RFC 2119 [2].

The special terminology used in this document is defined below. The majority of these terms are taken as-is from HTTP/1.1 [4] and are reproduced here for reference. A thorough understanding of HTTP/1.1 is assumed on the part of the reader.

connection:
A transport layer virtual circuit established between two programs for the purpose of communication.

message:
The basic unit of HTTP communication, consisting of a structured sequence of octets matching the syntax defined in Section 4 of HTTP/1.1 [4] and transmitted via the connection.

request:
An HTTP request message, as defined in Section 5 of HTTP/1.1 [4].

response:
An HTTP response message, as defined in Section 6 of HTTP/1.1 [4].

resource:
A network data object or service that can be identified by a URI, as defined in Section 3.2 of HTTP/1.1 [4]. Resources may be available in multiple representations (e.g., multiple languages, data formats, size, resolutions) or vary in other ways.

client:
A program that establishes connections for the purpose of sending requests.

server:
An application program that accepts connections in order to service requests by sending back responses. Any given program may be capable of being both a client and a server; our use of these terms refers only to the role being performed by the program for a particular connection, rather than to the program's capabilities in general. Likewise, any server may act as an origin server, surrogate, gateway, or tunnel, switching behavior based on the nature of each request.

origin server:
The server on which a given resource resides or is to be created.

proxy:
An intermediary program which acts as both a server and a client for the purpose of making requests on behalf of other clients. Requests are serviced internally or by passing them on, with possible translation, to other servers. A proxy MUST implement both the client and server requirements of this specification.

cache:
A program's local store of response messages and the subsystem that controls its message storage, retrieval, and deletion. A cache stores cachable responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests. Any client or server may include a cache, though a cache cannot be used by a server that is acting as a tunnel.

cachable:
A response is cachable if a cache is allowed to store a copy of the response message for use in answering subsequent requests. The rules for determining the cachability of HTTP responses are defined in Section 13 of [4]. Even if a resource is cachable, there may be additional constraints on whether a cache can use the cached copy for a particular request.

surrogate:
A gateway co-located with an origin server, or at a different point in the network, delegated the authority to operate on behalf of, and typically working in close co-operation with, one or more origin servers. Responses are typically delivered from an internal cache. Surrogates may derive cache entries from the origin server or from another of the origin server's delegates. In some cases a surrogate may tunnel such requests.

Where close co-operation between origin servers and surrogates exists, this enables modifications of some protocol requirements, including the Cache-Control directives in [4]. Such modifications have yet to be fully specified.

Devices commonly known as "reverse proxies" and "(origin) server accelerators" are both more properly defined as surrogates.

New definitions:

ICAP resource:
Similar to an HTTP resource as described above, but the URI refers to an ICAP service that performs adaptations of HTTP messages.

ICAP server:
Similar to an HTTP server as described above, except that the application services ICAP requests.

ICAP client:
A program that establishes connections to ICAP servers for the purpose of sending requests. An ICAP client is often, but not always, a surrogate acting on behalf of a user.

## 3. ICAP Overall Operation

Before describing ICAP's semantics in detail, we will first give a general overview of the protocol's major functions and expected uses. As described earlier, ICAP focuses on modification of HTTP requests (Section 3.1), and modification of HTTP responses (Section 3.2).

### 3.1 Request Modification

In "request modification" (reqmod) mode, an ICAP client sends an HTTP request to an ICAP server. The ICAP server may then:

1) Send back a modified version of the request. The ICAP client may then perform the modified request by contacting an origin server; or, pipeline the modified request to another ICAP server for further modification.

2) Send back an HTTP response to the request. This is used to provide information useful to the user in case of an error (e.g., "you sent a request to view a page you are not allowed to see").

3) Return an error.

ICAP clients MUST be able to handle all three types of responses. However, in line with the guidance provided for HTTP surrogates in Section 13.8 of [4], ICAP client implementors do have flexibility in handling errors. If the ICAP server returns an error, the ICAP client may (for example) return the error to the user, execute the unadapted request as it arrived from the client, or re-try the adaptation again.

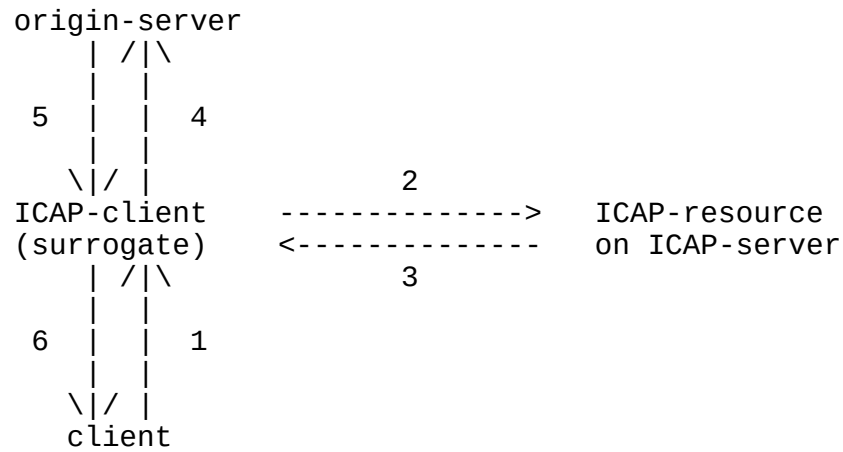
We will illustrate this method with an example application: content filtering. Consider a surrogate that receives a request from a client for a web page on an origin server. The surrogate, acting as an ICAP client, sends the client's request to an ICAP server that performs URI-based content filtering. If access to the requested URI is allowed, the request is returned to the ICAP client unmodified. However, if the ICAP server chooses to disallow access to the requested resources, it may either:

1) Modify the request so that it points to a page containing an error message instead of the original URI.

2) Return an encapsulated HTTP response that indicates an HTTP error.

This method can be used for a variety of other applications; for example, anonymization, modification of the Accept: headers to handle special device requirements, and so forth.

Typical data flow:




1. A client makes a request to a ICAP-capable surrogate (ICAP client) for an object on an origin server.

2. The surrogate sends the request to the ICAP server.

3. The ICAP server executes the ICAP resource's service on the request and sends the possibly modified request, or a response to the request back to the ICAP client.

If Step 3 returned a request:

4. The surrogate sends the request, possibly different from original client request, to the origin server.

5. The origin server responds to request.

6. The surrogate sends the reply (from either the ICAP server or the origin server) to the client.

### 3.2 Response Modification

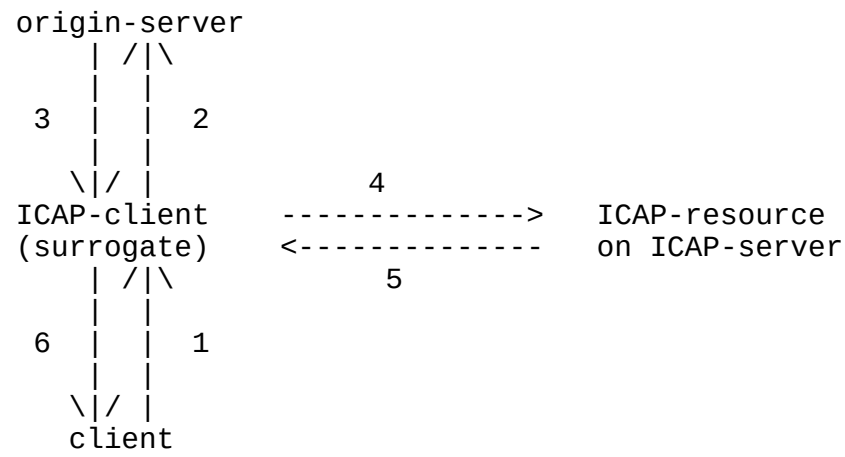
In the "response modification" (respmod) mode, an ICAP client sends an HTTP response to an ICAP server. (The response sent by the ICAP client typically has been generated by an origin server.) The ICAP server may then:

1) Send back a modified version of the response.

2) Return an error.

The response modification method is intended for post-processing performed on an HTTP response before it is delivered to a client. Examples include formatting HTML for display on special devices, human language translation, virus checking, and so forth.

Typical data flow:




1. A client makes a request to a ICAP-capable surrogate (ICAP client) for an object on an origin server.

2. The surrogate sends the request to the origin server.

3. The origin server responds to request.

4. The ICAP-capable surrogate sends the origin server’s reply to the ICAP server.

5. The ICAP server executes the ICAP resource’s service on the origin server’s reply and sends the possibly modified reply back to the ICAP client.

6. The surrogate sends the reply, possibly modified from the original origin server's reply, to the client.

# 4. Protocol Semantics

## 4.1 General Operation

ICAP is a request/response protocol similar in semantics and usage to HTTP/1.1 [4]. Despite the similarity, ICAP is not HTTP, nor is it an application protocol that runs over HTTP. This means, for example, that ICAP messages can not be forwarded by HTTP surrogates. Our reasons for not building directly on top of HTTP are discussed in Section 8.1.

ICAP uses TCP/IP as a transport protocol. The default port is 1344, but other ports may be used. The TCP flow is initiated by the ICAP client to a passively listening ICAP server.

ICAP messages consist of requests from client to server and responses from server to client. Requests and responses use the generic message format of RFC 2822 [3] -- that is, a start-line (either a request line or a status line), a number of header fields (also known as "headers"), an empty line (i.e., a line with nothing preceding the CRLF) indicating the end of the header fields, and a message-body.

The header lines of an ICAP message specify the ICAP resource being requested as well as other meta-data such as cache control information. The message body of an ICAP request contains the (encapsulated) HTTP messages that are being modified.

As in HTTP/1.1, a single transport connection MAY (perhaps even SHOULD) be re-used for multiple request/response pairs. The rules for doing so in ICAP are the same as described in Section 8.1.2.2 of [4]. Specifically, requests are matched up with responses by allowing only one outstanding request on a transport connection at a time. Multiple parallel connections MAY be used as in HTTP.

## 4.2 ICAP URIs

All ICAP requests specify the ICAP resource being requested from the server using an ICAP URI. This MUST be an absolute URI that specifies both the complete hostname and the path of the resource being requested. For definitive information on URL syntax and semantics, see "Uniform Resource Identifiers (URI): Generic Syntax and Semantics," RFC 2396 [1], Section 3. The URI structure defined by ICAP is roughly:

```
ICAP_URI = Scheme ":" Net_Path [ "?" Query ]

Scheme = "icap"

Net_Path = "//" Authority [ Abs_Path ]

Authority = [ userinfo "@" ] host [ ":" port ]
```

ICAP adds the new scheme "icap" to the ones defined in RFC 2396. If the port is empty or not given, port 1344 is assumed. An example ICAP URI line might look like this:

```
icap://icap.example.net:2000/services/icap-service-1
```

An ICAP server MUST be able to recognize all of its hosts names, including any aliases, local variations, and numeric IP addresses of its interfaces.

Any arguments that an ICAP client wishes to pass to an ICAP service to modify the nature of the service MAY be passed as part of the ICAP-URI, using the standard "?"-encoding of attribute-value pairs used in HTTP. For example:

```
icap://icap.net/service?mode=translate&lang=french
```

## 4.3 ICAP Headers

The following sections define the valid headers for ICAP messages. Section 4.3.1 describes headers common to both requests and responses. Request-specific and response-specific headers are described in Sections 4.3.2 and 4.3.3, respectively.

User-defined header extensions are allowed. In compliance with the precedent established by the Internet mail format [3] and later adopted by HTTP [4], all user-defined headers MUST follow the "X-" naming convention ("X-Extension-Header: Foo"). ICAP implementations MAY ignore any "X-" headers without loss of compliance with the protocol as defined in this document.

Each header field consists of a name followed by a colon (":") and the field value. Field names are case-insensitive. ICAP follows the rules describe in section 4.2 of [4].

### 4.3.1 Headers Common to Requests and Responses

The headers of all ICAP messages MAY include the following directives, defined in ICAP the same as they are in HTTP:

Cache-Control
Connection
Date
Expires
Pragma
Trailer
Upgrade

Note in particular that the "Transfer-Encoding" option is not allowed. The special transfer-encoding requirements of ICAP bodies are described in Section 4.4.

The Upgrade header MAY be used to negotiate Transport-Layer Security on an ICAP connection, exactly as described for HTTP/1.1 in [4].

The ICAP-specific headers defined are:

Encapsulated (See Section 4.4)

### 4.3.2 Request Headers

Similar to HTTP, ICAP requests MUST start with a request line that contains a method, the complete URI of the ICAP resource being requested, and an ICAP version string. The current version number of ICAP is "1.0".

This version of ICAP defines three methods:

REQMOD - for Request Modification (Section 4.8)
RESPMOD - for Response Modification (Section 4.9)
OPTIONS - to learn about configuration (Section 4.10)

The OPTIONS method MUST be implemented by all ICAP servers. All other methods are optional and MAY be implemented.

User-defined extension methods are allowed. Before attempting to use an extension method, an ICAP client SHOULD use the OPTIONS method to query the ICAP server’s list of supported methods; see Section 4.10. (If an ICAP server receives a request for an unknown method, it MUST give a 501 error response as described in the next section.)

Given the URI rules described in Section 4.2, a well-formed ICAP request line looks like the following example:

RESPMOD icap://icap.example.net/translate?mode=french ICAP/1.0

A number of request-specific headers are allowed in ICAP requests, following the same semantics as the corresponding HTTP request headers (Section 5.3 of [4]). These are:

Authorization
Allow (see Section 4.6)
From (see Section 14.22 of [4])
Host (REQUIRED in ICAP as it is in HTTP/1.1)
Referer (see Section 14.36 of [4])
User-Agent

In addition to HTTP-like headers, there are also request headers unique to ICAP defined:

Preview (see Section 4.5)

### 4.3.3 Response Headers

ICAP responses MUST start with an ICAP status line, similar in form to that used by HTTP, including the ICAP version and a status code. For example:

ICAP/1.0 200 OK

Semantics of ICAP status codes in ICAP match the status codes defined by HTTP (Section 6.1.1 and 10 of [4]), except where otherwise indicated in this document; n.b. 100 (Section 4.5) and 204 (Section 4.6).

ICAP error codes that differ from their HTTP counterparts are:

100 - Continue after ICAP Preview (Section 4.5).

204 - No modifications needed (Section 4.6).

400 - Bad request.

404 - ICAP Service not found.

405 - Method not allowed for service (e.g., RESPMOD requested for service that supports only REQMOD).

408 - Request timeout. ICAP server gave up waiting for a request from an ICAP client.

500 - Server error. Error on the ICAP server, such as "out of disk space".

501 - Method not implemented. This response is illegal for an OPTIONS request since implementation of OPTIONS is mandatory.

502 - Bad Gateway. This is an ICAP proxy and proxying produced an error.

503 - Service overloaded. The ICAP server has exceeded a maximum connection limit associated with this service; the ICAP client should not exceed this limit in the future.

505 - ICAP version not supported by server.

As in HTTP, the 4xx class of error codes indicate client errors, and the 5xx class indicate server errors.

ICAP's response-header fields allow the server to pass additional information in the response that cannot be placed in the ICAP's status line.

A response-specific header is allowed in ICAP requests, following the same semantics as the corresponding HTTP response headers (Section 6.2 of [4]). This is:

Server (see Section 14.38 of [4])

In addition to HTTP-like headers, there is also a response header unique to ICAP defined:

ISTag (see Section 4.7)

#### 4.3.4 ICAP-Related Headers in HTTP Messages

When an ICAP-enabled HTTP surrogate makes an HTTP request to an origin server, it is often useful to advise the origin server of the surrogate's ICAP capabilities. Origin servers can use this information to modify its response accordingly. For example, an origin server may choose not to insert an advertisement into a page if it knows that a downstream ICAP server can insert the ad instead.

Although this ICAP specification can not mandate how HTTP is used in communication between HTTP clients and servers, we do suggest a convention: such headers (if used) SHOULD start with "X-ICAP". HTTP clients with ICAP services SHOULD minimally include an "X-ICAP-Version: 1.0" header along with their application-specific headers.

### 4.4 ICAP Bodies: Encapsulation of HTTP Messages

The ICAP encapsulation model is a lightweight means of packaging any number of HTTP message sections into an encapsulating ICAP message-body, in order to allow the vectoring of requests, responses, and request/response pairs to an ICAP server.

This is accomplished by concatenating interesting message parts (encapsulatED sections) into a single ICAP message-body (the encapsulatING message). The encapsulated sections may be the headers or bodies of HTTP messages.

Encapsulated bodies MUST be transferred using the "chunked" transfer-coding described in Section 3.6.1 of [4]. However, encapsulated headers MUST NOT be chunked. In other words, an ICAP message-body switches from being non-chunked to chunked as the body passes from the encapsulated header to encapsulated body section. (See Examples in Sections 4.8.3 and 4.9.3.). The motivation behind this decision is described in Section 8.2.

#### 4.4.1 The "Encapsulated" Header

The offset of each encapsulated section's start relative to the start of the encapsulating message's body is noted using the "Encapsulated" header. This header MUST be included in every ICAP message. For example, the header

```
Encapsulated: req-hdr=0, res-hdr=45, res-body=100
```

indicates a message that encapsulates a group of request headers, a group of response headers, and then a response body. Each of these is included at the byte-offsets listed. The byte-offsets are in decimal notation for consistency with HTTP's Content-Length header.

The special entity "null-body" indicates there is no encapsulated body in the ICAP message.

The syntax of an Encapsulated header is:

```
encapsulated_header: "Encapsulated: " encapsulated_list
encapsulated_list: encapsulated_entity |
                   encapsulated_entity ", " encapsulated_list
encapsulated_entity: reqhdr | reshdr | reqbody | resbody | optbody
reqhdr  = "req-hdr" "=" (decimal integer)
reshdr  = "res-hdr" "=" (decimal integer)
reqbody = { "req-body" | "null-body" } "=" (decimal integer)
resbody = { "res-body" | "null-body" } "=" (decimal integer)
optbody = { "opt-body" | "null-body" } "=" (decimal integer)
```

There are semantic restrictions on Encapsulated headers beyond the syntactic restrictions. The order in which the encapsulated parts appear in the encapsulating message-body MUST be the same as the order in which the parts are named in the Encapsulated header. In other words, the offsets listed in the Encapsulated line MUST be monotonically increasing. In addition, the legal forms of the Encapsulated header depend on the method being used (REQMOD, RESPMOD, or OPTIONS). Specifically:

```
REQMOD  request  encapsulated_list: [reqhdr] reqbody
REQMOD  response encapsulated_list: {[reqhdr] reqbody} |
                                    {[reshdr] resbody}
RESPMOD request  encapsulated_list: [reqhdr] [reshdr] resbody
RESPMOD response encapsulated_list: [reshdr] resbody
OPTIONS response encapsulated_list: optbody
```

In the above grammar, note that encapsulated headers are always optional. At most one body per encapsulated message is allowed. If no encapsulated body is presented, the "null-body" header is used instead; this is useful because it indicates the length of the header section.

Examples of legal Encapsulated headers:

```
/* REQMOD request: This encapsulated HTTP request's headers start
 * at offset 0; the HTTP request body (e.g., in a POST) starts
 * at 412. */
Encapsulated: req-hdr=0, req-body=412

/* REQMOD request: Similar to the above, but no request body is
 * present (e.g., a GET).  We use the null-body directive instead.
 * In both this case and the previous one, we can tell from the
 * Encapsulated header that the request headers were 412 bytes
 * long. */
Encapsulated: req-hdr=0, null-body=412

/* REQMOD response: ICAP server returned a modified request,
 * with body */
Encapsulated: req-hdr=0, req-body=512

/* RESPMOD request: Request headers at 0, response headers at 822,
 * response body at 1655.  Note that no request body is allowed in
 * RESPMOD requests. */
Encapsulated: req-hdr=0, res-hdr=822, res-body=1655

/* RESPMOD or REQMOD response: header and body returned */
Encapsulated: res-hdr=0, res-body=749
```

```
/* OPTIONS response when there IS an options body */
Encapsulated: opt-body=0

/* OPTIONS response when there IS NOT an options body */
Encapsulated: null-body=0
```

#### 4.4.2 Encapsulated HTTP Headers

By default, ICAP messages may encapsulate HTTP message headers and entity bodies. HTTP headers MUST start with the request-line or status-line for requests and responses, respectively, followed by interesting HTTP headers.

The encapsulated headers MUST be terminated by a blank line, in order to make them human readable, and in order to terminate line-by-line HTTP parsers.

HTTP/1.1 makes a distinction between end-to-end headers and hop-by-hop headers (see Section 13.5.1 of [4]). End-to-end headers are meaningful to the ultimate recipient of a message, whereas hop-by-hop headers are meaningful only for a single transport-layer connection. Hop-by-hop headers include Connection, Keep-Alive, and so forth. All end-to-end HTTP headers SHOULD be encapsulated, and all hop-by-hop headers MUST NOT be encapsulated.

Despite the above restrictions on encapsulation, the hop-by-hop Proxy-Authenticate and Proxy-Authorization headers MUST be forwarded to the ICAP server in the ICAP header section (not the encapsulated message). This allows propagation of client credentials that might have been sent to the ICAP client in cases where the ICAP client is also an HTTP surrogate. Note that this does not contradict HTTP/1.1, which explicitly states "A proxy MAY relay the credentials from the client request to the next proxy if that is the mechanism by which the proxies cooperatively authenticate a given request." (Section 14.34).

The Via header of an encapsulated message SHOULD be modified by an ICAP server as if the encapsulated message were traveling through an HTTP surrogate. The Via header added by an ICAP server MUST specify protocol as ICAP/1.0.

### 4.5 Message Preview

ICAP REQMOD or RESPMOD requests sent by the ICAP client to the ICAP server may include a "preview". This feature allows an ICAP server to see the beginning of a transaction, then decide if it wants to

opt-out of the transaction early instead of receiving the remainder of the request message. Previewing can yield significant performance improvements in a variety of situations, such as the following:

- Virus-checkers can certify a large fraction of files as "clean" just by looking at the file type, file name extension, and the first few bytes of the file. Only the remaining files need to be transmitted to the virus-checking ICAP server in their entirety.

- Content filters can use Preview to decide if an HTTP entity needs to be inspected (the HTTP file type alone is not enough in cases where "text" actually turns out to be graphics data). The magic numbers at the front of the file can identify a file as a JPEG or GIF.

- If an ICAP server wants to transcode all GIF87 files into GIF89 files, then the GIF87 files could quickly be detected by looking at the first few body bytes of the file.

- If an ICAP server wants to force all cacheable files to expire in 24 hours or less, then this could be implemented by selecting HTTP messages with expiries more than 24 hours in the future.

ICAP servers SHOULD use the OPTIONS method (see Section 4.10) to specify how many bytes of preview are needed for a particular ICAP application on a per-resource basis. Clients SHOULD be able to provide Previews of at least 4096 bytes. Clients furthermore SHOULD provide a Preview when using any ICAP resource that has indicated a Preview is useful. (This indication might be provided via the OPTIONS method, or some other "out-of-band" configuration.) Clients SHOULD NOT provide a larger Preview than a server has indicated it is willing to accept.

To effect a Preview, an ICAP client MUST add a "Preview:" header to its request headers indicating the length of the preview. The ICAP client then sends:

- all of the encapsulated header sections, and

- the beginning of the encapsulated body section, if any, up to the number of bytes advertised in the Preview (possibly 0).

After the Preview is sent, the client stops and waits for an intermediate response from the ICAP server before continuing. This mechanism is similar to the "100-Continue" feature found in HTTP, except that the stop-and-wait point can be within the message body. In contrast, HTTP requires that the point must be the boundary between the headers and body.

For example, to effect a Preview consisting of only encapsulated HTTP headers, the ICAP client would add the following header to the ICAP request:

```
Preview: 0
```

This indicates that the ICAP client will send only the encapsulated header sections to the ICAP server, then it will send a zero-length chunk and stop and wait for a "go ahead" to send more encapsulated body bytes to the ICAP server.

Similarly, the ICAP header:

```
Preview: 4096
```

Indicates that the ICAP client will attempt to send 4096 bytes of origin server data in the encapsulated body of the ICAP request to the ICAP server. It is important to note that the actual transfer may be less, because the ICAP client is acting like a surrogate and is not looking ahead to find the total length of the origin server response. The entire ICAP encapsulated header section(s) will be sent, followed by up to 4096 bytes of encapsulated HTTP body. The chunk body terminator "0\r\n\r\n" is always included in these transactions.

After sending the preview, the ICAP client will wait for a response from the ICAP server. The response MUST be one of the following:

- 204 No Content. The ICAP server does not want to (or can not) modify the ICAP client's request. The ICAP client MUST treat this the same as if it had sent the entire message to the ICAP server and an identical message was returned.

- ICAP reqmod or respmod response, depending what method was the original request. See Section 4.8.2 and 4.9.2 for the format of reqmod and respmod responses.

- 100 Continue. If the entire encapsulated HTTP body did not fit in the preview, the ICAP client MUST send the remainder of its ICAP message, starting from the first chunk after the preview. If the entire message fit in the preview (detected by the "EOF" symbol explained below), then the ICAP server MUST NOT respond with 100 Continue.

When an ICAP client is performing a preview, it may not yet know how many bytes will ultimately be available in the arriving HTTP message that it is relaying to the HTTP server. Therefore, ICAP defines a way for ICAP clients to indicate "EOF" to ICAP servers if one

unexpectedly arrives during the preview process. This is a particularly useful optimization if a header-only HTTP response arrives at the ICAP client (i.e., zero bytes of body); only a single round trip will be needed for the complete ICAP server response.

We define an HTTP chunk-extension of "ieof" to indicate that an ICAP chunk is the last chunk (see [4]). The ICAP server MUST strip this chunk extension before passing the chunk data to an ICAP application process.

For example, consider an ICAP client that has just received HTTP response headers from an origin server and initiates an ICAP RESPMOD transaction to an ICAP server. It does not know yet how many body bytes will be arriving from the origin server because the server is not using the Content-Length header. The ICAP client informs the ICAP server that it will be sending a 1024-byte preview using a "Preview: 1024" request header. If the HTTP origin server then closes its connection to the ICAP client before sending any data (i.e., it provides a zero-byte body), the corresponding zero-byte preview for that zero-byte origin response would appear as follows:

```
\r\n
0; ieof\r\n\r\n
```

If an ICAP server sees this preview, it knows from the presence of "ieof" that the client will not be sending any more chunk data. In this case, the server MUST respond with the modified response or a 204 No Content message right away. It MUST NOT send a 100-Continue response in this case. (In contrast, if the origin response had been 1 byte or larger, the "ieof" would not have appeared. In that case, an ICAP server MAY reply with 100-Continue, a modified response, or 204 No Content.)

In another example, if the preview is 1024 bytes and the origin response is 1024 bytes in two chunks, then the encapsulation would appear as follows:

```
200\r\n
<512 bytes of data>\r\n
200\r\n
<512 bytes of data>\r\n
0; ieof\r\n\r\n

<204 or modified response> (100 Continue disallowed due to ieof)
```

If the preview is 1024 bytes and the origin response is 1025 bytes (and the ICAP server responds with 100-continue), then these chunks would appear on the wire:

```
200\r\n
<512 bytes of data>\r\n
200\r\n
<512 bytes of data>\r\n
0\r\n

<100 Continue Message>

1\r\n
<1 byte of data>\r\n
0\r\n\r\n  <no ieof because we are no longer in preview mode>
```

Once the ICAP server receives the eof indicator, it finishes reading the current chunk stream.

Note that when offering a Preview, the ICAP client is committing to temporarily buffer the previewed portion of the message so that it can honor a "204 No Content" response. The remainder of the message is not necessarily buffered; it might be pipelined directly from another source to the ICAP server after a 100-Continue.

## 4.6 "204 No Content" Responses outside of Previews

An ICAP client MAY choose to honor "204 No Content" responses for an entire message. This is the decision of the client because it imposes a burden on the client of buffering the entire message.

An ICAP client MAY include "Allow: 204" in its request headers, indicating that the server MAY reply to the message with a "204 No Content" response if the object does not need modification.

If an ICAP server receives a request that does not have "Allow: 204", it MUST NOT reply with a 204. In this case, an ICAP server MUST return the entire message back to the client, even though it is identical to the message it received.

The ONLY EXCEPTION to this rule is in the case of a message preview, as described in the previous section. If this is the case, an ICAP server can respond with a 204 No Content message in response to a message preview EVEN if the original request did not have the "Allow: 204" header.

## 4.7 ISTag Response Header

The ISTag ("ICAP Service Tag") response-header field provides a way for ICAP servers to send a service-specific "cookie" to ICAP clients that represents a service's current state. It is a 32-byte-maximum alphanumeric string of data (not including the null character) that

may, for example, be a representation of the software version or configuration of a service. An ISTag validates that previous ICAP server responses can still be considered fresh by an ICAP client that may be caching them. If a change on the ICAP server invalidates previous responses, the ICAP server can invalidate portions of the ICAP client's cache by changing its ISTag. The ISTag MUST be included in every ICAP response from an ICAP server.

For example, consider a virus-scanning ICAP service. The ISTag might be a combination of the virus scanner's software version and the release number of its virus signature database. When the database is updated, the ISTag can be changed to invalidate all previous responses that had been certified as "clean" and cached with the old ISTag.

ISTag is similar, but not identical, to the HTTP ETag. While an ETag is a validator for a particular entity (object), an ISTag validates all entities generated by a particular service (URI). A change in the ISTag invalidates all the other entities provided a service with the old ISTag, not just the entity whose response contained the updated ISTag.

The syntax of an ISTag is simply:

```
ISTag = "ISTag: " quoted-string
```

In this document we use the quoted-string definition defined in section 2.2 of [4].

For example:

```
ISTag: "874900-1994-1c02798"
```

### 4.8 Request Modification Mode

In this method, described in Section 3.1, an ICAP client sends an HTTP request to an ICAP server. The ICAP server returns a modified version of the request, an HTTP response, or (if the client indicates it supports 204 responses) an indication that no modification is required.

#### 4.8.1 Request

In REQMOD mode, the ICAP request MUST contain an encapsulated HTTP request. The headers and body (if any) MUST both be encapsulated, except that hop-by-hop headers are not encapsulated.

### 4.8.2 Response

The response from the ICAP server back to the ICAP client may take one of four forms:

- An error indication,
- A 204 indicating that the ICAP client's request requires no adaptation (see Section 4.6 for limitations of this response),
- An encapsulated, adapted version of the ICAP client's request, or
- An encapsulated HTTP error response. Note that Request Modification requests may only be satisfied with HTTP responses in cases when the HTTP response is an error (e.g., 403 Forbidden).

The first line of the response message MUST be a status line as described in Section 4.3.3. If the return code is a 2XX, the ICAP client SHOULD continue its normal execution of the request. If the ICAP client is a surrogate, this may include serving an object from its cache or forwarding the modified request to an origin server. Note it is valid for a 2XX ICAP response to contain an encapsulated HTTP error response, which in turn should be returned to the downstream client by the ICAP client.

For other return codes that indicate an error, the ICAP client MAY (for example) return the error to the downstream client or user, execute the unadapted request as it arrived from the client, or re-try the adaptation again.

The modified request headers, if any, MUST be returned to the ICAP client using appropriate encapsulation as described in Section 4.4.

### 4.8.3 Examples

Consider the following example, in which a surrogate receives a simple GET request from a client. The surrogate, acting as an ICAP client, then forwards this request to an ICAP server for modification. The ICAP server modifies the request headers and sends them back to the ICAP client. Our hypothetical ICAP server will modify several headers and strip the cookie from the original request.

In all of our examples, we include the extra meta-data added to the message due to chunking the encapsulated message body (if any). We assume that end-of-line terminations, and blank lines, are two-byte "CRLF" sequences.

ICAP Request Modification Example 1 - ICAP Request

```
REQMOD icap://icap-server.net/server?arg=87 ICAP/1.0
Host: icap-server.net
Encapsulated: req-hdr=0, null-body=170

GET / HTTP/1.1
Host: www.origin-server.com
Accept: text/html, text/plain
Accept-Encoding: compress
Cookie: ff39fk3jur@4ii0e02i
If-None-Match: "xyzzy", "r2d2xxxx"
```

ICAP Request Modification Example 1 - ICAP Response

```
ICAP/1.0 200 OK
Date: Mon, 10 Jan 2000  09:55:21 GMT
Server: ICAP-Server-Software/1.0
Connection: close
ISTag: "W3E4R7U9-L2E4-2"
Encapsulated: req-hdr=0, null-body=231

GET /modified-path HTTP/1.1
Host: www.origin-server.com
Via: 1.0 icap-server.net (ICAP Example ReqMod Service 1.1)
Accept: text/html, text/plain, image/gif
Accept-Encoding: gzip, compress
If-None-Match: "xyzzy", "r2d2xxxx"
```

The second example is similar to the first, except that the request being modified in this case is a POST instead of a GET. Note that the encapsulated Content-Length argument has been modified to reflect the modified body of the POST message. The outer ICAP message does not need a Content-Length header because it uses chunking (not shown).

In this second example, the Encapsulated header shows the division between the forwarded header and forwarded body, for both the request and the response.

ICAP Request Modification Example 2 - ICAP Request

```
REQMOD icap://icap-server.net/server?arg=87 ICAP/1.0
Host: icap-server.net
Encapsulated: req-hdr=0, req-body=147
```

```
POST /origin-resource/form.pl HTTP/1.1
Host: www.origin-server.com
Accept: text/html, text/plain
Accept-Encoding: compress
Pragma: no-cache

1e
I am posting this information.
0
```

---

ICAP Request Modification Example 2 - ICAP Response

---

```
ICAP/1.0 200 OK
Date: Mon, 10 Jan 2000  09:55:21 GMT
Server: ICAP-Server-Software/1.0
Connection: close
ISTag: "W3E4R7U9-L2E4-2"
Encapsulated: req-hdr=0, req-body=244

POST /origin-resource/form.pl HTTP/1.1
Host: www.origin-server.com
Via: 1.0 icap-server.net (ICAP Example ReqMod Service 1.1)
Accept: text/html, text/plain, image/gif
Accept-Encoding: gzip, compress
Pragma: no-cache
Content-Length: 45

2d
I am posting this information.  ICAP powered!
0
```

---

Finally, this third example shows an ICAP server returning an error response when it receives a Request Modification request.

ICAP Request Modification Example 3 - ICAP Request

---

```
REQMOD icap://icap-server.net/content-filter ICAP/1.0
Host: icap-server.net
Encapsulated: req-hdr=0, null-body=119

GET /naughty-content HTTP/1.1
Host: www.naughty-site.com
Accept: text/html, text/plain
Accept-Encoding: compress
```

---

ICAP Request Modification Example 3 - ICAP Response

```
-----------------------------------------------------------------
ICAP/1.0 200 OK
Date: Mon, 10 Jan 2000  09:55:21 GMT
Server: ICAP-Server-Software/1.0
Connection: close
ISTag: "W3E4R7U9-L2E4-2"
Encapsulated: res-hdr=0, res-body=213

HTTP/1.1 403 Forbidden
Date: Wed, 08 Nov 2000 16:02:10 GMT
Server: Apache/1.3.12 (Unix)
Last-Modified: Thu, 02 Nov 2000 13:51:37 GMT
ETag: "63600-1989-3a017169"
Content-Length: 58
Content-Type: text/html

3a
Sorry, you are not allowed to access that naughty content.
0

-----------------------------------------------------------------
```

### 4.9 Response Modification Mode

In this method, described in Section 3.2, an ICAP client sends an origin server's HTTP response to an ICAP server, and (if available) the original client request that caused that response. Similar to Request Modification method, the response from the ICAP server can be an adapted HTTP response, an error, or a 204 response code indicating that no adaptation is required.

#### 4.9.1 Request

Using encapsulation described in Section 4.4, the header and body of the HTTP response to be modified MUST be included in the ICAP body. If available, the header of the original client request SHOULD also be included. As with the other method, the hop-by-hop headers of the encapsulated messages MUST NOT be forwarded. The Encapsulated header MUST indicate the byte-offsets of the beginning of each of these four parts.

#### 4.9.2 Response

The response from the ICAP server looks just like a reply in the Request Modification method (Section 4.8); that is,

- An error indication,

- An encapsulated and potentially modified HTTP response header and response body, or

- An HTTP response 204 indicating that the ICAP client's request requires no adaptation.

The first line of the response message MUST be a status line as described in Section 4.3.3. If the return code is a 2XX, the ICAP client SHOULD continue its normal execution of the response. The ICAP client MAY re-examine the headers in the response's message headers in order to make further decisions about the response (e.g., its cachability).

For other return codes that indicate an error, the ICAP client SHOULD NOT return these directly to downstream client, since these errors only make sense in the ICAP client/server transaction.

The modified response headers, if any, MUST be returned to the ICAP client using appropriate encapsulation as described in Section 4.4.

### 4.9.3 Examples

In Example 4, an ICAP client is requesting modification of an entity that was returned as a result of a client GET. The original client GET was to an origin server at "www.origin-server.com"; the ICAP server is at "icap.example.org".

ICAP Response Modification Example 4 - ICAP Request

```
RESPMOD icap://icap.example.org/satisf ICAP/1.0
Host: icap.example.org
Encapsulated: req-hdr=0, res-hdr=137, res-body=296

GET /origin-resource HTTP/1.1
Host: www.origin-server.com
Accept: text/html, text/plain, image/gif
Accept-Encoding: gzip, compress

HTTP/1.1 200 OK
Date: Mon, 10 Jan 2000 09:52:22 GMT
Server: Apache/1.3.6 (Unix)
ETag: "63840-1ab7-378d415b"
Content-Type: text/html
Content-Length: 51
```

```
33
This is data that was returned by an origin server.
0

-----------------------------------------------------------------

ICAP Response Modification Example 4 - ICAP Response
-----------------------------------------------------------------
ICAP/1.0 200 OK
Date: Mon, 10 Jan 2000  09:55:21 GMT
Server: ICAP-Server-Software/1.0
Connection: close
ISTag: "W3E4R7U9-L2E4-2"
Encapsulated: res-hdr=0, res-body=222

HTTP/1.1 200 OK
Date: Mon, 10 Jan 2000  09:55:21 GMT
Via: 1.0 icap.example.org (ICAP Example RespMod Service 1.1)
Server: Apache/1.3.6 (Unix)
ETag: "63840-1ab7-378d415b"
Content-Type: text/html
Content-Length: 92

5c
This is data that was returned by an origin server, but with
value added by an ICAP server.
0

-----------------------------------------------------------------
```

### 4.10 OPTIONS Method

The ICAP "OPTIONS" method is used by the ICAP client to retrieve configuration information from the ICAP server. In this method, the ICAP client sends a request addressed to a specific ICAP resource and receives back a response with options that are specific to the service named by the URI. All OPTIONS requests MAY also return options that are global to the server (i.e., apply to all services).

#### 4.10.1 OPTIONS Request

The OPTIONS method consists of a request-line, as described in Section 4.3.2, such as the following example:

```
OPTIONS icap://icap.server.net/sample-service ICAP/1.0 User-Agent:
ICAP-client-XYZ/1.001
```

Other headers are also allowed as described in Section 4.3.1 and Section 4.3.2 (for example, Host).

### 4.10.2 OPTIONS Response

The OPTIONS response consists of a status line as described in section 4.3.3 followed by a series of header field names-value pairs optionally followed by an opt-body. Multiple values in the value field MUST be separated by commas. If an opt-body is present in the OPTIONS response, the Opt-body-type header describes the format of the opt-body.

The OPTIONS headers supported in this version of the protocol are:

-- Methods:

The method that is supported by this service. This header MUST be included in the OPTIONS response. The OPTIONS method MUST NOT be in the Methods' list since it MUST be supported by all the ICAP server implementations. Each service should have a distinct URI and support only one method in addition to OPTIONS (see Section 6.4).

For example:
Methods: RESPMOD

-- Service:

A text description of the vendor and product name. This header MAY be included in the OPTIONS response.

For example:
Service: XYZ Technology Server 1.0

-- ISTag:

See section 4.7 for details. This header MUST be included in the OPTIONS response.

For example:
ISTag: "5BDEEEA9-12E4-2"

-- Encapsulated:

This header MUST be included in the OPTIONS response; see Section 4.4.

For example:
Encapsulated: opt-body=0

-- Opt-body-type:

A token identifying the format of the opt-body. (Valid opt-body types are not defined by ICAP.) This header MUST be included in the OPTIONS response ONLY if an opt-body type is present.

For example:
Opt-body-type: XML-Policy-Table-1.0

-- Max-Connections:

The maximum number of ICAP connections the server is able to support. This header MAY be included in the OPTIONS response.

For example:
Max-Connections: 1500

-- Options-TTL:

The time (in seconds) for which this OPTIONS response is valid. If none is specified, the OPTIONS response does not expire. This header MAY be included in the OPTIONS response. The ICAP client MAY reissue an OPTIONS request once the Options-TTL expires.

For example:
Options-TTL: 3600

-- Date:

The server’s clock, specified as an RFC 1123 compliant date/time string. This header MAY be included in the OPTIONS response.

For example:
Date: Fri, 15 Jun 2001 04:33:55 GMT

-- Service-ID:

A short label identifying the ICAP service. It MAY be used in attribute header names. This header MAY be included in the OPTIONS response.

For example:
Service-ID: xyztech

-- Allow:

A directive declaring a list of optional ICAP features that this server has implemented. This header MAY be included in the OPTIONS response. In this document we define the value "204" to indicate that the ICAP server supports a 204 response.

For example:
Allow: 204

-- Preview:

The number of bytes to be sent by the ICAP client during a preview. This header MAY be included in the OPTIONS response.

For example:
Preview: 1024

-- Transfer-Preview:

A list of file extensions that should be previewed to the ICAP server before sending them in their entirety. This header MAY be included in the OPTIONS response. Multiple file extensions values should be separated by commas. The wildcard value "*" specifies the default behavior for all the file extensions not specified in any other Transfer-* header (see below).

For example:
Transfer-Preview: *

-- Transfer-Ignore:

A list of file extensions that should NOT be sent to the ICAP server. This header MAY be included in the OPTIONS response. Multiple file extensions should be separated by commas.

For example:
Transfer-Ignore: html

-- Transfer-Complete:

A list of file extensions that should be sent in their entirety (without preview) to the ICAP server. This header MAY be included in the OPTIONS response. Multiple file extensions values should be separated by commas.

For example:
Transfer-Complete: asp, bat, exe, com, ole

Note: If any of Transfer-* are sent, exactly one of them MUST contain the wildcard value "*" to specify the default. If no Transfer-* are sent, all responses will be sent in their entirety (without Preview).

### 4.10.3 OPTIONS Examples

In example 5, an ICAP Client sends an OPTIONS Request to an ICAP Service named icap.server.net/sample-service in order to get configuration information for the service provided.

ICAP OPTIONS Example 5 - ICAP OPTIONS Request

```
-------------------------------------------------------------------
OPTIONS icap://icap.server.net/sample-service ICAP/1.0
Host: icap.server.net
User-Agent: BazookaDotCom-ICAP-Client-Library/2.3

-------------------------------------------------------------------
```

ICAP OPTIONS Example 5 - ICAP OPTIONS Response

```
-------------------------------------------------------------------
ICAP/1.0 200 OK
Date: Mon, 10 Jan 2000  09:55:21 GMT
Methods: RESPMOD
Service: FOO Tech Server 1.0
ISTag: "W3E4R7U9-L2E4-2"
Encapsulated: null-body=0
Max-Connections: 1000
Options-TTL: 7200
Allow: 204
Preview: 2048
Transfer-Complete: asp, bat, exe, com
Transfer-Ignore: html
Transfer-Preview: *

-------------------------------------------------------------------
```

## 5. Caching

ICAP servers' responses MAY be cached by ICAP clients, just as any other surrogate might cache HTTP responses. Similar to HTTP, ICAP clients MAY always store a successful response (see sections 4.8.2 and 4.9.2) as a cache entry, and MAY return it without validation if it is fresh. ICAP servers use the caching directives described in HTTP/1.1 [4].

In Request Modification mode, the ICAP server MAY include caching directives in the ICAP header section of the ICAP response (NOT in the encapsulated HTTP request of the ICAP message body). In Response

Modification mode, the ICAP server MAY add or modify the HTTP caching directives located in the encapsulated HTTP response (NOT in the ICAP header section). Consequently, the ICAP client SHOULD look for caching directives in the ICAP headers in case of REQMOD, and in the encapsulated HTTP response in case of RESPMOD.

In cases where an ICAP server returns a modified version of an object created by an origin server, such as in Response Modification mode, the expiration of the ICAP-modified object MUST NOT be longer than that of the origin object. In other words, ICAP servers MUST NOT extend the lifetime of origin server objects, but MAY shorten it.

In cases where the ICAP server is the authoritative source of an ICAP response, such as in Request Modification mode, the ICAP server is not restricted in its expiration policy.

Note that the ISTag response-header may also be used to providing caching hints to clients; see Section 4.7.

# 6. Implementation Notes

## 6.1 Vectoring Points

The definition of the ICAP protocol itself only describes two different adaptation channels: modification (and satisfaction) of requests, and modifications of replies. However, an ICAP client implementation is likely to actually distinguish among four different classes of adaptation:

1. Adaptation of client requests. This is adaptation done every time a request arrives from a client. This is adaptation done when a request is "on its way into the cache". Factors such as the state of the objects currently cached will determine whether or not this request actually gets forwarded to an origin server (instead of, say, getting served off the cache’s disk). An example of this type of adaptation would be special access control or authentication services that must be performed on a per-client basis.

2. Adaptation of requests on their way to an origin server. Although this type of adaptation is also an adaptation of requests similar to (1), it describes requests that are "on their way out of the cache"; i.e., if a request actually requires that an origin server be contacted. These adaptation requests are not necessarily specific to particular clients. An example would be addition of "Accept:" headers for special devices; these adaptations can potentially apply to many clients.

3. Adaptations of responses coming from an origin server. This is the adaptation of an object "on its way into the cache". In other words, this is adaptation that a surrogate might want to perform on an object before caching it. The adapted object may subsequently served to many clients. An example of this type of adaptation is virus checking: a surrogate will want to check an incoming origin reply for viruses once, before allowing it into the cache -- not every time the cached object is served to a client.

   Adaptation of responses coming from the surrogate, heading back to the client. Although this type of adaptation, like (3), is the adaptation of a response, it is client-specific. Client reply adaptation is adaptation that is required every time an object is served to a client, even if all the replies come from the same cached object off of disk. Ad insertion is a common form of this kind of adaptation; e.g., if a popular (cached) object that rarely changes needs a different ad inserted into it every time it is served off disk to a client. Note that the relationship between adaptations of type (3) and (4) is analogous to the relationship between types (2) and (1).

Although the distinction among these four adaptation points is critical for ICAP client implementations, the distinction is not significant for the ICAP protocol itself. From the point of view of an ICAP server, a request is a request -- the ICAP server doesn't care what policy led the ICAP client to generate the request. We therefore did not make these four channels explicit in ICAP for simplicity.

## 6.2 Application Level Errors

Section 4 described "on the wire" protocol errors that MUST be standardized across implementations to ensure interoperability. In this section, we describe errors that are communicated between ICAP software and the clients and servers on which they are implemented. Although such errors are implementation dependent and do not necessarily need to be standardized because they are "within the box", they are presented here as advice to future implementors based on past implementation experience.

| Error name | Value |
|---|---|
| ICAP_CANT_CONNECT | 1000 |
| ICAP_SERVER_RESPONSE_CLOSE | 1001 |
| ICAP_SERVER_RESPONSE_RESET | 1002 |
| ICAP_SERVER_UNKNOWN_CODE | 1003 |
| ICAP_SERVER_UNEXPECTED_CLOSE_204 | 1004 |
| ICAP_SERVER_UNEXPECTED_CLOSE | 1005 |

1000 ICAP_CANT_CONNECT:
"Cannot connect to ICAP server".

The ICAP server is not connected on the socket. Maybe the ICAP server is dead or it is not connected on the socket.

1001 ICAP_SERVER_RESPONSE_CLOSE:
"ICAP Server closed connection while reading response".

The ICAP server TCP-shutdowns the connection before the ICAP client can send all the body data.

1002 ICAP_SERVER_RESPONSE_RESET:
"ICAP Server reset connection while reading response".

The ICAP server TCP-reset the connection before the ICAP client can send all the body data.

1003 ICAP_SERVER_UNKNOWN_CODE:
"ICAP Server sent unknown response code".

An unknown ICAP response code (see Section 4.x) was received by the ICAP client.

1004 ICAP_SERVER_UNEXPECTED_CLOSE_204:
"ICAP Server closed connection on 204 without 'Connection: close' header".

An ICAP server MUST send the "Connection: close" header if intends to close after the current transaction.

1005 ICAP_SERVER_UNEXPECTED_CLOSE:
"ICAP Server closed connection as ICAP client wrote body preview".

### 6.3 Use of Chunked Transfer-Encoding

For simplicity, ICAP messages MUST use the "chunked" transfer-encoding within the encapsulated body section as defined in HTTP/1.1 [4]. This requires that ICAP client implementations convert incoming objects "on the fly" to chunked from whatever transfer-encoding on which they arrive. However, the transformation is simple:

- For objects arriving using "Content-Length" headers, one big chunk can be created of the same size as indicated in the Content-Length header.

- For objects arriving using a TCP close to signal the end of the object, each incoming group of bytes read from the OS can be converted into a chunk (by writing the length of the bytes read, followed by the bytes themselves)

- For objects arriving using chunked encoding, they can be retransmitted as is (without re-chunking).

### 6.4 Distinct URIs for Distinct Services

ICAP servers SHOULD assign unique URIs to each service they provide, even if such services might theoretically be differentiated based on their method. In other words, a REQMOD and RESPMOD service should never have the same URI, even if they do something that is conceptually the same.

This situation in ICAP is similar to that found in HTTP where it might, in theory, be possible to perform a GET or a POST to the same URI and expect two different results. This kind of overloading of URIs only causes confusion and should be avoided.

## 7. Security Considerations

### 7.1 Authentication

Authentication in ICAP is very similar to proxy authentication in HTTP as specified in RFC 2617. Specifically, the following rules apply:

- WWW-Authenticate challenges and responses are for end-to-end authentication between a client (user) and an origin server. As any proxy, ICAP clients and ICAP servers MUST forward these headers without modification.

- If authentication is required between an ICAP client and ICAP server, hop-by-hop Proxy Authentication as described in RFC 2617 MUST be used.

There are potential applications where a user (as opposed to ICAP client) might have rights to access an ICAP service. In this version of the protocol, we assume that ICAP clients and ICAP servers are under the same administrative domain, and contained in a single trust domain. Therefore, in these cases, we assume that it is sufficient for users to authenticate themselves to the ICAP client (which is a surrogate from the point of view from the user). This type of authentication will also be Proxy Authentication as described in RFC 2617.

This standard explicitly excludes any method for a user to authenticate directly to an ICAP server; the ICAP client MUST be involved as described above.

## 7.2 Encryption

Users of ICAP should note well that ICAP messages are not encrypted for transit by default. In the absence of some other form of encryption at the link or network layers, eavesdroppers may be able to record the unencrypted transactions between ICAP clients and servers. As described in Section 4.3.1, the Upgrade header MAY be used to negotiate transport-layer security for an ICAP connection [5].

Note also that end-to-end encryption between a client and origin server is likely to preclude the use of value-added services by intermediaries such as surrogates. An ICAP server that is unable to decrypt a client’s messages will, of course, be unable to perform any transformations on it.

## 7.3 Service Validation

Normal HTTP surrogates, when operating correctly, should not affect the end-to-end semantics of messages that pass through them. This forms a well-defined criterion to validate that a surrogate is working correctly: a message should look the same before the surrogate as it does after the surrogate.

In contrast, ICAP is meant to cause changes in the semantics of messages on their way from origin servers to users. The criteria for a correctly operating surrogate are no longer as easy to define. This will make validation of ICAP services significantly more difficult. Incorrect adaptations may lead to security vulnerabilities that were not present in the unadapted content.

## 8. Motivations and Design Alternatives

This section describes some of our design decisions in more detail, and describes the ideas and motivations behind them. This section does not define protocol requirements, but hopefully sheds light on the requirements defined in previous sections. Nothing in this section carries the "force of law" or is part of the formal protocol specification.

In general, our guiding principle was to make ICAP the simplest possible protocol that would do the job, and no simpler. Some features were rejected where alternative (non-protocol-based) solutions could be found. In addition, we have intentionally left a number of issues at the discretion of the implementor, where we believe that doing so does not compromise interoperability.

### 8.1 To Be HTTP, or Not To Be

ICAP was initially designed as an application-layer protocol built to run on top of HTTP. This was desirable for a number of reasons. HTTP is well-understood in the community and has enjoyed significant investments in software infrastructure (clients, servers, parsers, etc.). Our initial designs focused on leveraging that existing work; we hoped that it would be possible to implement ICAP services simply, using CGI scripts run by existing web servers.

However, the devil (as always) proved to be in the details. Certain features that we considered important were impossible to implement with HTTP. For example, ICAP clients can stop and wait for a "100 Continue" message in the midst of a message-body; HTTP clients may only wait between the header and body. In addition, certain transformations of HTTP messages by surrogates are legal (and harmless for HTTP), but caused problems with ICAP's "header-in-header" encapsulation and other features.

Ultimately, we decided that the tangle of workarounds required to fit ICAP into HTTP was more complex and confusing than moving away from HTTP and defining a new (but similar) protocol.

### 8.2 Mandatory Use of Chunking

Chunking is mandatory in ICAP encapsulated bodies for three reasons. First, efficiency is important, and the chunked encoding allows both the client and server to keep the transport-layer connection open for later reuse. Second, ICAP servers (and their developers) should be encouraged to produce "incremental" responses where possible, to reduce the latency perceived by users. Chunked encoding is the only way to support this type of implementation. Finally, by

standardizing on a single encapsulation mechanism, we avoid the complexity that would be required in client and server software to support multiple mechanisms. This simplifies ICAP, particularly in the "body preview" feature described in Section 4.5.

While chunking of encapsulated bodies is mandatory, encapsulated headers are not chunked. There are two reasons for this decision. First, in cases where a chunked HTTP message body is being encapsulated in an ICAP message, the ICAP client (HTTP server) can copy it directly from the HTTP client to the ICAP server without un-chunking and then re-chunking it. Second, many header-parser implementations have difficulty dealing with headers that come in multiple chunks. Earlier drafts of this document mandated that a chunk boundary not come within a header. For clarity, chunking of encapsulated headers has simply been disallowed.

### 8.3 Use of the null-body directive in the Encapsulated header

There is a disadvantage to not using the chunked transfer-encoding for encapsulated header part of an ICAP message. Specifically, parsers do not know in advance how much header data is coming (e.g., for buffer allocation). ICAP does not allow chunking in the header part for reasons described in Section 8.2. To compensate, the "null-body" directive allows the final header’s length to be determined, despite it not being chunked.

## 9. References

[1] Berners-Lee, T., Fielding, R. and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax and Semantics", RFC 2396, August 1998.

[2] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[3] Resnick, P., "Internet Message Format", RFC 2822, April 2001.

[4] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[5] Khare, R. and S. Lawrence, "Upgrading to TLS Within HTTP/1.1", RFC 2817, May 2000.

## 10. Contributors

ICAP is based on an original idea by John Martin and Peter Danzig. Many individuals and organizations have contributed to the development of ICAP, including the following contributors (past and present):

Lee Duggs
Network Appliance, Inc.
495 East Java Dr.
Sunnyvale, CA 94089 USA

Phone: (408) 822-6000
EMail: lee.duggs@netapp.com

Paul Eastham
Network Appliance, Inc.
495 East Java Dr.
Sunnyvale, CA 94089 USA

Phone: (408) 822-6000
EMail: eastham@netapp.com

Debbie Futcher
Network Appliance, Inc.
495 East Java Dr.
Sunnyvale, CA 94089 USA

Phone: (408) 822-6000
EMail: deborah.futcher@netapp.com

Don Gillies
Network Appliance, Inc.
495 East Java Dr.
Sunnyvale, CA 94089 USA

Phone: (408) 822-6000
EMail: gillies@netapp.com

Steven La
Network Appliance, Inc.
495 East Java Dr.
Sunnyvale, CA 94089 USA

Phone: (408) 822-6000
EMail: steven.la@netapp.com

John Martin
Network Appliance, Inc.
495 East Java Dr.
Sunnyvale, CA 94089 USA

Phone: (408) 822-6000
EMail: jmartin@netapp.com

Jeff Merrick
Network Appliance, Inc.
495 East Java Dr.
Sunnyvale, CA 94089 USA

Phone: (408) 822-6000
EMail: jeffrey.merrick@netapp.com

John Schuster
Network Appliance, Inc.
495 East Java Dr.
Sunnyvale, CA 94089 USA

Phone: (408) 822-6000
EMail: john.schuster@netapp.com

Edward Sharp
Network Appliance, Inc.
495 East Java Dr.
Sunnyvale, CA 94089 USA

Phone: (408) 822-6000
EMail: edward.sharp@netapp.com

Peter Danzig
Akamai Technologies
1400 Fashion Island Blvd
San Mateo, CA 94404 USA

Phone: (650) 372-5757
EMail: danzig@akamai.com

Mark Nottingham
Akamai Technologies
1400 Fashion Island Blvd
San Mateo, CA 94404 USA

Phone: (650) 372-5757
EMail: mnot@akamai.com

Nitin Sharma
Akamai Technologies
1400 Fashion Island Blvd
San Mateo, CA 94404 USA

Phone: (650) 372-5757
EMail: nitin@akamai.com

Hilarie Orman
Novell, Inc.
122 East 1700 South
Provo, UT 84606 USA

Phone: (801) 861-7021
EMail: horman@novell.com

Craig Blitz
Novell, Inc.
122 East 1700 South
Provo, UT 84606 USA

Phone: (801) 861-7021
EMail: cblitz@novell.com

Gary Tomlinson
Novell, Inc.
122 East 1700 South
Provo, UT 84606 USA

Phone: (801) 861-7021
EMail: garyt@novell.com

Andre Beck
Bell Laboratories / Lucent Technologies
101 Crawfords Corner Road
Holmdel, New Jersey 07733-3030

Phone: (732) 332-5983
EMail: abeck@bell-labs.com

Markus Hofmann
Bell Laboratories / Lucent Technologies
101 Crawfords Corner Road
Holmdel, New Jersey 07733-3030

Phone: (732) 332-5983
EMail: hofmann@bell-labs.com

David Bryant
CacheFlow, Inc.
650 Almanor Avenue
Sunnyvale, California 94086

Phone: (888) 462-3568
EMail: david.bryant@cacheflow.com

## Appendix A BNF Grammar for ICAP Messages

This grammar is specified in terms of the augmented Backus-Naur Form (BNF) similar to that used by the HTTP/1.1 specification (See Section 2.1 of [4]). Implementors will need to be familiar with the notation in order to understand this specification.

Many header values (where noted) have exactly the same grammar and semantics as in HTTP/1.1. We do not reproduce those grammars here.

```
ICAP-Version = "ICAP/1.0"

ICAP-Message = Request | Response

Request      = Request-Line
               *(Request-Header CRLF)
               CRLF
               [ Request-Body ]

Request-Line = Method SP ICAP_URI SP ICAP-Version CRLF

Method       = "REQMOD"         ; Section 4.8
             | "RESPMOD"        ; Section 4.9
             | "OPTIONS"        ; Section 4.10
             | Extension-Method ; Section 4.3.2

Extension-Method = token

ICAP_URI = Scheme ":" Net_Path [ "?" Query ]  ; Section 4.2

Scheme      = "icap"

Net_Path    = "//" Authority [ Abs_Path ]

Authority   = [ userinfo "@" ] host [ ":" port ]


Request-Header     = Request-Fields ":" [ Generic-Field-Value ]

Request-Fields     = Request-Field-Name
                   | Common-Field-Name

; Header fields specific to requests
Request-Field-Name = "Authorization"   ; Section 4.3.2
                   | "Allow"           ; Section 4.3.2
                   | "From"            ; Section 4.3.2
                   | "Host"            ; Section 4.3.2
                   | "Referer"         ; Section 4.3.2
```

```
                    | "User-Agent"      ; Section 4.3.2
                    | "Preview"         ; Section 4.5

   ; Header fields common to both requests and responses
   Common-Field-Name  = "Cache-Control"   ; Section 4.3.1
                    | "Connection"      ; Section 4.3.1
                    | "Date"            ; Section 4.3.1
                    | "Expires"         ; Section 4.3.1
                    | "Pragma"          ; Section 4.3.1
                    | "Trailer"         ; Section 4.3.1
                    | "Upgrade"         ; Section 4.3.1
                    | "Encapsulated"    ; Section 4.4
                    | Extension-Field-Name   ; Section 4.3

   Extension-Field-Name  = "X-" token

   Generic-Field-Value   = *( Generic-Field-Content | LWS )
   Generic-Field-Content = <the OCTETs making up the field-value
                            and consisting of either *TEXT or
                            combinations of token, separators,
                            and quoted-string>

   Request-Body = *OCTET   ; See Sections 4.4 and 4.5 for semantics

   Response    = Status-Line
                 *(Response-Header CRLF)
                 CRLF
                 [ Response-Body ]

   Status-Line = ICAP-Version SP Status-Code SP Reason-Phrase CRLF

   Status-Code = "100"  ; Section 4.5
               | "101"  ; Section 10.1.2 of [4]
               | "200"  ; Section 10.2.1 of [4]
               | "201"  ; Section 10.2.2 of [4]
               | "202"  ; Section 10.2.3 of [4]
               | "203"  ; Section 10.2.4 of [4]
               | "204"  ; Section 4.6
               | "205"  ; Section 10.2.6 of [4]
               | "206"  ; Section 10.2.7 of [4]
               | "300"  ; Section 10.3.1 of [4]
               | "301"  ; Section 10.3.2 of [4]
               | "302"  ; Section 10.3.3 of [4]
               | "303"  ; Section 10.3.4 of [4]
               | "304"  ; Section 10.3.5 of [4]
               | "305"  ; Section 10.3.6 of [4]
               | "306"  ; Section 10.3.7 of [4]
               | "307"  ; Section 10.3.8 of [4]
```

```
                | "400"  ; Section 4.3.3
                | "401"  ; Section 10.4.2 of [4]
                | "402"  ; Section 10.4.3 of [4]
                | "403"  ; Section 10.4.4 of [4]
                | "404"  ; Section 4.3.3
                | "405"  ; Section 4.3.3
                | "406"  ; Section 10.4.7 of [4]
                | "407"  ; Section 10.4.8 of [4]
                | "408"  ; Section 4.3.3
                | "409"  ; Section 10.4.10 of [4]
                | "410"  ; Section 10.4.11 of [4]
                | "411"  ; Section 10.4.12 of [4]
                | "412"  ; Section 10.4.13 of [4]
                | "413"  ; Section 10.4.14 of [4]
                | "414"  ; Section 10.4.15 of [4]
                | "415"  ; Section 10.4.16 of [4]
                | "416"  ; Section 10.4.17 of [4]
                | "417"  ; Section 10.4.18 of [4]
                | "500"  ; Section 4.3.3
                | "501"  ; Section 4.3.3
                | "502"  ; Section 4.3.3
                | "503"  ; Section 4.3.3
                | "504"  ; Section 10.5.5 of [4]
                | "505"  ; Section 4.3.3
                | Extension-Code

   Extension-Code = 3DIGIT

   Reason-Phrase = *<TEXT, excluding CR, LF>

   Response-Header     = Response-Fields ":" [ Generic-Field-Value ]

   Response-Fields     = Response-Field-Name
                       | Common-Field-Name

   Response-Field-Name = "Server"         ; Section 4.3.3
                       | "ISTag"          ; Section 4.7

   Response-Body = *OCTET  ; See Sections 4.4 and 4.5 for semantics
```

## Authors' Addresses

Jeremy Elson
University of California Los Angeles
Department of Computer Science
3440 Boelter Hall
Los Angeles CA 90095

Phone: (310) 206-3925
EMail: jelson@cs.ucla.edu

Alberto Cerpa
University of California Los Angeles
Department of Computer Science
3440 Boelter Hall
Los Angeles CA 90095

Phone: (310) 206-3925
EMail: cerpa@cs.ucla.edu

ICAP discussion currently takes place at
icap-discussions@yahoogroups.com.
For more information, see
http://groups.yahoo.com/group/icap-discussions/.

## Full Copyright Statement



## Acknowledgement

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