ESPRIT PROJECT EP28425

Broadband Integrated Satellite Network Traffic Evaluations

Deliverable 1.3

Generation of User Profiles

Status / Version : **DELIVERABLE / FINAL**
Date : **September 30, 1999**
Distribution : **Public**
Code : **BISANTE/DEL13**
Author (s) : **G. Kotsis (Ed)**

Abstract: The goal of this deliverable is to present the workload generation component of the BISANTE layered approach for network load modelling which is based on user behaviour profiles (UBP). An UBP consists of a set of models, at least one per layer, and associated parameters and values, describing the transition from user behaviour to actual network load. The workload generator activates instances of such profiles at simulation time.

The BISANTE Consortium consists of :

Thomson-CSF Communications	Partner	France
Netway	Partner	Austria
Solinet	Partner	Germany
University of Vienna	Associated Partner	Austria
University of Surrey	Associated Partner	United Kingdom
Institut National des Télécommunications (INT)	Associated Partner	France

TABLE OF CONTENTS

1. OUTLINE

The **goals** of work package one, Application Characteristics and Users´ Behaviour Modelling, were threefold:

1. identify relevant classes of applications,
2. get an understanding of user behaviour, and
3. develop parametrisable user profiles.

While the contributions to the first two goals were reported in DEL1.1 and DEL1.2, this deliverable covers the third aspect, namely the generation of user profiles. In that sense, this deliverable also serves as the input for the transition from work package one to work package three, Simulation Prototype, as it sets the basis for the specification and implementation of the workload generator.

Recalling the **layered methodology** elaborated in DEL1.1 and DEL1.2, we defined a user profile as a set of models and associated parameter/value pairs covering all layers in the hierarchy. The analyst simply chooses the desired mix of user profiles and attaches them to the appropriate nodes in the network scenario. Upon simulation time, the models at the top layer, the workload generator layer, will start creating user sessions according to the parameters specified in the user behaviour profile. The model parameters define the type of session that will be activated and the interarrival time of sessions.

While the basic mechanism of passing on signals between layers to activate models at lower layers and to return feedback on system behaviour to higher layers has already been described in DEL1.2, this deliverable will discuss the steps necessary to create such **user behaviour profiles** .

In **Section 2,** we will give a general overview of the types of models that can be used at the workload generator layer, the session layer, the application layer, the command layer, and the service layer. We will describes the process of combining application characteristics and user behaviour appropriate models.

In **Section 3**, for the services, that will be specified and implemented in workpackage three, a more detailed discussion is given.

To illustrate the generation of user profiles, we have elaborated on the four case studies. The results are reported in an additional deliverable, kept consortium confidential due to sensitive data contained in the models, DEL1.3-CS.

2. THE MODELS AT LAYERS 1 TO 5

The basic functionality of a model in the BISANTE framework is to generate events of a certain type at a certain rate. For each layer, such a generic model template exists, which has two parameters: a probability for choosing the type of event, and the interarrival time of events.

Recalling from DEL1.2, we note, that in addition to the generic model, also a dummy model is defined for each layer, which simply passes on signals from ist higher layer to ist lower layer.

The generic model can be extended in the following ways:

1. Information on the current state of the system can be added and this information can influence the parameters (cf. First order Markov chains). For example, in a web modelling case study, a user might be in state “information browsing”, represented by a high frequency of submitting http-requests, or in the state of “information reading”, represented by larger think times.
2. The parameter values may not only depend on the current state, but on the state history (cf. Higher order Markov chains).
3. The parameters depend on the observed/simulated system performance reported to the layer as feedback from lower layers. Consider for example a video-conference, where the user reduces the window size of the video if the frame rate decreases to an unsatisfying level.

The actual type of model chosen always depends on the system under study, so it is impossible to give a complete discussion of all possible models. We believe – and will try to verify this claim in the implementation and experiments – that the hierarchical approach already contributes to the representativeness of the workload model and that therefor the models at each layer can be of simpler nature rather than if one would have to model the whole workload generation process within a single model.

In the following sections, we briefly discuss the generic model instances and possible extensions at each layer.

2.1. WORKLOAD GENERATOR

The **Workload generator** module at the top level is responsible for generating the user sessions. Parameters for this model are the type of session created and the interarrival rate of sessions. Both parameters can either be obtained from real measurements, but can also be chosen to mimic hypothetical load to test the system performance under stress conditions.

The generic template has to be adapted by setting those parameters according to the needs of the study. Models can also be more complex in that the parameter values depend on the system behaviour as observed during the simulation. Thus, the BISANTE approach allows for the inclusion of feedback.

Recalling the survey of modelling approaches given in DEL1.1, any type of user behaviour model, which have been discussed in Section 4.1, can be applied at that level. Examples include user behaviour graphs, which control the behaviour of users according to the state the model is currently in, or Petri nets, where the flow of tokens represents the activity of users.

Another crucial task at the top level, is to identify the types of sessions that have to be modelled. We propose (and apply in the case studies) clustering techniques to identify and group similar sessions. Similar approaches have been discussed in DEL1.1 in section 4.2 (Behaviour Based User Classification).

2.2. SESSION LAYER

At the **session layer**, the generic template creates/starts applications. Again, type and arrival rates are the model parameters, which can be either static or dynamic (changing over time). The set of applications that can potentially be started is defined by the type of session, the interarrival times may depend on the user behaviour and on the system feedback as reported to the session layer in terms of signals from the lower layer (application layer).

Again, any model that is able to represent the concept of states, the transition between states, and the triggering of events based on the state (history) can be applied at the session layer. Therefore, Timed Petri nets are a possible powerful, but rather complex choice. Depending on the chosen system under study and on the questions that have to be answered by the simulation, more simplified state-based models might be sufficient. In the case studies, we will implement and compare several types of models, including higher-order markov chains.

2.3. APPLICATION LAYER

At the **application layer**, a sequence of commands and (user) think times is created. The generic model controls this sequence. The type of commands available is determined by the application chosen at the higher level. User thinks times can be a function of the observed (simulated) system performance.

With respect to the types of models that can be applied at this layer, the same argument given for the session layer holds.

2.4. COMMAND LAYER

The **command layer** is responsible for activating services at the next lower layer. Parameters are the type of service (typically, there will be a 1:1 relation between commands and services, but exceptions are possible), and the frequency of starting a service.

With respect to the types of models that can be applied at this layer, the same argument given for the session layer holds.

2.5. SERVICE LAYER

The **service layer** is responsible for actually generating the traffic. Parameters are the traffic volume and a characterisation of the stream of requests over time (e.g. the interarrival time of packets).

At this level, the models discussed in DEL1.1, Section 3, could be applied. For the type of services to be implemented within BISANTE, we give a more detailed discussion in the following section.

3. THE BISANTE SERVICES

What services do within the BISANTE workload generation framework has been explained in the previous section. In this section, all services that are needed in the BISANTE case studies will be explained in detail.

First, the most important features of the services will be described according to the service specification. Most emphasis here will go into the description of protocol headers and states. Then, those parts that will be included into the models will be defined. Here, the parts actually causing network load are of special importance. Parts being responsible for defining protocol internal states will be neglected if possible. If necessary, the differences between ns and OPNET implementations will be pointed out. Here, a maximum reuse of existing implementations in either of these simulators is desired.

Some protocols such as HTTP/1.0 will be described using an augmented Backus-Naur Form (BNF). Details of this notation are explained in the Appendix B.

Most protocols are defined in the Internet Request for Comments (RFCs), available, for example, from http://gd.tuwien.ac.at. Many standards have been updated or obsoleted by other standards over the years. A list of currently valid standards can be found in RFCs with numbers XX00. The RFC standards described below have been taken from RFC 2500.

3.1. HTTP/1.0

The Hypertext Transfer Protocol Version 1.0 (HTTP/1.0) is defined in the RFC 1945 specification. It is used to transport WWW traffic, both over intranets and the internet. It is also used as a generic protocol for communication between user agents and proxies/gateways to other Internet protocols such as SMTP, NNTP, FTP, Gopher and so on. HTTP/1.0 is an application level, generic, stateless and object-oriented protocol. Special formats of, for example, URLs, date and time strings etc. will not be explained in detail, as they will be represented in the HTTP/1.0 model by their length only.

HTTP/1.0 operates according to the request-response paradigm. A client establishes a connection to a server and sends a request to the server in the form of request method, URI and protocol version, followed by a MIME-like message containing request modifiers, client information and possible body content. The server responds with a status line, including the message´s protocol version and a success or error code, followed by a MIME-like message containing server information, entity meta-information and possible body content.

3.1.1. HTTP-HEADERS

HTTP-header fields include *General-Headers*, *Request-Headers*, *Response-Headers* and *Entity-Headers*.

Each header field of a name followed immediately by a colon (":"), a single space (SP) character, and the field value. Field names are case-insensitive. Header fields can be extended over multiple lines by preceding each extra line with at least one SP or HT, though this is not recommended.

HTTP-header = field-name ":" [field-value] CRLF

field-name = token

*field-value = *(field-content | LWS)*

*field-content = <the OCTETs making up the field-value and consisting of either *TEXT or combinations of token, tspecials, and quoted-string>*

The order in which header fields are received is not significant.

General-Header fields can be found both in requests and responses.

General-Header = Date | Pragma

3.1.2. HTTP-UNIFORM RESOURCE LOCATOR (URL)

The "http" scheme is used to locate network resources via the HTTP protocol. This section defines the scheme-specific syntax and semantics for http URLs.

http_URL = "http:" "//" host [":" port] [abs_path]

host = <A legal Internet host domain name or IP address (in dotted-decimal form),as defined by Section 2.1 of RFC 1123>

*port = *DIGIT*

If the port is empty or not given, port 80 is assumed. The semantics are that the identified resource is located at the server listening for TCP connections on that port of that host, and the Request-URI for the resource is *abs_path*. If the *abs_path* is not present in the URL, it must be given as "/" when used as a Request-URI .

3.1.3. HTTP-TIMESTAMPS

All HTTP/1.0 date/time stamps must be represented in Universal Time (UT), also known as Greenwich Mean Time (GMT), without exception. This is indicated in the first two formats by the inclusion of "GMT" as the three-letter abbreviation for time zone, and should be assumed when reading the asctime format.

HTTP-date = rfc1123-date | rfc850-date | asctime-date

rfc1123-date = wkday "," SP date1 SP time SP "GMT"

rfc850-date = weekday "," SP date2 SP time SP "GMT"

asctime-date = wkday SP date3 SP time SP 4DIGIT

date1 = 2DIGIT SP month SP 4DIGIT ; day month year (e.g., 02 Jun 1982)

date2 = 2DIGIT "-" month "-" 2DIGIT ; day-month-year (e.g., 02-Jun-82)

date3 = month SP (2DIGIT | (SP 1DIGIT)) ; month day (e.g., Jun 2)

time = 2DIGIT ":" 2DIGIT ":" 2DIGIT ; 00:00:00 - 23:59:59

wkday = "Mon" | "Tue" | "Wed" | "Thu" | "Fri" | "Sat" | "Sun"

weekday = "Monday" | "Tuesday" | "Wednesday"| "Thursday" | "Friday" | "Saturday" |

"Sunday"

month = "Jan" | "Feb" | "Mar" | "Apr" | "May" | "Jun" | "Jul" | "Aug" | "Sep" | "Oct"

| "Nov" | "Dec"

3.1.4. HTTP MESSAGES

HTTP messages include requests (sent from client to server) and responses (sent from server to client):

HTTP-Message = Full-Request | Full-Response

Note: HTTP/0.9 includes also Simple-Request and Simple-Response, which will not be treated here.

Both messages may include optional header fields (headers) and an entity body. The entity is separated from the headers by a null line (a line with nothing preceding the CRLF.

3.1.5. HTTP-REQUEST

A Full-Request has the format

*Full-Request = Request-Line *(General-Header | Request-Header | Entity-Header)*

CRLF [Entity-Body]

Request-Line = Method SP Request-URI SP HTTP-Version CRLF

Request-URI = absoluteURI | abs_path

HTTP-Version= “HTTP/1.0”

The Method token indicates the method to be performed. The Method is case-sensitive.

Method = "GET | "HEAD" | "POST" | extension-method

extension-method = token

The *absoluteURI* is only allowed, if the request is made to a proxy. In the *Request-Header*, the client can pass additional information about the request and the client itself.

Request-Header = Authorization | From | If-Modified-Since | Referer | User-Agent

Authorization = "Authorization" ":" credentials

From = "From" ":" mailbox

If-Modified-Since = "If-Modified-Since" ":" HTTP-date

Referer = "Referer" ":" (absoluteURI | relativeURI)

User-Agent = "User-Agent" ":" 1(product | comment)*

A user agent that wishes to authenticate itself with a server -- usually, but not necessarily, after receiving a 401 response -- may do so by including an *Authorization* request-header field with the request. The *From* request-header field, if given, should contain an Internet e-mail address for the human user who controls the requesting user agent. The If-Modified-Since request-header field is used with the GET method to make it conditional: if the requested resource has not been modified since the time specified in this field, a copy of the resource will not be returned from the server; instead, a 304 (not modified) response will be returned without any Entity-Body. The Referer request-header field allows the client to specify, for the server's benefit, the address (URI) of the resource from which the Request-URI was obtained. The User-Agent request-header field contains information about the user agent originating the request.

3.1.6. HTTP-RESPONSE

A *Full-Response* message has the format

Full-Response = *Status-Line*(General-Header | Response-Header | Entity-Header)*

CRLF [Entity-Body]

Status-Line = *HTTP-Version SP Status-Code SP Reason-Phrase CRLF*

The Status-Code element is a 3-digit integer result code of the attempt to understand and satisfy the request. The Reason-Phrase is intended to give a short textual description of the Status-Code. The first digit of the Status-Code defines the class of response. The last two digits do not have any categorization role. There are 5 values for the first digit:

- 1xx: Informational - Not used, but reserved for future use
- 2xx: Success - The action was successfully received, understood, and accepted.
- 3xx: Redirection - Further action must be taken in order to complete the request
- 4xx: Client Error - The request contains bad syntax or cannot be fulfilled
- 5xx: Server Error - The server failed to fulfill an apparently valid request

The individual values of the numeric status codes defined for HTTP/1.0, and an example set of corresponding Reason-Phrase's, are presented below:

Status-Code	=	*"200"*	*; OK*
	|	*"201"*	*; Created*
	|	*"202"*	*; Accepted*
	|	*"204"*	*; No Content*
	|	*"301"*	*; Moved Permanently*
	|	*"302"*	*; Moved Temporarily*
	|	*"304"*	*; Not Modified*
	|	*"400"*	*; Bad Request*
	|	*"401"*	*; Unauthorized*
	|	*"403"*	*; Forbidden*

| "404" ; Not Found
| "500" ; Internal Server Error
| "501" ; Not Implemented
| "502" ; Bad Gateway
| "503" ; Service Unavailable
| extension-code
extension-code = 3DIGIT
Reason-Phrase = *<TEXT, excluding CR, LF>

The response header fields allow the server to pass additional information about the response which cannot be placed in the Status-Line. These header fields give information about the server and about further access to the resource identified by the Request-URI.

Response-Header = Location | Server | WWW-Authenticate
Location = "Location" ":" absoluteURI
Server = "Server" ":" 1*(product | comment)
WWW-Authenticate = "WWW-Authenticate" ":" 1#challenge

The Location response-header field defines the exact location of the resource that was identified by the Request-URI. For 3xx responses, the location must indicate the server's preferred URL for automatic redirection to the resource. Only one absolute URL is allowed. The Server response-header field contains information about the software used by the origin server to handle the request. The WWW-Authenticate response-header field must be included in 401 (unauthorized) response messages. The field value consists of at least one challenge that indicates the authentication scheme(s) and parameters applicable to the Request-URI.

3.1.7. ENTITY

Full-Request and Full-Response messages may transfer an entity within some requests and responses. An entity consists of Entity-Header fields and (usually) an Entity-Body. Entity-Header fields define optional meta-information about the Entity-Body or, if no body is present, about the resource identified by the request.

Entity-Header = Allow | Content-Encoding | Content-Length | Content-Type | Expires
| Last-Modified | extension-header
extension-header = HTTP-header
Expires = "Expires" ":" HTTP-date
Last-Modified = "Last-Modified" ":" HTTP-date
Entity-Body = *OCTET

The Expires entity-header field gives the date/time after which the entity should be considered stale. The Last-Modified entity-header field indicates the date and time at which the sender believes the resource was last modified.

3.1.8. HTTP-METHODS

The set of HTTP/1.0 Methods will be described in this section.

The GET method means retrieve whatever information (in the form of an entity) is identified by the *Request-URI*. The semantics of the GET method changes to a "conditional GET" if the request message includes an If-Modified-Since header field. A conditional GET method requests that the identified resource be transferred only if it has been modified since the date given by the *If-Modified-Since* header.

The HEAD method is identical to GET except that the server must not return any Entity-Body in the response. The meta-information contained in the HTTP headers in response to a HEAD request should be identical to the information sent in response to a GET request. There is no "conditional HEAD" request analogous to the conditional GET.

The POST method is used to request that the destination server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line.

3.1.9. MODELLING HTTP/1.0

The BISANTE HTTP/1.0 model will include GET and POST methods. Modelling these will be done by some stochastic point process situated in the application or command layer of the user model, computing the inter-arrival time between two request messages. The inter-arrival time may depend on the request method (GET or POST) and the document (MIME) type. Thus, these decisions will also be made at the application or command layer.

The request messages will generally have the format

Full-Request	=	*Request-Line *(Request-Header)*
		CRLF [Entity-Body]
Request-Line	=	*Method SP Request-URI SP HTTP-Version CRLF*
Request-URI	=	*abs_path*
HTTP-Version	=	*“HTTP/1.0”*
Method	=	*"GET | POST "*
Request-Header	=	*User-Agent*
If-Modified-Since	=	*"If-Modified-Since" ":" "Sun, 06 Nov 1994 08:49:37 GMT"*
User-Agent	=	*"User-Agent" ":" "Netscape 4.6"*
Entity-Body	=	**OCTET*

The content of the request message is regarded as unimportant, the result will only depend on whether the message is a GET or POST. Instead, the length of the request message will be created by adding the fixed sized parts and generating varying sized parts:

First, it is decided, whether the message is a GET or POST. Then, the size of the Request-URI will be computed. If the message is a GET, it will be decided, whether the object can be retrieved from the cache. If not, the request message will be sent to the server.

If the message is a POST, the size of the Entity-Body will be computed and then, the message will be sent.

The *Full-Response* messages have the format

```
Full-Response      =    Status-Line*( Response-Header | Entity-Header )
                        CRLF [ Entity-Body ]
Status-Line        =    "HTTP/1.0" SP Status-Code SP Reason-Phrase CRLF
Status-Code        =    "200"   ; OK
                   |    "201"   ; Created
                   |    "202"   ; Accepted
                   |    "204"   ; No Content
                   |    "301"   ; Moved Permanently
                   |    "302"   ; Moved Temporarily
                   |    "304"   ; Not Modified
                   |    "400"   ; Bad Request
                   |    "401"   ; Unauthorized
                   |    "403"   ; Forbidden
                   |    "404"   ; Not Found
                   |    "500"   ; Internal Server Error
                   |    "501"   ; Not Implemented
                   |    "502"   ; Bad Gateway
                   |    "503"   ; Service Unavailable
Reason-Phrase      =    *<TEXT, excluding CR, LF>
Response-Header    =  Server
Server             = "Server" ":" " CERN/3.0 libwww/2.17"
Entity-Header      = Expires | Last-Modified
extension-header   = HTTP-header
Expires            = "Expires" ":" "Sun, 06 Nov 1994 08:49:37 GMT"
Last-Modified      = "Last-Modified" ":"  "Sun, 06 Nov 1994 08:49:37 GMT"
Entity-Body        = *OCTET
```

When receiving a request message, the server will compute one of the possible *Status-Codes* according to the request method (GET or POST) some probability. If the code represents an error, the corresponding error message will be sent back without *Entity-Header* and *Entity-Body*. If the *Status-Code="200"*, an *Entity-Header* and, according to some distribution, the size of the *Entity-Body* will be computed.

3.1.10. MODELLING CACHING

Caches can have an important influence on the data size transported over the network When modelling the influence of caches, one of the following methods can be chosen:

- Detailed: Each client will be associated with a cache stack of *N* MB. At first, each cache is said to be empty. When downloading a new HTTP object, it will be put onto the cache stack. Additionally, each cache stack is associated with a locality stack, holding probabilities p1, p2 to pn. pk denotes the probability that the next downloaded object is the same that has been downloaded k steps before. If this object is still in the cache stack, it will be retrieved and the object will be put on top of the cache stack. If it is not in the cache stack, a new object will be downloaded and put on top of the stack. If it does not fit into the cache stack anymore, objects are removed from below of the cache stack (Least Recently Used LRU), until it fits in.
- Aggregated: In an aggregated model, there is only one probability p denoting the probability that the next object is found in the cache. Such probabilities p can be found either directly by analysing logfiles, or by running detailed models first, and then analysing their behaviour. This way, the probability may depend on the cache size *N*.

Each model can be enhanced by modelling the influence of document expiration and If-modified-since rules. The data for these probabilities can be found in logfiles themselves.

3.2. TELNET

The TELNET protocol is defined in RFC 854 and is meant for process-process or terminal-terminal communication. When a TELNET connection is to be created at a remote server, usually the client tries to open a TCP connection at port 23 of the server offering the TELNET service.

TELNET has the following properties:

Each TELNET protocol must be able to support a minimum set of commands. A virtual terminal is defined that exactly supports these command set, and is called Network Virtual Terminal (NVT). Each TELNET protocol must therefore be able to act as an NVT. Above that, each TELNET protocol can then negotiate with its partner about further options it is willing to do, or it is unwilling to support. For this, TELNET supports the four messages DO, DON´T, WILL and WON´T.

If terminal A wants to use a certain option XXX itself, it sends a WILL XXX to B. If B is willing to accept this, it sends a DO XXX, else it sends a DON´T XXX. Upon receiving the answer from B, A will either use option XXX (DO XXX9 or will not (DON´T XXX). If A wants B to use option YYY, it sends DO YYY to B, which then answers with WILL YYY or WON´T YYY. Again, A regards the answer as acknowledgement or denial of the request.

If both terminals cannot agree upon any option, each at least will support the NVT.

3.2.1. The Network Virtual Terminal

The Network Virtual Terminal (NVT) is a bi-directional character device. The NVT has a printer and a keyboard. The printer responds to incoming data and the keyboard produces outgoing data which is sent over the TELNET connection and, if "echoes" are desired, to the NVT's printer as well. "Echoes" will not be expected to traverse the network (although options exist to enable a "remote" echoing mode of operation, no host is required to implement this option). The code set is seven-bit USASCII in an eight-bit field, except as modified herein. Any code conversion and timing considerations are local problems and do not affect the NVT.

An NVT consists of a keyboard and a printer. If an NVT receives data from its partner, it sends this to the printer. If it receives data from the keyboard, it sends this to its partner.

Unless otherwise negotiated, NVTs support the following behaviour:

1. The NVT tries to accumulate a complete line before it transmits it over the network.
2. When a process has completed sending data to an NVT printer and has no queued input from the NVT keyboard for further processing (i.e., when a process at one end of a TELNET connection cannot proceed without input from the other end), the process must transmit the TELNET Go Ahead (GA) command.

3.2.2. Control Functions

TELNET defines five standard functions with standard, yet not required meanings. If a system does not provide this function to its users, it can treat it as a no-operation. Otherwise, it must be provided to local and network users. The standard functions are:

Interrupt Process (IP): Stops the currently running process

Abort Output (AO): Allows the running process to run to completion, but stops the output of the process to the

Are You There (AYT): Allows printable evidence that the system at the other side is still up and running.

Erase Character (EC): Deletes the last preceding undeleted character.

Erase Line (EL): Deletes all characters in the current line of input

Additionally, TELNET provides special SYNC messages to transport these signals instantaneously over the TCP connection.

3.2.3. Sending data from keyboards to printers.

There is a standard set of keys that are defined for NVT keyboards and printers and must be support by all TELNET implementations. Whenever a normal key has been pressed, its US-ASCII code is stored in a line buffer at the user side. As soon as an end-of-line character <CR LF> is received from the keyboard, the line buffer is sent to the partner. If special commands are to be sent, the command is preceded by one byte with value 255, being followed by either one or two bytes, one for the command code, the other for a possible option. Commands include the above described special functions, as well as commands for DO, DON´T, WILL and WON´T. Sending the byte 255 must be done by preceding it with 255.

3.2.4. MODELLING TELNET

The BISANTE TELNET model will include two steps:

1. First, the connection is made. In this phase, a fixed or stochastic amount of data is sent from both sides to model negotiation of options. After this, no more negotiation will take place.
2. The user behaviour will be modelled by applying a client-server model. At the application or command layer, a point process will compute inter-arrival times for user inputs. After typing in a complete line, the data is sent to the server, where a certain amount of output is generated and sent back.

No extra models will be included for commands, as the length of each keyboard or printer data sent will be modelled with one general distribution only.

3.3. FTP

The FTP protocol is defined in RFC 959 and is meant for the transfer of files from one Internet computer to the other.

For data transfer, FTP requires two independent TCP connections. First, the User Protocol Interpreter (user-PI) establishes a TELNET connection to the Server Protocol Interpreter (server-PI) on port 25.

3.3.1. CONTROL CONNECTION

Using the TELNET connection, the user-PI may send standard commands to the server-PI. These commands in the line oriented TELNET style, each line being closed with an end-of-line code. Each line starts with a FTP command that is at most 4 characters long, then, separated with SPs, the command options follow.

Every command must generate at least one reply, there may be more than one. Each FTP reply starts with a three digit number, followed by one SP and one line of text, and terminated by the TELNET end-of-line. In some cases, replies may contain more than one line. Here, the reply starts with a three digit number, followed by a “-“, then several lines of text. The last line of the reply will start with the same three digit number, followed by an SP and one line of text, again closed with an end-of-line.

3.3.2. DATA CONNECTION

The mechanics of transferring data consists of setting up the data connection to the appropriate ports and choosing the parameters for transfer. Both the user and the server-DTPs have a default data port. The user-process default data port is the same as the control connection port (25). The server-process default data port is the port adjacent to the control connection port (24).

The passive data transfer process (this may be a user-DTP or a second server-DTP) shall "listen" on the data port prior to sending a transfer request command. The FTP request command determines the direction of the data transfer. The server, upon receiving the transfer request, will initiate the data connection to the data port. When the connection is established, the data transfer begins between DTP's, and the server-PI sends a confirming reply to the user-PI. Every FTP implementation must support the use of the default data ports, and only the USER-PI can initiate a change to non-default ports.

The whole concept can be seen in Figure 1.

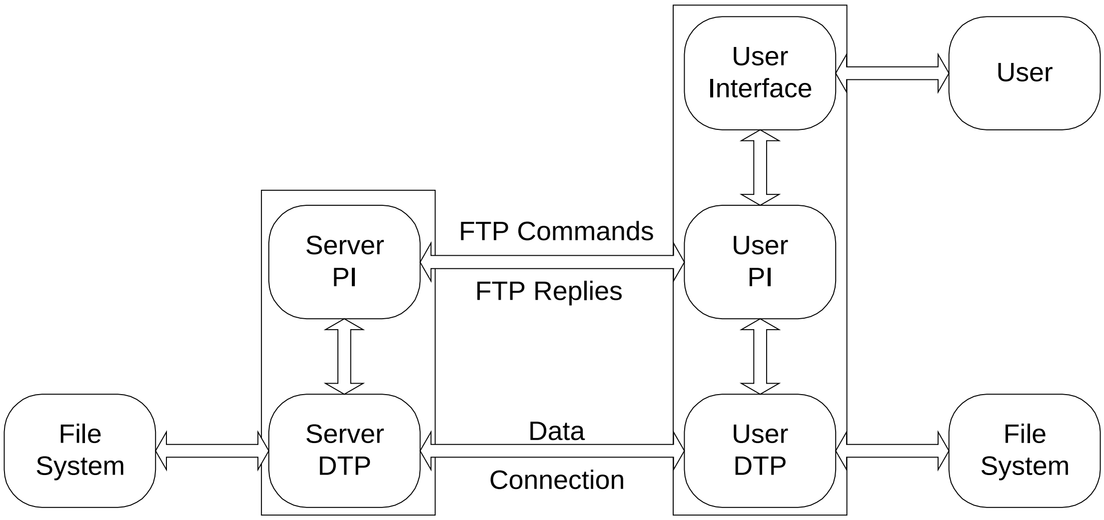


Figure 1: FTP concept.

3.3.3. DATA TYPES

FTP allows several different data types to be sent:

ASCII

> This is the default type and must be accepted by all FTP implementations. It is intended primarily for the transfer of text files, except when both hosts would find the EBCDIC type more convenient. The sender converts the data from an internal character representation to the standard 8-bit NVT-ASCII representation (see the Telnet specification). The receiver will convert the data from the standard form to his own internal form.
>
> In accordance with the NVT standard, the <CRLF> sequence should be used where necessary to denote the end of a line of text. (See the discussion of file structure at the end of the Section on Data Representation and Storage.) Using the standard NVT-ASCII representation means that data must be interpreted as 8-bit bytes.

EBCDIC

> For transmission, the data are represented as 8-bit EBCDIC characters. The character code is the only difference between the functional specifications of EBCDIC and ASCII types.

IMAGE

> The data are sent as contiguous bits which, for transfer, are packed into the 8-bit transfer bytes. The receiving site must store the data as contiguous bits. The structure of the storage system might necessitate the padding of the file (or of

each record, for a record-structured file) to some convenient boundary (byte, word or block). This padding, which must be all zeros, may occur only at the end of the file (or at the end of each record) and there must be a way of identifying the padding bits so that they may be stripped off if the file is retrieved. The padding transformation should be well publicised to enable a user to process a file at the storage site.

Image type is intended for the efficient storage and retrieval of files and for the transfer of binary data. It is recommended that this type be accepted by all FTP implementations.

LOCAL

The data is transferred in logical bytes of the size specified by the obligatory second parameter, Byte size. The value of Byte size must be a decimal integer; there is no default value. The logical byte size is not necessarily the same as the transfer byte size. If there is a difference in byte sizes, then the logical bytes should be packed contiguously, disregarding transfer byte boundaries and with any necessary padding at the end.

3.3.4. TRANSMISSION MODES

All data transfers must be completed with an end-of-file (EOF) which may be explicitly stated or implied by the closing of the data connection. For files with record structure, all end-of-record markers (EOR) are explicit, including the final. For files transmitted in page structure a "last-page" page type is used.

In FTP, there are three transmission modes:

STREAM MODE

The data is transmitted as a stream of bytes. There is no restriction on the representation type used; record structures are allowed.

In a record structured file EOR and EOF will each be by a two-byte control code. The first byte of the control will be all ones, the escape character. The second byte have the low order bit on and zeros elsewhere for EOR and the second low order bit on for EOF; that is, the byte will have value 1 for EOR and value 2 for EOF. EOR and EOF may be indicated together on the last byte transmitted by turning low order bits on (i.e., the value 3). If a byte of all ones was intended to be sent as data, it should be repeated in the second byte of the control code.

If the structure is a file structure, the EOF is indicated by the sending host closing the data connection and all bytes are data bytes.

BLOCK MODE

The file is transmitted as a series of data blocks preceded one or more header bytes. The header bytes contain a count field, and descriptor code. The count field indicates the total length of the data block in bytes, thus marking beginning of the next data block (there are no filler bits). The descriptor code defines: last block in the file (EOF) last block in the record (EOR), restart or suspect data (i.e., the data being transferred is suspected of errors and is not reliable). This last code is NOT intended for error control within FTP. It is motivated by the desire of sites exchanging certain types of data to send and receive all the data despite local, but to indicate in the transmission that certain portions are suspect. Record structures are allowed in this mode, and any representation type may be used.

The header consists of the three bytes. Of the 24 bits of header information, the 16 low order bits shall represent byte count, and the 8 high order bits shall represent descriptor codes as shown below.

COMPRESSED MODE

There are three kinds of information to be sent: regular data, sent in a byte string; compressed data, consisting of replications or filler; and control information, sent two-byte escape sequence. If n>0 bytes (up to 127) of data are sent, these n bytes are preceded by a byte with the left-most bit set to 0 and the right-most 7 bits containing the number n.

3.3.5. MODELLING FTP

The BISANTE FTP model will behave as follows:

- The model decides to start FTP and opens a command connection to the desired server.
- The user model can choose between two possible commands:
  1. Commands that do not open data connections (like CWD). Here, replies will sent back from the server as single lines over the open control connection.
  2. Commands, that open a data connection. In this case, the server will open a data connection on the local port 25 and the data will be sent as binary in stream mode. Such commands include RETR, STOR and LIST.
- In both cases, the number of transferred bytes will be drawn from distributions.

3.4. SMTP

The Simple Mail Transfer Protocol (SMTP) is defined in RFC 821 and is meant for the transfer of messages from one Internet computer to the other. Messages are usually created by mail applications and will be sent to the Internet computer holding the mailbox of the receiver. Messages may be sent directly to the receiver´s host, or may be forwarded in a predefined or dynamically defined sequence of SMTP hosts.

3.4.1. SMTP PROCEDURES

The SMTP procedures are defined as follows:

1. First, the sender-SMTP opens a TCP connection to port 25 of the receiver host.
2. Then, the sender-SMTP starts to send a sequence of simple commands to the receiver-SMTP.
3. Each command consists of 4 characters defining the command, then a list with readable options, each separated wit a SP. Each line is ended with a <CRLF>.
4. Each commands must be replied by a reply message. This may contain one or more lines, each line being ended with <CRLF>. Each single reply-line starts with a numeric code, then, separated with an SP, a textual representation of the reply

code. In a multi-line answer, each but the last line will contain first a numeric code, then a "-", then the text. The format of the last line will be again a numeric code, then an SP and possible text.

5. Finally, the sender-SMTP sends the mail through the same TCP connection. Each line must be ended with <CRLF>. The end-of-text is marked by a single line holding ". <CRLF>".

The model of SMTP can be seen in Figure 2.

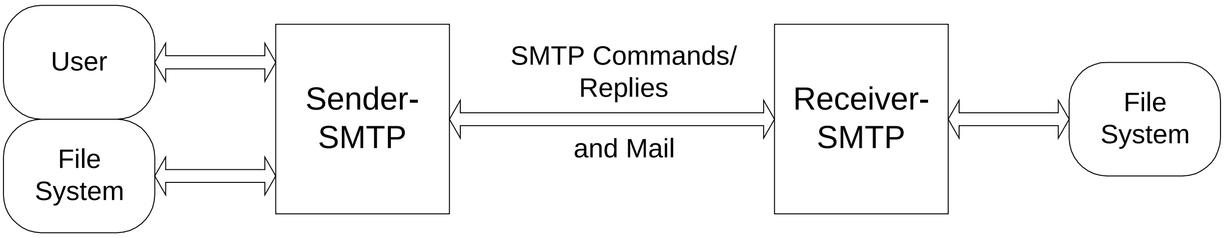


Figure 2: SMTP model.

3.4.2. SMTP COMMANDS

In SMTP, the sender-SMTP may send the following commands:

HELLO (HELO)

The syntax for this command is:
HELO <SP> <domain> <CRLF>
With this command, the sender-SMTP tells the receiver-SMTP its domain name.

MAIL (MAIL)

The syntax for this command is:
MAIL <SP> FROM:<reverse-path> <CRLF>
This command tells the SMTP-receiver that a new mail transaction is starting and to reset all its state tables and buffers, including any recipients or mail data. It gives the reverse-path which can be used to report errors. If accepted, the receiver-SMTP returns a 250 OK reply.

RECIPIENT (RCPT)

The syntax for this command is:
RCPT <SP> TO:<forward-path> <CRLF>
This command gives a forward-path identifying one recipient. If accepted, the receiver-SMTP returns a 250 OK reply, and stores the forward-path. If the recipient is unknown the receiver-SMTP returns a 550 Failure reply. This second step of the procedure can be repeated any number of times.

DATA (DATA)

The syntax for this command is:
DATA <CRLF>
If accepted, the receiver-SMTP returns a 354 Intermediate reply and considers all succeeding lines to be the message text. When the end of text is received and stored the SMTP-receiver sends a 250 OK reply. Since the mail data is sent on the transmission channel the end of the mail data must be indicated so that the

command and reply dialog can be resumed. SMTP indicates the end of the mail data by sending a line containing only a period. A transparency procedure is used to prevent this from interfering with the user's text.
Please note that the mail data includes the memo header items such as Date, Subject, To, Cc, From. The end of mail data indicator also confirms the mail transaction and tells the receiver-SMTP to now process the stored recipients and mail data. If accepted, the receiver-SMTP returns a 250 OK reply. The DATA command should fail only if the mail transaction was incomplete (for example, no recipients), or if resources are not available.

VERIFY (VRFY)

The syntax for this command is:
VRFY username <CRLF>
With this name, the sender-SMTP may verify that the specified mailbox exists at the receiver host.

EXPAND (EXPN)

The syntax for this command is:
EXPN mailing-list <CRLF>
With this command, the sender-SMTP may expand a mailing-list defined at the receiver-SMTP. If this list exists, the receiver-SMPT will send back a multi-line answer, each line holding one mailbox name.

QUIT (QUIT)

The syntax for this command is:
QUIT <CRLF>
With this command, the sender-SMTP closes the connection.

3.4.3. MODELLING SMTP

The SMTP model will consists of a sender part and a receiver part. The following procedures will be implemented:

1. The sender part will first open a TCP connection on port 25 of the receiver host.
2. Then, a number of commands and replies will be sent (Only the length of these messages will be modelled). It will be assumed that the receiving mailbox always exists.
3. Finally, the message body will be sent, followed by a reply from the receiver model.
4. Finally, the connection will be closed.

3.5. POP3

The Post Office Protocol Version 3 (POP3) is defined in RFC 1939 and is meant for retrieving messages that have been intermediately stored in servers permanently connected to the Internet. This might be necessary for all users having only temporarily access to the Internet, for example via dial-up connections over telephone lines.

POP3 allows seeing information of stored messages, retrieving and deleting them.

3.5.1. POP3 PROCEDURES

Initially, the POP3 server listens on TCP port 110 for incoming connection requests. Clients wishing to retrieve messages open a TCP connection there and start sending commands consisting of 4 characters, followed by a list of options separated with SP. Each command must be replied either by a status indicator "+OK" or "-ERR", indicating success or failure of the command, followed by one or more lines. Multi-lines replies are terminated by a single line holding a period "." and <CRLF>.

POP3 progresses through a number of states during one session.

3.5.2. THE AUTHORIZATION STATE

Once a connection has been established and the POP3 server has sent greetings, the session enters the AUTHORIZATION state. In this state, the client must identify itself to the server.

Possible Commands are:

USER name

> With this command, the client tells the server the user name it wishes to retreive messages for.

PASS string

> Here, the client sends the user password.

APOP name digest

> An additional method of identifying the user in order to prevent the client from sending the username/password data too often.

QUIT

> If the QUIT command is issued in the AUTHORIZATION state, the session will not enter the UPDATE state.

3.5.3. THE TRANSACTION STATE

Once the client has successfully done this, it enters the TRANSACTION state. In this state, the client can issue commands and retrieve message information, messages themselves or mark or unmark messages as deleted. When entering this state, the server first obtains a lock on the user mailbox, preventing other instances of POP3 to open the mailbox in parallel. Then, each message is assigned a unique number starting with 1. The client may now issue any command appropriate in this state. Finally, the client will issue the QUIT command, forcing the server to enter the UPDATE state.

Possible commands are:

STAT

> Upon receiving this command, the server sends back one line of reply, holding the status indicator, the number of messages in the mailbox and the total number of bytes of all messages.

LIST [msg]

If a message number is specified, the server will reply one line, starting with the status indicator, then holding the message number and the size of the message in bytes.

If no message number is given, the server will reply first with one line holding the status indicator and the number of messages and total bytes, then with a multi-line answer, where for each message a line starts with the message number, followed by the size of the message in bytes.

RETR [msg]

If the POP3 server responds with a positive indicator, the rest of the reply will hold the message body of message msg. The message will end with a line containing a period "." followed by a <CRLF>.

DELE msg

Upon receiving this command, the server will mark message number msg as deleted. The message, however, will be physically deleted only in the update state.

NOOP

The POP3 server does nothing. Used for preventing the connection from a time-out.

RSET

All messages marked as deleted are unmarked again.

QUIT

Upon receiving this command, the session will enter the UPDATE state, where all messages marked as deleted will be physically removed.

3.5.4. THE UPDATE STATE

Finally, the client will send the QUIT command, after which the session enters the UPDATE state. In this state, the server will release any resources acquired in the session and will delete all messages that have been marked as "deleted".

3.5.5. MODELLING POP3

The POP3 models will consist of two parts, the client model and the server model. The client model will first open a TCP connection to the server on port 110. Then, the client model will send several messages simulating the USER, PASS, LIST and RETR commands. The server will answer with reply commands, where the reply message length will be short for USER and PASS and longer for list and RETR. RETR commands can be sent for several times, until the connection will be closed with a QUIT command.

3.6. NNTP

The Network News Transfer Protocol (NNTP) is defined in RFC 977 and is meant for the distribution of news articles over an intranet or the Internet. It consists of central servers communicating with each other. New articles are spread from one server to another in a way that only those articles are sent which are unknown to the receiving server. Clients connecting to these servers may retrieve information about newsgroups and articles, or may download headers and article bodies. Clients may also create new articles or newsgroups.

3.6.1. NNTP PROCEDURES

Typically, such an NNTP server waits on TCP port 119 for connection requests. Upon the creation of a TCP connection, the client will send commands to the server, requesting information about new newsgroups, new articles and so on. Each command consists of one line of readable ASCII text, terminated by an end-of-line <CRLF>. The first characters define the command, which may be then followed by one or several parameters, separated by an SP character. For each command, the server will send a reply, consisting of one or several lines, each line being terminated by <CRLF>. Each reply will start with one line containing the status response. This line starts with a three digit number describing the return code, then followed by other parameters. In the case of a multi-line answer, the following lines will be treated as belonging to the answer, until one line containing only a period, followed by a <CRLF> is sent.

The following commands may be sent to the server:

NEWGROUPS

Format:

NEWGROUPS date time [GMT] [<distributions>]

A list of newsgroups created since <date and time> will be listed in the same format as the LIST command.

The optional parameter "distributions" is a list of distribution groups, enclosed in angle brackets. If specified, the distribution portion of a new newsgroup (e.g, 'net' in 'net.wombat') will be examined for a match with the distribution categories listed, and only those new newsgroups which match will be listed. If more than one distribution group is to be listed, they must be separated by commas within the angle brackets.

LIST

Format:
LIST
Returns a list of valid newsgroups and associated information. Each newsgroup is sent as a line of text in the following format:
group last first p
where <group> is the name of the newsgroup, <last> is the number of the last known article currently in that newsgroup, <first> is the number of the first article currently in newsgroup, and <p> is either 'y' or 'n' indicating whether posting to this newsgroup is allowed ('y') or prohibited ('n').
The <first> and <last> fields will always be numeric. They may have leading zeros. If the <last> field evaluates to less than the <first> field, there are no articles currently on file in the newsgroup.

GROUP

Format:

GROUP ggg

The required parameter ggg is the name of the newsgroup to be selected (e.g. "net.news"). A list of valid newsgroups may be obtained from the LIST command.

The successful selection response will return the article numbers of the first and last articles in the group, and an estimate of the number of articles on file in the group. It is not necessary that the estimate be correct, although that is helpful; it must only be equal to or larger than the actual number of articles on file. (Some implementations will actually count the number of articles on file. Others will just subtract first article number from last to get an estimate.)

When a valid group is selected by means of this command, the internally maintained "current article pointer" is set to the first article in the group.

NEWNEWS

Format:

NEWNEWS newsgroups date time [GMT] [<distribution>]

A list of message-ids of articles posted or received to the specified newsgroup since "date" will be listed. The format of the listing will be one message-id per line, as though text were being sent. A single line consisting solely of one period followed by CR-LF will terminate the list.

The optional parameter "distributions" is a list of distribution groups, enclosed in angle brackets. If specified, the distribution portion of an article's newsgroup (e.g, 'net' in 'net.wombat') will be examined for a match with the distribution categories listed, and only those articles which have at least one newsgroup belonging to the list of distributions will be listed. If more than one distribution group is to be supplied, they must be separated by commas within the angle brackets.

ARTICLE

Format:

1. ARTICLE <message-id>
2. ARTICLE [message number]

Display the header, a blank line, then the body (text) of the specified article.

HEAD

Format:

1. HEAD <message-id>
2. HEAD [message number]

The HEAD command is similar to the ARTICLE command, except that it returns the header lines only.

BODY

Format:

1. BODY <message-id>
2. BODY [message number]

The BODY command is similar to the ARTICLE command, except that it returns the body text lines only.

STAT

Format:

1. STAT <message-id>
2. STAT [message number]

The STAT command is similar to the ARTICLE command except that no text is returned. When selecting by message number within a group, the STAT command serves to set the current article pointer without sending text. The returned acknowledgement response will contain the message-id, which may be of some value. Using the STAT command to select by message-id is valid but of questionable value, since a selection by message-id does NOT alter the "current article pointer".

NEXT

Format:

NEXT

The internally maintained "current article pointer" is advanced to the next article in the current newsgroup. If no more articles remain in the current group, an error message is returned and the current article remains selected.

The internally-maintained "current article pointer" is set by this command.

A response indicating the current article number, and the message-id string will be returned. No text is sent in response to this command.

LAST

Format:

LAST

The internally maintained "current article pointer" is set to the previous article in the current newsgroup. If already positioned at the first article of the newsgroup, an error message is returned and the current article remains selected.

The internally-maintained "current article pointer" is set by this command..

POST

Format:

POST

If posting is allowed, response code 340 is returned to indicate that the article to be posted should be sent. Response code 440 indicates that posting is prohibited for some installation-dependent reason.

If posting is permitted, the article should be presented in the format specified by RFC850, and should include all required header lines. After the article's header and body have been completely sent by the client to the server, a further response code will be returned to indicate success or failure of the posting attempt.

IHAVE

Format:

IHAVE <messageid>

The IHAVE command informs the server that the client has an article whose id is <messageid>. If the server desires a copy of that article, it will return a response instructing the client to send the entire article. If the server does not want the article (if, for example, the server already has a copy of it), a response indicating that the article is not wanted will be returned.

QUIT

Format:

QUIT

The server process acknowledges the QUIT command and then closes the connection to the client. This is the preferred method for a client to indicate that it has finished all its transactions with the NNTP server.

If a client simply disconnects (or the connection times out, or some other fault occurs), the server should gracefully cease its attempts to service the client.

3.6.2. MODELLING NNTP

The NNTP model will consist of two independent submodels:

1. The client model open a TCP connection to port 119 of the server model. It will then send a sequence of commands to the server and will receive replies. The command type is not important and will be omitted. The command message length will be derived from suitable distributions. After K commands, the client will end the connection.
2. The NNTP server model will wait for connection requests. It will then wait for command messages and send back replies of messages of length N. The message length will be derived from distributions.

3.7. RTSP

The Real Time Streaming Protocol RTSP is defined in RFC 2326 and provides a VCR like control for real time contents like audio/video on demand. RTSP is mainly based on HTTP/1.1, i.e. it has a client-server like structure, where a client requests information about a real time stream and starts and stops downloads (streams). Unlike HTTP/1.1, RTPS is state oriented, i.e. at all times, the protocol is in one of several possible states.

3.7.1. TERMINOLOGY

Some of the terminology has been adopted from HTTP/1.1. Terms not listed here are defined as in HTTP/1.1.

Aggregate control: The control of the multiple streams using a single timeline by the server. For audio/video feeds, this means that the client may issue a single play or pause message to control both the audio and video feeds.

Conference: a multiparty, multimedia presentation, where "multi" implies greater than or equal to one.

Client: The client requests continuous media data from the media server.

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

Container file: A file which may contain multiple media streams which often comprise a presentation when played together. RTSP servers may offer aggregate control on these files, though the concept of a container file is not embedded in the protocol.

Continuous media: Data where there is a timing relationship between source and sink; that is, the sink must reproduce the timing relationship that existed at the source. The most common examples of continuous media are audio and motion video.

Entity: The information transferred as the payload of a request or response. An entity consists of metainformation in the form of entity-header fields and content in the form of an entity- body.

Media initialization: Datatype/codec specific initialization. This includes such things as clockrates, color tables, etc. Any transport-independent information which is required by a client for playback of a media stream occurs in the media initialization phase of stream setup.

Media parameter: Parameter specific to a media type that may be changed before or during stream playback.

Media server: The server providing playback or recording services for one or more media streams. Different media streams within a presentation may originate from different media servers. A media server may reside on the same or a different host as the web server the presentation is invoked from.

Media server indirection: Redirection of a media client to a different media server.

(Media) stream: A single media instance, e.g., an audio stream or a video stream as well as a single whiteboard or shared application group. When using RTP, a stream consists of all RTP and RTCP packets created by a source within an RTP session.

Message: The basic unit of RTSP communication, consisting of a structured sequence of octets.

Participant: Member of a conference. A participant may be a machine, e.g., a media record or playback server.

Presentation: A set of one or more streams presented to the client as a complete media feed. In most cases in the RTSP context, this implies aggregate control of those streams, but does not have to.

Presentation description: A presentation description contains information about one or more media streams within a presentation, such as the set of encodings, network addresses and information about the content. Other IETF protocols such as SDP (RFC 2327) use the term "session" for a live presentation. The presentation description may take several different formats, including but not limited to the session description format SDP.

Response: An RTSP response. If an HTTP response is meant, that is indicated explicitly.

Request: An RTSP request. If an HTTP request is meant, that is indicated explicitly.

RTSP session: A complete RTSP "transaction", e.g., the viewing of a movie. A session typically consists of a client setting up a transport mechanism for the continuous media stream (SETUP), starting the stream with PLAY or RECORD, and closing the stream with TEARDOWN.

Transport initialization: The negotiation of transport information (e.g., port numbers, transport protocols) between the client and the server.

3.7.2. RTSP STATES

RTSP controls a stream which may be sent via a separate protocol, independent of the control channel. For example, RTSP control may occur on a TCP connection while the data flows via UDP. Thus, data delivery continues even if no RTSP requests are received by the media server. Also, during its lifetime, a single media stream may be controlled by RTSP requests issued sequentially on different TCP connections. Therefore, the server needs to maintain "session state" to be able to correlate RTSP requests with a stream Many methods in RTSP do not contribute to state. However, the following play a central role in defining the allocation and usage of stream resources on the server: SETUP, PLAY, RECORD, PAUSE, and TEARDOWN.

- SETUP: Causes the server to allocate resources for a stream and start an RTSP session.
- PLAY and RECORD: Starts data transmission on a stream allocated via SETUP.
- PAUSE: Temporarily halts a stream without freeing server resources.
- TEARDOWN: Frees resources associated with the stream. The RTSP session ceases to exist on the server.

RTSP methods that contribute to state use the Session header field to identify the RTSP session whose state is being manipulated. The server generates session identifiers in response to SETUP requests.

3.7.3. REQUESTS

A request message from a client to a server or vice versa includes, within the first line of that message, the method to be applied to the resource, the identifier of the resource, and the protocol version in use.

Request-Line = *Method SP Request-URI SP RTSP-Version CRLF*

Request = *Request-Line *(general-header | request-header | entity-header) CRLF [message-body]*

Method = *"DESCRIBE" | "ANNOUNCE" | "GET_PARAMETER" | "OPTIONS" | "PAUSE" | "PLAY" | "RECORD" | "REDIRECT" |*

SETUP" | "SET_PARAMETER" | "TEARDOWN" | extension-method

extension-method = token

Request-URI = "" | absolute_URI*

*RTSP-Version = "RTSP" "/" 1*DIGIT "." 1*DIGIT*

3.7.4. RESPONSE

After receiving and interpreting a request message, the recipient responds with an RTSP response message.

*Response = Status-Line *(general-header | response-header | entity-header) CRLF [message-body]*

The first line of a Response message is the Status-Line, consisting of the protocol version followed by a numeric status code, and the textual phrase associated with the status code, with each element separated by SP characters. No CR or LF is allowed except in the final CRLF sequence.

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

The Status-Code element is a 3-digit integer result code of the attempt to understand and satisfy the request. The Reason-Phrase is intended to give a short textual description of the Status-Code. The Status-Code is intended for use by automata and the Reason-Phrase is intended for the human user. The client is not required to examine or display the Reason-Phrase.

3.7.5. RTSP METHODS

In this section, the most important RTSP methods will be described.

DESCRIBE

The DESCRIBE method retrieves the description of a presentation or media object identified by the request URL from a server. The DESCRIBE reply-response pair constitutes the media initialisation phase of RTSP. Example:

C->S: DESCRIBE rtsp://server.example.com/fizzle/foo RTSP/1.0

CSeq: 312

Accept: application/sdp, application/rtsl, application/mheg

S->C: RTSP/1.0 200 OK

CSeq: 312

Date: 23 Jan 1997 15:35:06 GMT

Content-Type: application/sdp

Content-Length: 376

v=0

o=mhandley 2890844526 2890842807 IN IP4 126.16.64.4
s=SDP Seminar
i=A Seminar on the session description protocol
u=http://www.cs.ucl.ac.uk/staff/M.Handley/sdp.03.ps
e=mjh@isi.edu (Mark Handley)
c=IN IP4 224.2.17.12/127
t=2873397496 2873404696
a=recvonly
m=audio 3456 RTP/AVP 0
m=video 2232 RTP/AVP 31
m=whiteboard 32416 UDP WB
a=orient:portrait

SETUP:

The SETUP request for a URI specifies the transport mechanism to be used for the streamed media. The Transport header specifies the transport parameters acceptable to the client for data transmission; the response will contain the transport parameters selected by the server.

C->S: SETUP rtsp://example.com/foo/bar/baz.rm RTSP/1.0
CSeq: 302
Transport: RTP/AVP;unicast;client_port=4588-4589

S->C: RTSP/1.0 200 OK
CSeq: 302
Date: 23 Jan 1997 15:35:06 GMT
Session: 47112344
Transport: RTP/AVP;unicast;
client_port=4588-4589;server_port=6256-6257

PLAY:

The PLAY method tells the server to start sending data via the mechanism specified in SETUP. A client MUST NOT issue a PLAY request until any outstanding SETUP requests have been acknowledged as successful. The PLAY request positions the normal play time to the beginning of the range specified and delivers stream data until the end of the range is reached.

C->S: PLAY rtsp://audio.example.com/twister.en RTSP/1.0
CSeq: 833
Session: 12345678
Range: smpte=0:10:20-;time=19970123T153600Z

S->C: RTSP/1.0 200 OK
CSeq: 833
Date: 23 Jan 1997 15:35:06 GMT
Range: smpte=0:10:22-;time=19970123T153600Z

PAUSE:

The PAUSE request causes the stream delivery to be interrupted (halted) temporarily. If the request URL names a stream, only playback and recording of that stream is halted. For example, for audio, this is equivalent to muting. If the request URL names a presentation or group of streams, delivery of all currently active streams within the presentation or group is halted. After resuming playback or recording, synchronization of the tracks MUST be maintained. Any server resources are kept, though servers MAY close the session and free resources after being paused for the duration specified with the timeout parameter of the Session header in the SETUP message. Example:

C->S: PAUSE rtsp://example.com/fizzle/foo RTSP/1.0

CSeq: 834

Session: 12345678

S->C: RTSP/1.0 200 OK

CSeq: 834

Date: 23 Jan 1997 15:35:06 GMT

TEARDOWN:

The TEARDOWN request stops the stream delivery for the given URI, freeing the resources associated with it. If the URI is the presentation URI for this presentation, any RTSP session identifier associated with the session is no longer valid. Unless all transport parameters are defined by the session description, a SETUP request has to be issued before the session can be played again. Example:

C->S: TEARDOWN rtsp://example.com/fizzle/foo RTSP/1.0

CSeq: 892

Session: 12345678

S->C: RTSP/1.0 200 OK

CSeq: 892

3.7.6. MODELLING RTSP

When modelling RTSP, the natural sequence of requests will be followed:

1. SETUP
2. PLAY
3. TEARDOWN

After TEARDOWN, the application will close and the user will do something else.

3.8. RTP

The Real Time Protocol RTP is defined in RFC 1889 and is meant for the delivery of real time data such as videoconferencing data consisting of audio and video data. RTP allows either unicast or multicast services and is augmented by a control protocol called the real time control protocol RTCP to allow monitoring of the delivery. There is no guarantee of QoS in RTP. Furthermore, RTP does not guarantee delivery or prevents out-of-order delivery, nor does it assume that the underlying network is reliable and delivers packets in order. The sequence numbers included in RTP allow the receiver to reconstruct the sender's packet sequence, but sequence numbers might also be used to determine the proper location of a packet, for example in video decoding, without necessarily decoding packets in sequence

RTP itself is not a complete protocol but rather a framework left open for application specific extensions. Therefore, in addition to this document, a complete specification of RTP for a particular application will require one or more companion documents:

- a profile specification document, which defines a set of payload type codes and their mapping to payload formats (e.g., media encodings). A profile may also define extensions or modifications to RTP that are specific to a particular class of applications. Typically an application will operate under only one profile. A profile for audio and video data may be found in the companion RFC TBD.
- payload format specification documents, which define how a particular payload, such as an audio or video encoding, is to be carried in RTP.

3.8.1. DEFINITIONS

RTP payload: The data transported by RTP in a packet, for example audio samples or compressed video data. The payload format and interpretation are beyond the scope of this document.

RTP packet: A data packet consisting of the fixed RTP header, a possibly empty list of contributing sources (see below), and the payload data. Some underlying protocols may require an encapsulation of the RTP packet to be defined. Typically one packet of the underlying protocol contains a single RTP packet, but several RTP packets may be contained if permitted by the encapsulation method.

RTCP packet: A control packet consisting of a fixed header part similar to that of RTP data packets, followed by structured elements that vary depending upon the RTCP packet type. The formats are defined in Section 6. Typically, multiple RTCP packets are sent together as a compound RTCP packet in a single packet of the underlying protocol; this is enabled by the length field in the fixed header of each RTCP packet.

Port: The "abstraction that transport protocols use to distinguish among multiple destinations within a given host computer. TCP/IP protocols identify ports using small positive integers." The transport selectors (TSEL) used by the OSI transport layer are equivalent to ports. RTP depends upon the lower-layer protocol to provide some mechanism such as ports to multiplex the RTP and RTCP packets of a session.

Transport address: The combination of a network address and port that identifies a transport-level endpoint, for example an IP address and a UDP port. Packets are transmitted from a source transport address to a destination transport address.

RTP session: The association among a set of participants communicating with RTP. For each participant, the session is defined by a particular pair of destination transport addresses (one network address plus a port pair for RTP and RTCP). The destination transport address pair may be common for all participants, as in the case of IP multicast, or may be different for each, as in the case of individual unicast network addresses plus a common port pair. In a multimedia session, each medium is carried in a separate RTP session with its own RTCP packets. The multiple RTP sessions are distinguished by different port number pairs and/or different multicast addresses.

Synchronisation source (SSRC): The source of a stream of RTP packets, identified by a 32-bit numeric SSRC identifier carried in the RTP header so as not to be dependent upon the network address. All packets from a synchronisation source form part of the same timing and sequence number space, so a receiver groups packets by synchronisation source for playback. Examples of synchronisation sources include the sender of a stream of packets derived from a signal source such as a microphone or a camera, or an RTP mixer (see below). A synchronisation source may change its data format, e.g., audio encoding, over time. The SSRC identifier is a randomly chosen value meant to be globally unique within a particular RTP session. A participant need not use the same SSRC identifier for all the RTP sessions in a multimedia session; the binding of the SSRC identifiers is provided through RTCP. If a participant generates multiple streams in one RTP session, for example from separate video cameras, each must be identified as a different SSRC.

Contributing source (CSRC): A source of a stream of RTP packets that has contributed to the combined stream produced by an RTP mixer (see below). The mixer inserts a list of the SSRC identifiers of the sources that contributed to the generation of a particular packet into the RTP header of that packet. This list is called the CSRC list. An example application is audio conferencing where a mixer indicates all the talkers whose speech was combined to produce the outgoing packet, allowing the receiver to indicate the current talker, even though all the audio packets contain the same SSRC identifier (that of the mixer).

End system: An application that generates the content to be sent in RTP packets and/or consumes the content of received RTP packets. An end system can act as one or more synchronisation sources in a particular RTP session, but typically only one.

Mixer: An intermediate system that receives RTP packets from one or more sources, possibly changes the data format, combines the packets in some manner and then forwards a new RTP packet. Since the timing among multiple input sources will not generally be synchronised, the mixer will make timing adjustments among the streams and generate its own timing for the combined stream. Thus, all data packets originating from a mixer will be identified as having the mixer as their synchronisation source. Translator: An intermediate system that forwards RTP packets with their synchronisation source identifier intact. Examples of translators include devices that convert encodings without mixing, replicators from multicast to unicast, and application- level filters in firewalls.

Monitor: An application that receives RTCP packets sent by participants in an RTP session, in particular the reception reports, and estimates the current quality of service for distribution monitoring, fault diagnosis and long-term statistics. The monitor function is likely to be built into the application(s) participating in the session, but may also be a separate application that does not otherwise participate and does not send or receive the RTP data packets. These are called third party monitors.

Non-RTP means: Protocols and mechanisms that may be needed in addition to RTP to provide a usable service. In particular, for multimedia conferences, a conference control application may distribute multicast addresses and keys for encryption, negotiate the encryption algorithm to be used, and define dynamic mappings between RTP payload type values and the payload formats they represent for formats that do not have a predefined payload type value. For simple applications, electronic mail or a conference database may also be used. The specification of such protocols and mechanisms is outside the scope of this document.

Wallclock time (absolute time) is represented using the timestamp format of the Network Time Protocol (NTP), which is in seconds relative to 0h UTC on 1 January 1900 [5]. The full resolution NTP timestamp is a 64-bit unsigned fixed-point number with the integer part in the first 32 bits and the fractional part in the last 32 bits. In some fields where a more compact representation is appropriate, only the middle 32 bits are used; that is, the low 16 bits of the integer part and the high 16 bits of the fractional part. The high 16 bits of the integer part must be determined independently.

Byte order is big-endian.

3.8.2. RTP SCENARIOS

RTP delivery can be based on IP unicasting between a pair of participants or IP multicasting, if supported by the underlying network.

In an audio conference using unicasting, all participants will reserve two UDP ports for the conference, one for the RTP data packets, the other for the RTCP control packets. In this case, the application must maintain a list of IP addresses of all participants and ports, and must send the same RTP packet to all participants separately.

In an audio conference using multicasting, one conference manager will obtain an IP multicast address and two ports for it, again one for the RTP data packets, the other for the RTCP control packets. This information will be distributed amongst all participants.

In an audio/video conference, there will be a separation between the audio stream and the video stream. Both streams will use two different RTP sessions, being sent to two different port pairs. Additionally, two different IP multicasting addresses might be used for the two sessions, the conference is transported via multicasting.

3.8.3. RTP DATA TRANSFER PROTOCOL

RTP is used to transport the actual data. It consists of a header carrying sequence and timing information, followed by the real-time data.

3.8.3.1.RTP FIXED HEADER FIELDS

The RTP header has the following format:

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|X|  CC   |M|     PT      |       sequence number         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           timestamp                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           synchronisation source (SSRC) identifier            |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|            contributing source (CSRC) identifiers             |
|                             ....                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3 : RTP packet format.

The first twelve octets are present in every RTP packet, while the list of CSRC identifiers is present only when inserted by a mixer. The fields have the following meaning:

version (V): 2 bits

> This field identifies the version of RTP. The version defined by this specification is two (2). (The value 1 is used by the first draft version of RTP and the value 0 is used by the protocol initially implemented in the "vat" audio tool.)

padding (P): 1 bit

> If the padding bit is set, the packet contains one or more additional padding octets at the end which are not part of the payload. The last octet of the padding contains a count of how many padding octets should be ignored. Padding may be needed by some encryption algorithms with fixed block sizes or for carrying several RTP packets in a lower-layer protocol data unit.

extension (X): 1 bit

> If the extension bit is set, the fixed header is followed by exactly one header extension.

CSRC count (CC): 4 bits

> The CSRC count contains the number of CSRC identifiers that follow the fixed header.

marker (M): 1 bit

> The interpretation of the marker is defined by a profile. It is intended to allow significant events such as frame boundaries to be marked in the packet stream. A profile may define additional marker bits or specify that there is no marker bit by changing the number of bits in the payload type field.

payload type (PT): 7 bits

This field identifies the format of the RTP payload and determines its interpretation by the application. A profile specifies a default static mapping of payload type codes to payload formats. Additional payload type codes may be defined dynamically through non-RTP means. An initial set of default mappings for audio and video is specified in the companion profile Internet-Draft draft-ietf-avt-profile, and may be extended in future editions of the Assigned Numbers RFC. An RTP sender emits a single RTP payload type at any given time; this field is not intended for multiplexing separate media streams.

sequence number: 16 bits

The sequence number increments by one for each RTP data packet sent, and may be used by the receiver to detect packet loss and to restore packet sequence. The initial value of the sequence number is random (unpredictable) to make known-plaintext attacks on encryption more difficult, even if the source itself does not encrypt, because the packets may flow through a translator that does. Techniques for choosing unpredictable numbers are discussed in.

timestamp: 32 bits

The timestamp reflects the sampling instant of the first octet in the RTP data packet. The sampling instant must be derived from a clock that increments monotonically and linearly in time to allow synchronisation and jitter calculations. The resolution of the clock must be sufficient for the desired synchronisation accuracy and for measuring packet arrival jitter (one tick per video frame is typically not sufficient). The clock frequency is dependent on the format of data carried as payload and is specified statically in the profile or payload format specification that defines the format, or may be specified dynamically for payload formats defined through non-RTP means. If RTP packets are generated periodically, the nominal sampling instant as determined from the sampling clock is to be used, not a reading of the system clock. As an example, for fixed-rate audio the timestamp clock would likely increment by one for each sampling period. If an audio application reads blocks covering 160 sampling periods from the input device, the timestamp would be increased by 160 for each such block, regardless of whether the block is transmitted in a packet or dropped as silent.

The initial value of the timestamp is random, as for the sequence number. Several consecutive RTP packets may have equal timestamps if they are (logically) generated at once, e.g., belong to the same video frame. Consecutive RTP packets may contain timestamps that are not monotonic if the data is not transmitted in the order it was sampled, as in the case of MPEG interpolated video frames. (The sequence numbers of the packets as transmitted will still be monotonic.)

SSRC: 32 bits

The SSRC field identifies the synchronisation source. This identifier is chosen randomly, with the intent that no two synchronisation sources within the same RTP session will have the same SSRC identifier. Although the probability of multiple sources choosing the same identifier is low, all RTP implementations must be prepared to detect and resolve collisions. If a source changes its source transport address, it must also choose a new SSRC identifier to avoid being interpreted as a looped source.

CSRC list: 0 to 15 items, 32 bits each

The CSRC list identifies the contributing sources for the payload contained in this packet. The number of identifiers is given by the CC field. If there are more than 15 contributing sources, only 15 may be identified. CSRC identifiers are inserted by mixers, using the SSRC identifiers of contributing sources. For example, for audio packets the SSRC identifiers of all sources that were mixed together to create a packet are listed, allowing correct talker indication at the receiver.

3.8.3.2. PROFILE-SPECIFIC MODIFICATIONS TO THE RTP HEADER

The existing RTP data packet header is believed to be complete for the set of functions required in common across all the application classes that RTP might support. However, in keeping with the ALF design principle, the header may be tailored through modifications or additions defined in a profile specification while still allowing profile-independent monitoring and recording tools to function.

If the X-bit in the RTP header is one, a variable-length header extension is appended to the RTP header. The header extension contains a 16-bit length field that counts the number of 32-bit words in the extension.

3.8.4. THE RTP CONTOL PROTOCOL RTCP

The RTP Control Protocol is based on the periodic transmission of control packets to all participants in the session.

RTCP provides feedback of the data connection quality. To do this, both senders and receivers periodically send reports to all other session members. As all senders and receivers send such reports, there are ways included to scale the report transmission in case too many participants take part in the session. Finally, RTCP carries information such as the CNAME for permanent identification of participants, as well as minimal session control information to allow participants to enter and leave sessions.

3.8.4.1. THE RTCP PACKETS

RTCP packets are required to carry the following information:

SR: Sender report, for transmission and reception of statistics from participants that are active senders.

RR: Receiver report, for reception statistics from participants that are not active senders.

SDES: Source description items.

BYE: Indicates end of participation.

APP: Application specific function.

Each RTCP packet starts with a fixed header and may then be followed by variable length parts. Furthermore, multiple RTCP packets may be concatenated to form a compound packet that can be sent in one PDU of the lower transport protocol such as UDP. The following restrictions apply for compound RTCP packets:

- The first RTCP packet in a compound packet must always be a report packet.

- If the number of sources for which reception statistics are being reported exceeds 31, then additional RR packets should follow the initial report packet.
- SDES: An SDES packet including the CNAME item must be included in each compound RTCP packet.
- Other RTCP packet types may follow in any order, except that BYE packets should be the last packet.

3.8.4.2.RTCP TRANSMISSION INTERVAL

As the number of sent RTCP packets increases with the number of participants, a control mechanism must influence the rate at which RTCP packets are sent.

For each session, it is assumed that the data traffic is subject to an aggregate limit called the session bandwidth to be divided amongst the participants. This bandwidth might be reserved or it might be just a reasonable share. The session bandwidth parameter is expected to be supplied by a session management application when it invokes a media application. Session bandwidth calculations will also include lower-layer transport and control protocols such as UDP and IP.

The idea now is to provide a fixed percentage of the session bandwidth to RTCP packets, as a default value, the standard uses 5 percent. An algorithm then must calculate the rate at which RTCP packets are created. This rate must be the same for all participants. The following restrictions also apply:

- Sender are collectively allocated ¼ of the control traffic.
- The interval must be greater than 5 seconds.
- The interval between RTCP packets is varied randomly over the range [0.5, 1.5] times the calculated interval. The first RTCP packet of a session also delayed by a random variation of half the minimum RTCP interval.
- A dynamic estimate of the average compound RTCP packet size is calculated, including all those received and sent.

As the number of participants is an important parameter for the calculation of the RTCP interval, new sites must be added to the count as soon as they have sent packets including their SSRC or CSRC.

3.8.4.3.SENDER AND RECEIVER REPORTS

Both SR and RR start with a 8 bytes long fixed header holding the following fields:

version (V): 2 bits

> Identifies the version of RTP, which is the same in RTCP packets as in RTP data packets. The version defined by this specification is two (2).

padding (P): 1 bit

> If the padding bit is set, this RTCP packet contains some additional padding octets at the end which are not part of the control information. The last octet of the padding is a count of how many padding octets should be ignored. Padding may be needed by some encryption algorithms with fixed block sizes. In a

compound RTCP packet, padding should only be required on the last individual packet because the compound packet is encrypted as a whole.

reception report count (RC): 5 bits

The number of reception report blocks contained in this packet. A value of zero is valid.

packet type (PT): 8 bits

Contains the constant 200 to identify this as an RTCP SR packet.

length: 16 bits

The length of this RTCP packet in 32-bit words minus one, including the header and any padding. (The offset of one makes zero a valid length and avoids a possible infinite loop in scanning a compound RTCP packet, while counting 32-bit words avoids a validity check for a multiple of 4.)

SSRC: 32 bits

The synchronization source identifier for the originator of this SR packet.

The only difference between SR and RR packets is the fact that SR packets include a 20-byte sender information section. The SR is issued if a site has sent any data packets during the interval since issuing the last report or the previous one, otherwise the RR is issued:

NTP timestamp: 64 bits

Indicates the wallclock time when this report was sent so that it may be used in combination with timestamps returned in reception reports from other receivers to measure round-trip propagation to those receivers.

RTP timestamp: 32 bits

Corresponds to the same time as the NTP timestamp (above), but in the same units and with the same random offset as the RTP timestamps in data packets. This correspondence may be used for intra- and inter-media synchronisation for sources whose NTP timestamps are synchronised, and may be used by media-independent receivers to estimate the nominal RTP clock frequency.

sender’s packet count: 32 bits

The total number of RTP data packets transmitted by the sender since starting transmission up until the time this SR packet was generated.

sender’s octet count: 32 bits

The total number of payload octets (i.e., not including header or padding) transmitted in RTP data packets by the sender since starting transmission up until the time this SR packet was generated. The count is reset if the sender changes its SSRC identifier. This field can be used to estimate the average payload data rate.

The SR and RR will include zero or more reception report blocks for each of the SSRC from which this receiver has received RTP data packets since the last report. CSRC are not included. Each reception report block consists of 24 bytes holding the following fields:

SSRC_n (source identifier): 32 bits

The SSRC identifier of the source to which the information in this reception report block pertains.

fraction lost: 8 bits

The fraction of RTP data packets from source SSRC_n lost since the previous SR or RR packet was sent, expressed as a fixed point number with the binary point at the left edge of the field.

cumulative number of packets lost: 24 bits

The total number of RTP data packets from source SSRC_n that have been lost since the beginning of reception.

extended highest sequence number received: 32 bits

The low 16 bits contain the highest sequence number received in an RTP data packet from source SSRC_n, and the most significant16 bits extend that sequence number with the corresponding count of sequence number cycles.

interarrival jitter: 32 bits

An estimate of the statistical variance of the RTP data packet interarrival time, measured in timestamp units and expressed as an unsigned integer.

3.8.4.4.SOURCE DESCRIPTION RTCP PACKETS

Source description RTCP packets (SDES) is a three-level structure composed of a 4-byte header and zero or more chunks, each of which is composed of items describing the source identified. Each chunk consists of an SSRC/CSRC identifier followed by a list of zero or more items, which carry information about the SSRC/CSRC. Each item starts at a 32-bit boundary with an 8-bit type field, an 8-bit octet count describing the length of the text, and the text itself. Item types include CNAME, NAME, EMAIL, PHONE and others.

3.8.4.5. BYE: GOODBYE RTCP PACKET

The BYE packet consists of a 4-byte fixed header, followed by one or more SSRC/CSRC identifiers that will be removed from the session, followed by an optional text describing the removal.

3.8.5. RTP TRANSLATORS

A translator connects two or more transport level clouds. It forwards all RTP packets with their SSRC intact. Some kinds of translators will forward the RTP data untouched, while others will pass through the data and change the encoding.

3.8.6. RTP MIXERS

A mixer receives streams of RTP data packets from one or more sources, possibly changes the data format, combines the streams in some manner and then forwards the combined stream. Since the timing among multiple sources in general will not be synchronised, the mixer will produce its own timing for the combined stream, representing a synchronisation source of its own. Thus, all data packets forwarded by the mixer will carry the SSRC identifier of the mixer. The mixer though will add the SSRC of the individual sources as CSRC into the packets.

Mixers may also be cascaded, thus concatenating several CSRC lists into one.

3.8.7. MODELLING RTP SESSIONS

RTP session will be used in two possible scenarios:

- Live Audio/Video conference.
- Audio/Video on demand (See RTSP).

RTP will primarily be used in connection with live audio/video conferences over the Mbone. Audio/video on demand will be modelled by the RealServer SureStream protocol.

Modelling RTP sessions will be done by first creating sessions and session identifiers SSRC. Based upon some default values or initial measurements, the session bandwidth will be calculated, and 5 percent of this bandwidth will be devoted to RTCP, thus computing the RTCP interval with the algorithm stated in RFC 1889. Both session bandwidth and number of participants can change throughout the session.

RTP packets will be created according to some packet creating point process. The process will be aware of the available bandwidth and will encode its content accordingly. Audio codecs will sample audio data at a fixed sampling rate, thus keeping the data rate constant is inherent to audio streaming. Video codecs, on the other hand, will buffer frames and will change the quantization parameter to achieve a fixed data rate. Here, the traffic will have a larger variance compared to audio traffic. The overall rate, though, must not exceed the limit as imposed by the session bandwidth. The packet sizes will be created by a second process, including observed autocorrelations.

3.9. SURESTREAM

The RealServer provides audio/video on demand for access over the internet. As data delivery protocol, RealServer uses SureStream, to access this media, the newer versions provide a RTSP interface. SureStream has the following properties:

- Creates one file for all connection rate environments
- Streams to different connection rates in a heterogeneous environment
- Seamlessly switches to different streams based on changing network conditions
- Prioritizes key frames and audio data over partial frame data
- Streams backward compatible files to older RealPlayers
- Thins video key frames for the lowest possible data rate throughput
- Supports Mbone over Intranet and Internet (detected automatically)
- Supports hierarchical streams over splitters.

When encoding, the user must select one or several of a list of available bitrates. RealServer then encodes the sampled audio/video signal for all selected bitrates and saves them in one file only. The used encoders are:

- RealVideo: for low bandwidth video
- RealVideo Fractal: for high bandwidth video
- RealAudio: Uses different encoders at 6.5, 8, 8.5, 12 and 20 kb/sec. The codecs are meant for different contents, for example voice only, voice with background music, stereo music and so on.

When downstreaming the content, at first, the player at the client host has a default value for its normal bitrate and a value for its maximum birate. The client first downloads a few seconds of content encoded for the normal bitrate without playing the content and fills a buffer with it. Also, the observed bitrate and packet error rate are sampled at this time.

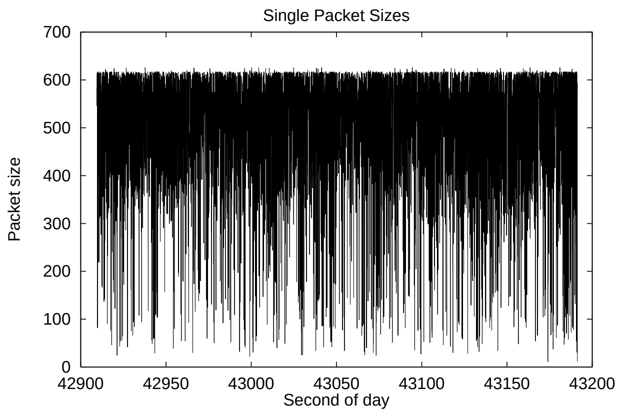
When the buffer is full, enough statistical data has been sampled to define the current available bitrate and error probability. At the RealServer Adaptive Stream Management (ASM) part, rules are defined such as “if the observed bitrate is between 5kb and 15kb and the error probability is below 2.5%, choose the stream encoded with rate x kb. Based on the statistics, the RealServer then chooses a rule that is satisfied by the conditions and switches to the stream that is chosen by the rule. At all times, though, the average stream bitrate must not exceed the maximum bitrate chosen at the client.

During the download, this strategy is continued by continuously collecting statistics about the available bitrate and error probability. Based upon reports sent by the client, the RealServer will switch to higher or lower bitrate streams as appropriate.

In addition to the available bandwidth and error rate, the client will also watch its buffer. If it starts filling up, the bitrate at the server side must be decreased. If it gets empty, the bitrate should be increased. The length of the buffer, though, is directly linked to the available bandwidth. A dramatically dropping of the available bitrate will force the client to halt the presentation and to start filling up the buffer again.

An additional strategy to cope with dropping bandwidth due to congestion to to start thinning out the stream. In this strategy, the server will drop intermediate frames and will just send key frames. This way, a higher picture quality can be maintained, while the number of frames per second can be varied dynamically.

Packet sizes observed from downstreaming a clip from a RealServer can be seen in Figure 4. In the client, 112 Kb dual ISDN was set for normal and maximum bandwidth. The negotiated connection rate was 10KB (80Kb) per second. When observing each individual packet, no direct connection between the individual sizes can be seen. This is also demonstrated by the autocorrelation function seen in Figure 5.



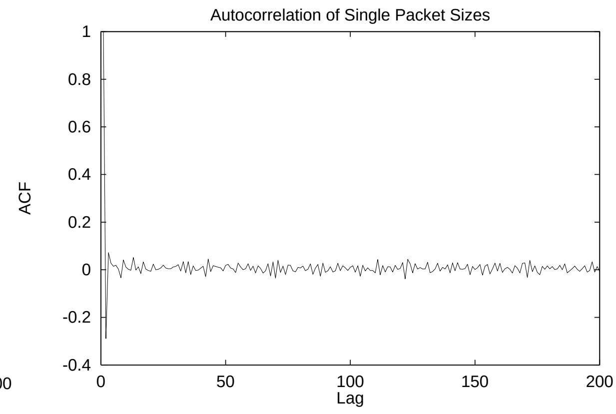
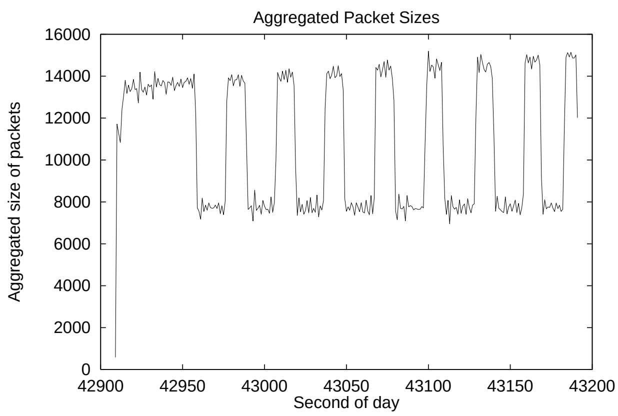


Figure 4 : Real Audio/Video traffic.

Figure 5 : Autocorrelation function.

Aggregating over one second shows a clear connection between the individual time slots can be seen in Figure 6 and Figure 7. It is obvious that SureStream chooses one of two possible encoded streams. The lower rate stream uses 60 Kb/sec, while the higher one uses 112 Kb/sec.



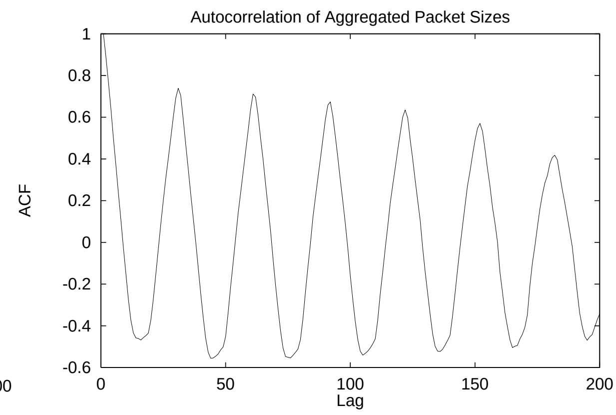


Figure 6 : Aggregated UDP packets.

Figure 7: ACF of aggregated traffic.

What cannot be seen in the plots is the traffic from client to server and on the RTSP tcp connection. Basically, the client acknowledges server data every second with a short UDP packet. Additionally, at all switching points, the client sends a short status update message to the server on the RTSP connection, that is in turn acknowledged with a small message. After this, the server switches to another stream rate.

3.9.1. MODELLING SURESTREAM

The SureStream server model will start streaming on a pre-defined bitrate stream. As there is no direct connection between the sizes of successing packets, the stochastic process will be time-aggregated over one second. For each time-slot, the process will produce a fixed size of transferred bits plus some minor stochastic variation with zero mean. A second process will do the same for the number of packets to send. The bits will then be equally distributed amongst the packets.

Modelling the SureStream Service must include a model for the rate adaption mechanism. The model will use a virtual buffer that holds audio/video data for several seconds. The buffer is first filled with data at rate *r* for at least 10 seconds.. The data in the buffer is then marked with rate *a*. The client model retrieves data from the buffer at a rate that is equal to that of the first data item.

The client periodically sends rate updates to the server on the RTSP connection. Whether the client sends this, if some internal limit is not met (smoothed data rate of the last *k* seconds), or according to an internal timer is not known right now. Upon the reception of such a message, the server acknowledges it and changes the stream rate. If the current bitrate is higher than the negotiated/available average bitrate, the server will drop its rate to the next lower available encoded bitrate. If the current rate is lower, the server will switch to the next higher bitrate.

Upon network congestion, the buffer starts emptying. If the buffer contents drop below a certain threshold, the client sends a status update to the server, which in turn will thin its traffic, i.e. will send only parts of its data/every *l* frame only.

If the buffer gets empty, the client stalls the presentation and starts refilling it for at least 10 seconds again.

The following table contains the RealServer target audience, together with the resulting average bitrate that is sent:

Target audience	Average bitrate
28K Modems	20 Kbps
56K Modems	34 Kbps
Single ISDN	45 Kbps
Dual ISDN	80 Kbps
xDSL/Cable Modem	220 Kbps
Corporate LAN	150-1024 Kbps

Table 1 : Average bitrates.

3.10. UNKNOWN PROTOCOLS

The measured logfiles will contain (aggregated) information about IP packets being sent from one host to another. For each such dataset, the port number at the destination host will also be known. This port number will eventually identify the protocol/application that generated the packet in the first place. When modelling such traffic, the approach will be twofold:

1. First, those ports are identified that contribute a significant amount of traffic. Modelling applications that are hardly used or do not generated notable traffic are regarded as unimportant and will be omitted.
2. Secondly, models for the protocols/applications generating traffic on the important ports will be derived

The models will fall into exactly one of several possible categories:

1. The protocol of the destination port is known: Here, a specific application model based on exactly this data will be derived. Consider for example the traffic measured on port 25 (SMTP). For this data, a point process will generate time

points of the arrival commands a user issues inside his/her mail client (application layer). At such time points, another point process will generate a sequence of command/reply messages (service layer). The length of these messages will be drawn from distributions generated from the measured data.

2. The protocol of the destination port is not known, but there is a certain application (type) that is known to use this port (like database accesses, game servers): Here, the same approach as in point 1 will be used.

3. The protocol of the destination port is not known and no application is known for it: For this kind of traffic, dummy applications will be created that just start a certain service that is not related to a specific protocol.. The service itself will generate messages to Internet hosts as measured in the logfiles.

3.11. PROCESS COMMUNICATION (CSCW)

Vienna/Solinet. Vienna models the Video/Audio part, Solinet should provide details of their CSCW application.

4. CONCLUSIONS

5. APPENDIX B: AUGMENTED BNF

Some mechanisms specified in this document are described in an augmented Backus-Naur Form (BNF). Implementers will need to be familiar with the notation in order to understand this specification.

5.1. BASIC NOTATION

The augmented BNF includes the following constructs:

name = definition

The name of a rule is simply the name itself (without any enclosing "<" and ">") and is separated from its definition by the equal character "=". White-space is only significant in that indentation of continuation lines is used to indicate a rule definition that spans more than one line. Certain basic rules are in uppercase, such as SP, LWS, HT, CRLF, DIGIT, ALPHA, etc. Angle brackets are used within definitions whenever their presence will facilitate discerning the use of rule names.

"literal"

Quotation marks surround literal text. Unless stated otherwise, the text is case-insensitive.

rule1 | rule2

Elements separated by a bar ("I") are alternatives, e.g., "yes | no" will accept yes or no.

(rule1 rule2)

Elements enclosed in parentheses are treated as a single element. Thus, "(elem (foo | bar) elem)" allows the token sequences "elem foo elem" and "elem bar elem".

*rule

The character "*" preceding an element indicates repetition. The full form is "<n>*<m>element" indicating at least <n> and at most <m> occurrences of element. Default values are 0 and infinity so that "*(element)" allows any number, including zero; "1*element" requires at least one; and "1*2element" allows one or two.

[rule]

Square brackets enclose optional elements; "[foo bar]" is equivalent to "*1(foo bar)".

N rule

Specific repetition: "<n>(element)" is equivalent to "<n>*<n>(element)"; that is, exactly <n> occurrences of (element). Thus 2DIGIT is a 2-digit number, and 3ALPHA is a string of three alphabetic characters.

#rule

A construct "#" is defined, similar to "*", for defining lists of elements. The full form is "<n>#<m>element" indicating at least <n> and at most <m> elements, each separated by one or more commas (",") and optional linear whitespace (LWS). This makes the usual form of lists very easy; a rule such as "(*LWS element *(*LWS "," *LWS element))" can be shown as "1#element". Wherever this construct is used, null elements are allowed, but do not contribute to the count of elements present. That is, "(element), , (element)" is permitted, but counts as only two elements. Therefore, where at least one element is required, at least one non-null element must be present. Default values are 0 and infinity so that "#(element)" allows any number, including zero; "1#element" requires at least one; and "1#2element" allows one or two.

; comment

A semi-colon, set off some distance to the right of rule text, starts a comment that continues to the end of line. This is a simple way of including useful notes in parallel with the specifications.

implied *LWS

The grammar described by this specification is word-based. Except where noted otherwise, linear whitespace (LWS) can beincluded between any two adjacent words (token or quoted-string), and between adjacent tokens and delimiters (tspecials), without changing the interpretation of a field. At least one delimiter (tspecials) must exist between any two tokens, since they would otherwise be interpreted as a single token.

5.2. BASIC RULES

The following rules are used to describe basic parsing constructs.

```
OCTET          =  <any 8-bit sequence of data>
CHAR           =  <any US-ASCII character (octets 0 - 127)>
UPALPHA        =  <any US-ASCII uppercase letter "A".."Z">
LOALPHA        =  <any US-ASCII lowercase letter "a".."z">
ALPHA          =  UPALPHA | LOALPHA
DIGIT          =  <any US-ASCII digit "0".."9">
CTL            =  <any US-ASCII control character(octets 0 - 31) and DEL (127)>
CR             =  <US-ASCII CR, carriage return (13)>
LF             =  <US-ASCII LF, linefeed (10)>
SP             =  <US-ASCII SP, space (32)>
HT             =  <US-ASCII HT, horizontal-tab (9)>
<">            =  <US-ASCII double-quote mark (34)>
CRLF           =  CR LF
LWS            =  [CRLF] 1*( SP | HT )
TEXT           =  <any OCTET except CTLs, but including LWS>
HEX            =  "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" | "d" | "e" | "f" | DIGIT
word           =  token | quoted-string
token          =  1*<any CHAR except CTLs or tspecials>
tspecials      =  "(" | ")" | "<" | ">" | "@" | "," | ";" | ":" | "\" | <"> | "/" | "[" | "]" | "?" | "=" | "{" | "}" |
                  SP | HT
comment        =  "(" *( ctext | comment ) ")"
ctext          =  <any TEXT excluding "(" and ")">
quoted-string  =  ( <"> *(qdtext) <"> )
qdtext         =  <any CHAR except <"> and CTLs, but including LWS>
```

5.3. UNIFORM RESOURCE IDENTIFIERS (URI)

```
URI           =   ( absoluteURI | relativeURI ) [ "#" fragment ]
absoluteURI   =   scheme ":" *( uchar | reserved )
relativeURI   =   net_path | abs_path | rel_path
net_path      =   "//" net_loc [ abs_path ]
abs_path      =   "/" rel_path
rel_path      =   [ path ] [ ";" params ] [ "?" query ]
path          =   fsegment *( "/" segment )
fsegment      =   1*pchar
segment       =   *pchar
params        =   param *( ";" param )
param         =   *( pchar | "/" )
scheme        =   1*( ALPHA | DIGIT | "+" | "-" | "." )
net_loc       =   *( pchar | ";" | "?" )
query         =   *( uchar | reserved )
fragment      =   *( uchar | reserved )
pchar         =   uchar | ":" | "@" | "&" | "=" | "+"
uchar         =   unreserved | escape
unreserved    =   ALPHA | DIGIT | safe | extra | national
escape        =   "%" HEX HEX
reserved      =   ";" | "/" | "?" | ":" | "@" | "&" | "=" | "+"
extra         =   "!" | "*" | "'" | "(" | ")" | ","
safe          =   "$" | "-" | "_" | "."
unsafe        =   CTL | SP | <"> | "#" | "%" | "<" | ">"
national      =   <any OCTET excluding ALPHA, DIGIT, reserved, extra, safe, and unsafe>
```

6. APPENDIX C: ABBREVIATIONS

Cache	A program´s local storage to temporarily store objects in case, they will be needed in the near future.
Multipurpose Internet Mail Extensions (MIME)	Standard describing data types in e-mail and news messages.
Uniform Resource Identifier (URI)	Unique identifier for a resource (file, server service, ...). In HTTP/1.0, URIs are just strings identifying the location, name or other characteristics of a resource in the net.
Uniform Resource Locator (URL)	Identifier for locating resources in an intranet or the Internet.

:.

:

: