NORDUGRID-TECH-9
17/6/2006

THE HTTP(S,G) AND SOAP SERVER/FRAMEWORK

Code and Usage Description

A.Konstantinov*

*aleks@fys.uio.no

Contents

1 Introduction

The HTTP SOAP framework (httpsd) is a set of C++ classes and code to make it easier to write SOAP over HTTP over GSI or SSL services. Pure HTTP is also possible. As result it includes HTTP(S,G) server.

The code is provided as part of NorduGrid ARC software and uses some shared pieces of code, including third-party software. It can be obtained from `http://ftp.nordugrid.org/download/` by downloading any source package currently available. Required third-party software include Globus Toolkit® [1](for globus_io), gSOAP[2], VOMS(optional). One of included services uses GACL (included in sources).

The code builds into standalone server which may listen on 4 TCP/IP ports (2 by default) for incoming connections with different transport protocols and understands subset of HTTP wrapped with GSI (Globus and CERN implementations), SSL or pure HTTP.

There are following C++ classes availble:

- Server side:
  - HTTPS_Connector,
  - HTTP_Service,
  - HTTP_ServiceAdv.
- Client side
  - HTTP_Client
  - HTTP_ClientSOAP

2 Classes

2.1 HTTPS_Connector

Defined in connector.h

```
class HTTPS_Connector {
 public:
  unsigned int pid;
  /// Constructor
  /// @arg url - URL to which this connector is bound
  /// @arg serv - services to be served by this connector
  /// @arg auths - authorization rules to be processed
  /// @arg vos - VOs to be matched
  HTTP_Connector(const char* url,HTTP_Services& serv,std::list<AuthEvaluator*>&auths,std::list<Auth
  virtual ~HTTP_Connector(void);
  operator bool(void) { return initialized; };
  virtual size_t read(char* buf, size_t l);
  virtual int write(const char* buf, size_t l);
  virtual size_t readline(char* buf, size_t l);
  /// Called by loop_thread before loop(). loop_thread assumes user's
  /// identity was already established after loop_start() finished.
  virtual void loop_start(void);
  /// Called by loop_thread second (main function for processing requests)
  virtual void loop(void);
  /// Called by loop_thread after loop()
  virtual void loop_end(void);
  /// Start new thread for processing requests. Thread function is loop_thread()
  void loop_in_thread(void);
  /// Get remote subject (deprecated)
```

```
  const char* identity_subject(void) const { return user.DN(); };
  /// Get delegated proxy (deprecated)
  const char* identity_proxy(void) const { return user.proxy(); };
  /// Get remote identity
  AuthUser& identity(void) { return user; };
  /// Get authorization rules
  std::list<AuthEvaluator*>& authorizations(void) { return auths; };
  std::list<AuthVO>& VOs(void) { return vos; };
  const char* url(void) { return base_url.c_str(); };
  int send_file(const char* fname);
  int send_response_header(int keep_alive,int code,char* type,int size);
  int send_error_response(int keep_alive,int code,char* type,char* content);
  int skip_request(int &keep_alive);
  int skip_header(int &keep_alive);
};
```

It's purpose is to serve as a socket for accepting data from a client and to send a response from a server. This virtual class has implementations:

HTTP_Globus_Connector Globus GSI wrapper

HTTP_SSL_Connector OpenSSL wrapper

HTTP_Plain_Connector no wrapping

HTTP_GSSAPI_Connector CERN GSI implementation using GSSAPI functions directly

Corresponding classes for listening for incoming connections are available and inherited from

```
class HTTP_Listener {
 public:
  HTTP_Listener(const char* url,HTTP_Services& serv,std::list<AuthEvaluator*>& auths_,std::list<AuthV
  virtual ~HTTP_Listener() { };
  /// This method must be called on each new connection. Default
  /// behavior is to call callback if defined.
  /// Either rewrite this method or set callback.
  virtual void connect(void* arg) { if(cb) cb(*this,arg); };
  /// Set callback
  void callback(int (*cb_)(HTTP_Listener& l,void* arg)) { cb=cb_; };
  operator bool(void) { return initialized; };
};
```

Most important methods of *HTTPS_Connector* are:

size_t HTTP_Connector::read(char* buf, size_t l)

Reads at most **l** bytes into buffer **buf**. Returns number of read bytes. Returned 0 means it could not read data. This most probably happens due to closed connection.

int HTTP_Connector::write(const char* buf, size_t l)

Sends **l** bytes from buffer **buf** to network. Returns either 0 if data is sent or 1 ortherwise.

size_t HTTP_Connector::readline(char* buf, size_t l)

Reads line delimited by '\n' character. Characters '\n' and '\r' at end of line are stripped. Returns number of read characters.

void HTTP_Connector::loop(void)

Waits for a HTTP request coming from the open connection, initiates an instance of requested service and call corresponding methods. Exits after connection is closed.

int HTTP_Connector::skip_request(int &keep_alive)

reads and skips HTTP header and message body (if available). Variable *keep_alive* will be reset to 0 if information in header does not allow connection to continue.

int HTTP_Connector::skip_header(int &keep_alive)

skips HTTP header. Support for *keep_alive* currently is not implemented.

int HTTP_Connector::send_response_header(int keep_alive,int code,char* type,int size)

creates and sends response HTTP header incuding first line with response code provided in variable *code*. Variables *type* (if not NULL) and *size* (if not 0) are used to specify Content-Type and Content-Length accordingly. *keep_alive* informs client if server is willing to keep connection open.

int HTTP_Connector::send_file(const char* fname)

sends content of file named *fname* over open connection. Currently it is used to send error responses which contain user-readable information. But together with*send_response_header* and *stat_file* it can be used to implement minimalistic web server.

int HTTP_Connector::send_error_response(int keep_alive,int code,char* type,char* content)

sends response header containing response code *code* with Content-Type set to *type* and with body containing message in *content*. If *content* is NULL then file with name $ARC_LOCATION/share/error{value of *code*}.html is used for message body. Otherwise message is sent without body.

2.2 HTTP_Service

Defined in httpsd.h

```
class HTTP_Service {
 public:
  HTTP_Service(void);
  virtual ~HTTP_Service(void);
  virtual HTTP_Error get(const char* uri,int &keep_alive);
  virtual HTTP_Error put(const char* uri,int &keep_alive);
  virtual HTTP_Error post(const char* uri,int &keep_alive);
};
typedef enum {
  HTTP_OK = 200,
  HTTP_NOT_IMPLEMENTED = 501,
  HTTP_NOT_ALLOWED = 403,
  HTTP_NOT_FOUND = 404,
  HTTP_ERROR = 500,
  HTTP_FAILURE = -1
} HTTP_Error;
```

This is just a template for every service *instance* accessible through the HTTP_Connector. All functions return HTTP_NOT_IMPLEMENTED.

Implemented services must return HTTP_OK on success. Service is supposed to process and skip whole request (header and body) if it does not return HTTP_NOT_IMPLEMENTED or HTTP_NOT_FOUND. Otherwise calling HTTP_Connector will do that. Also service is supposed to send response to client by itself if it returned HTTP_OK or HTTP_FAILURE.

Following are function prototypes which are called by server code to configure server and to create service instance. Each service must have corresponding set of such functions.

```
typedef bool (*service_configurator)(istream& f,const char* uri, HTTP_Service_Properties &prop);
typedef HTTP_Service* (*service_creator)(HTTPS_Connector& c, const char* uri, void* arg);
class HTTP_Service_Properties {
 public:
  bool subtree;
  void* arg;
};
```

service_configurator is called during startup of server and is supposed to process configuration available through stream *f* and create service specific data structures. *uri* is URL for this particular service specified in server's configuration (can be relative). It should fill *prop* with information about service. Currently that is *subtree* which tells server code if this service is going to server all URLs starting from one specified in *uri*, and *arg* which should point to service specific information and is then passed to function responsible for creating service instances.

service_creator is called to create service instance when client requests that server. *c* is the HTTP_Connector transport class to be used for commnication with client, *uri* contains URL used to call service (absolute) and *arg* is the one filled by *service_configurator*.

2.3 HTTP_ServiceData and HTTP_ServiceAdv

Defined in service_soap.h there are 2 classes with methods which helps to write real services. HTTP_ServiceData provides method useful for transfering data

```
class HTTP_ServiceData:public HTTP_Service {
 protected:
  HTTP_Connector *c;
  // HTTP Header
  uint64_t range_start[MAX_RANGES];
  uint64_t range_end[MAX_RANGES];
  uint64_t entity_range_start;
  uint64_t entity_range_end;
  uint64_t entity_size;
  HTTP_Time entity_last_modified;
  HTTP_Time entity_expires;
  int nranges;
  bool range_passed;
  bool failure_parsing;
  uint64_t length;
  bool length_passed;
  bool entity_range_passed;
  bool entity_size_passed;
  bool entity_last_modified_passed;
  bool entity_expires_passed;
  bool unsupported_option_passed;
  virtual HTTP_Error parse_header(int &keep_alive);
  virtual HTTP_Error send_header(int &keep_alive,int code = 200);
  HTTP_Error send_header(int &keep_alive,uint64_t start,uint64_t end,bool partia
```

```
l,uint64_t full_size,const HTTP_Time& expires,const HTTP_Time& last_modified);
 public:
  HTTP_ServiceData(HTTP_Connector *c_);
  virtual ~HTTP_ServiceData(void);
};
```

It extracts information essential for data exchange from HTTP header and forms HTTP header to be sent to client.

HTTP_ServiceAdv helps to integrate gSOAP infrastructure. It provides functions for gSOAP for message transport. It also takes care of linking to gSOAP struct soap (*sp*).

```
class HTTP_ServiceAdv:public HTTP_ServiceData {
 protected:
  // SOAP
  bool ignore_soap_output;
  struct soap sp;
  struct Namespace* namespaces;
  char soap_fbuf[1024];
  int soap_fbuf_n;
  static int soap_fsend(struct soap *sp, const char* buf, size_t l);
  static int soap_flush(struct soap *sp);
  static size_t soap_frecv(struct soap* sp, char* buf, size_t l);
  static int soap_fopen(struct soap*, const char*, const char*, int);
  static int soap_fclose(struct soap*);
  static int soap_parse(struct soap *sp);
  void soap_init(void);
  void soap_deinit(void);
  virtual HTTP_Error post(const char* uri,int &keep_alive);
  HTTP_Error soap_post(const char* uri,int &keep_alive);
  virtual void soap_methods(void);
  void add_namespaces(struct Namespace* namespaces);
 public:
  HTTP_ServiceAdv(HTTP_Connector *c_);
  virtual ~HTTP_ServiceAdv(void);
};
```

HTTP_ServiceAdv SOAP capabilities

If You want Your service to use SOAP then it must:

- call *soap_init* in constructor and then set sp.namespaces to namespaces of Your SOAP methods and sp.user to pointer to pointer to service (this will be changed in a future),
- call *soap_deinit* in destructor,
- call *soap_post* in *post* method after processing HTTP header (You can use *parse_header* for that),
- implement *soap_methods* in a way gSOAP uses to process SOAP requests

```
void HTTP_Your_Service::soap_methods(void) {
  if((sp.error = soap_serve_YourNamespace__YourMethod1(&sp)) != SOAP_NO_METHOD) return;
  if((sp.error = soap_serve_YourNamespace__YourMethod2(&sp)) != SOAP_NO_METHOD) return;
}
```

HTTP_Error HTTP_ServiceAdv::parse_header(int &keep_alive)

This method parses content of HTTP header and places results into following fields:

range_start[],range_end[],nranges,range_pinstalaltionassed - data ranges requested by client (Range),

entity_range_start,entity_range_end,entity_range_passed - ranges data presented in body (Content-Range),

entity_size,entity_size_passed - size of data presented in body (Content-Range),

length,length_passed - size of body (Content-Length),

failure_parsing - method failed to parse header,

unsupported_option_passed - there was an option which requires to be processed but method does not support it,

HTTP_Error HTTP_ServiceAdv::send_header(int &keep_alive,int code = 200)

Sends response header which requires no body.

HTTP_Error HTTP_ServiceAdv::send_header(int &keep_alive,uint64_t start,uint64_t end,bool partial,uint64_t full_size)

Sends response header suitable for passing part of data set in body.

2.4 HTTP_Client

```
class HTTP_Client {
 public:
  typedef int (*get_callback_t)(unsigned long long offset,unsigned long long size,char* buf,void* arg
  typedef int (*put_callback_t)(unsigned long long offset,unsigned long long *size,char* buf);
  HTTP_Client(const char* base);
  ~HTTP_Client(void);
  operator bool(void);
  int connect(void);
  int disconnect(void);
  int PUT(const char* path,unsigned long long int offset,unsigned long long int size,const unsigned c
  int GET(const char* path,unsigned long long int offset,unsigned long long int size,get_callback_t c
  bool keep_alive(void);
  unsigned long long int size(void);
};
```

This methods allows to connect to remote site using HTTP, HTTPS or HTTPG protocol. Base URL is specified as constructor's argument *base*.

Actuall connection is done by calling method *connect*. This method can be called even if connection is already established. It returns 0 on success. To close connect use *disconnect*.

Method *GET* implements HTTP GET method. It takes *path* relative to base URL, sends GET request to server also providing the range of required data starting at *offset* of *size* length. Each time chink of data arrives it calls *callback* with *offset* and *size* of data in *buf*. callback can be called multiple times depending on requested and available size.

Method *PUT* implements HTTP PUT method. It sends in body the content of *buf* of length *size* and presents it to server as part of bigger dataset of size *fd_size* starting at *offset*.

2.5 HTTP_ClientSOAP

```
class HTTP_ClientSOAP: public HTTP_Client {
 public:
  HTTP_ClientSOAP(const char* base,struct soap *sp);
  ~HTTP_ClientSOAP(void);
};
```

This class takes care of initializing and configuring gSOAP structure *sp* so it can communicate tp server through HTTP_Client. Upon creation argument *base* is passed to HTTP_Client's constructor. Then *sp* can be used with gSOAP calls to implement SOAP client.

3 Server

3.1 Overview

Server is accessible from outside through 4 TCP/IP ports. Data is authenticated/wrapped/unwrapped using SSLv3 and GSI (Fig.1).

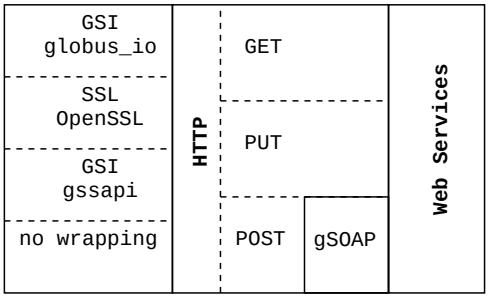

Figure 1: Server layout.

3.2 Configuration

Like most ARC daemons *server* accepts two kinds of configuration files described in [3]. Default location for old one is $ARC_LOCATION/etc/httpds.conf. Name of section for new configuration format is [*httpsd*].

It accepts all generic commands described in above mentioned manual.

Additionally it accepts commands:

gsiport TCP/IP port for GSI connections (Globus implementation),

sslport TCP/IP port for SSL connections,

gssapiport TCP/IP port for GSI connections (CERN implementation),

palinport TCP/IP port for non-wrapped connections,

plugin defines path to a shared library which contains implementation of one or more services.

Authorization is based on specified groups. Actual configuration of allowed operations for every method is configured using service-specific commands and *HTTP_Connector::authorizations* method.

Definition of service in old configuration format is done by block starting from *service* command and ending with *end*. In between there are service-specific commands like

```
service name URL
  command1
  command2
end
```

The *name* is one under which service is registered inside the program.

The *URL* can be either absolute or contain only a path or a port and a path. For example:
httpg://grid.uio.no:8000/logger
:8001/logger
/logger

In a new format each service is represented by subsection of main [*httpsd*] section with *name* defined by command *name* or by name of subsection. The *path/URL* is defined by command *path* and is mandatory.

```
[httpsd/name]
  name=name
  path=URL
  command1=args1
  command2=args2
```

Since 0.5.x all services which run in *httpsd* are compiled as shared libraries. Path to every library is specified using command *plugin* in main section of httpsd configuration. Usually httpsd can find those libraries by names of services without help of *plugin* command. But this may fail if name of the library and name of the service do not match. Or installation was done in non-usual way. So it is always better to supply path to libraries.

3.3 Building

Server with all services is part of NorduGrid toolkit and is built together with all other components of toolkit. For detailed instructions please read documentation available at http://www.nordugrid.org/papers.html .

3.4 Starting

After building (optionally) and installing ARC there should be SysV startup scripts installed in proper place. So You can start httpsd with command '*service httpsd start*' or something like '*/etc/rc.d/init.d/httpsd start*'.

Do not forget to edit configuration file /etc/arc.conf before starting service (if You do not have it yet, look for template at *share* directory of Your installation).

You can also run server directly. For supported options read [3].

References

[1] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *International Journal of Supercomputer Applications*, vol. 11, no. 2, pp. 115–128, 1997.

[2] R.A. van Engelen and others, "gSOAP." [Online]. Available: http://www.cs.fsu.edu/~engelen/soap.html

[3] A. Konstantinov, *Configuration and Authorisation of ARC (NorduGrid) Services*, The NorduGrid Collaboration, NORDUGRID-TECH-6.