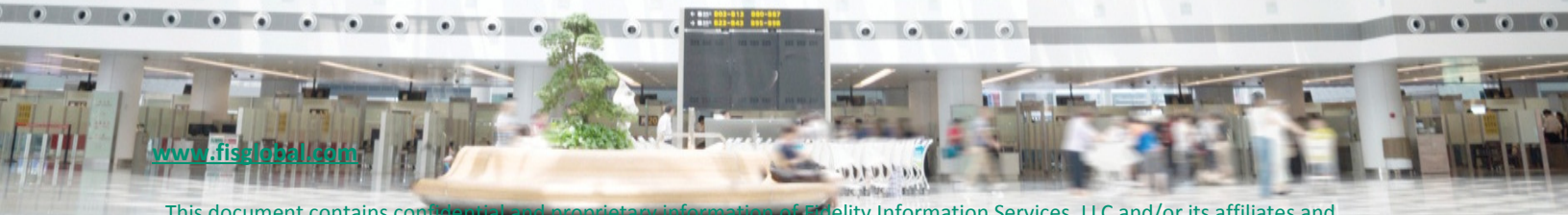
FIS API Manual

Version 1.2
March, 2017

Empowering
the Financial World



www.fisglobal.com

1. Revision History

Version #	Date	Author/Editor	Version/Revision Comments
1.0	October 2012	Sabria BENHAMIDA	Creation: this Manuel replace the document “API Presentation”
1.1	October 2012	François MILOT	Sample of coding
1.2	March 2017	Sabria BENHAMIDA	Migration to FIS format.

Contents

2. Preface

This Manual primarily aimed at IT managers and developers who will be working with FIS applications and/or accessing FIS worldwide network with their own applications but have very little or no experience of the FIS API protocol in general.

This document is made up of eight sections:

- FIS architecture.
- API definition.
- Network protocol and configuration
- Encoding rules
- Message format
- Overview of API connection
- Workflow Mechanisms
- Annex : Sample of c++ coding

3. FIS Architecture

FIS provides electronic-trading and order management solutions, as well as real-time market data feeds over its proprietary worldwide network, FIS Global Network (SGN).

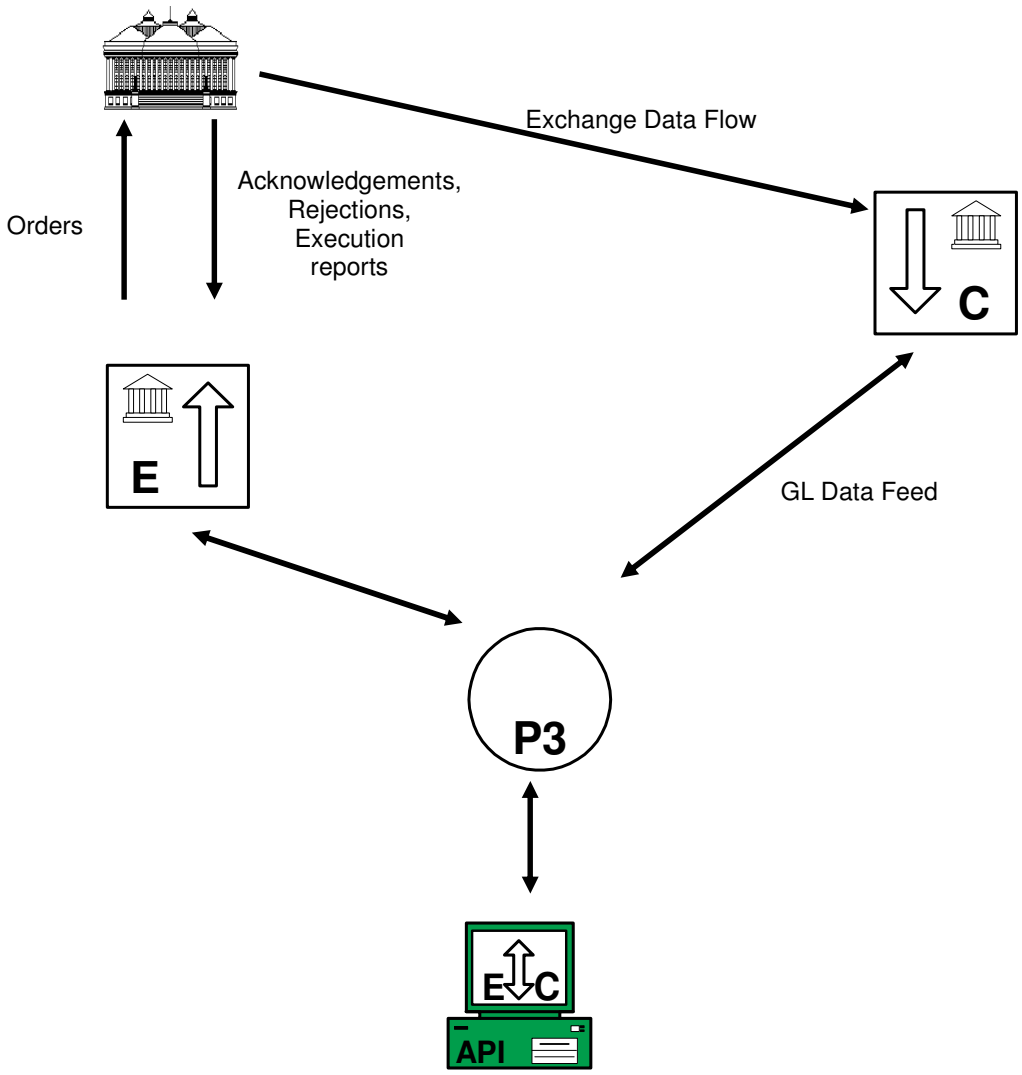
FIS (and API) solutions operate over this network according to architecture with a number of levels. The architecture is independent of the client's operating system and programming language. It is composed of the specialized software components described below:

- **FIS Global network (SGN)**: SGN is a proprietary worldwide network providing trading access and financial data feeds to over 90 stock exchanges. SGN has built-in support for the implementation of remote and external solutions.
- **Router (P3)**: The router is part of the communication gateway between the client applications and the FIS servers and network.
- **Market Data server (SLC)**: Provides real-time financial data feeds from one or more financial markets.
- **Trading server (SLE)**: Sends and monitors buy/sell orders to the market it is connected to.
- **FIS API**: FIS provides a complete range of APIs, allowing third-party applications to access the FIS network and servers.

4. Examples of FIS architecture

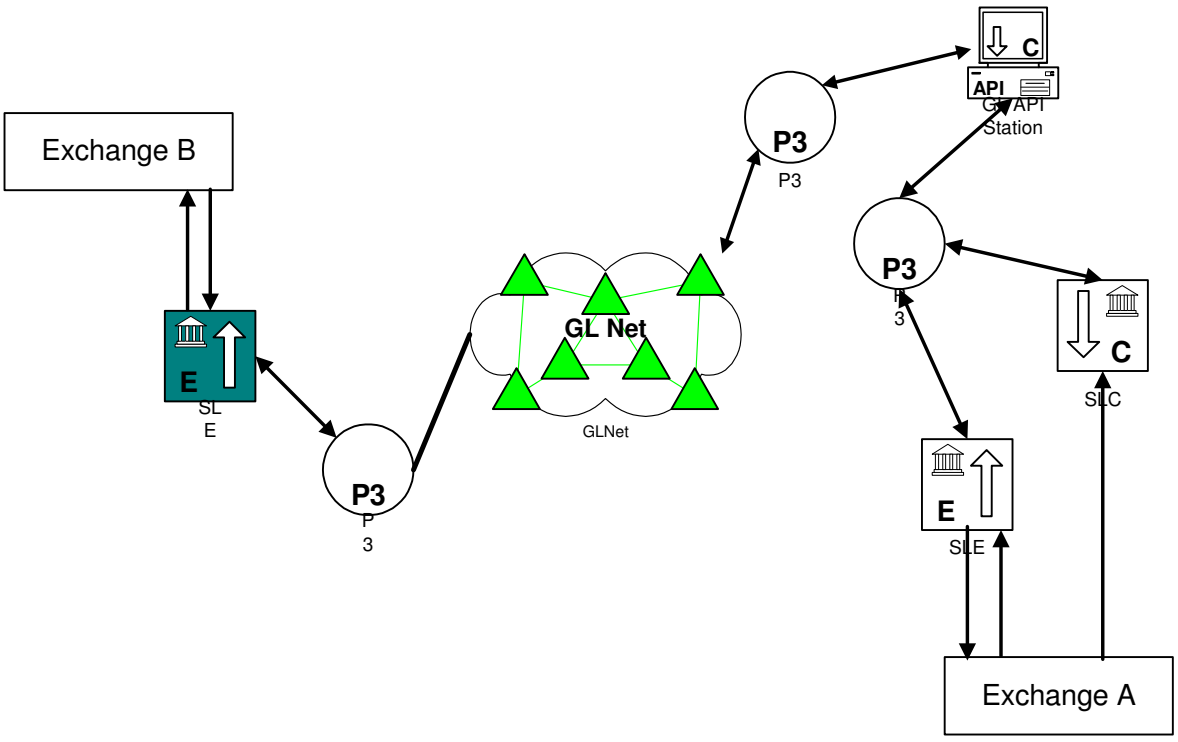
4.1 Local architecture without SGN access

The figure below illustrates a client receiving data flow from Stock Exchange. The SLC and SLE are in local.

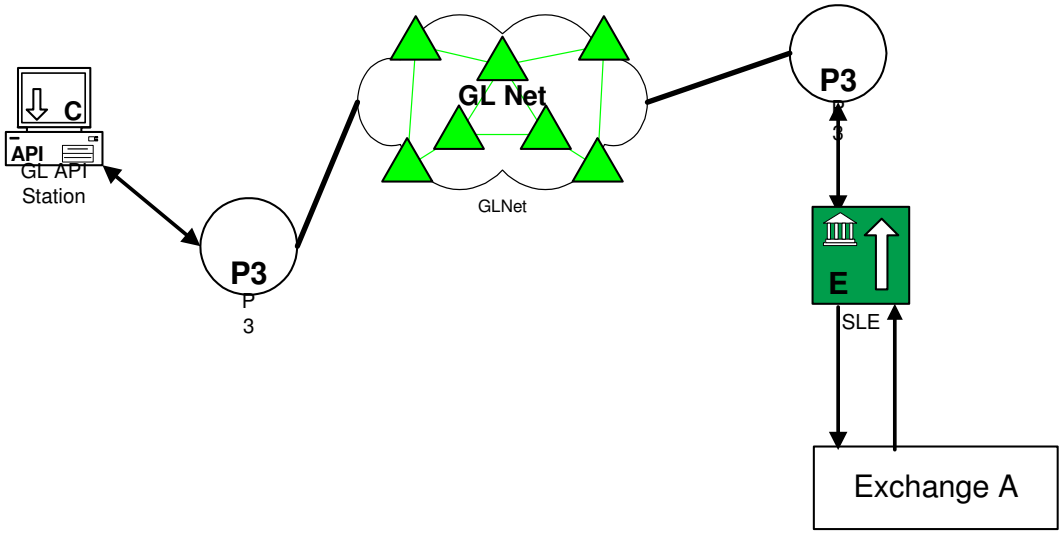


5. Architecture deported via SGN

The figure below illustrates the SGN network while in use by a local client receiving data flow from Stock Exchange B (remote SLC) and Stock Exchange A (local SLC):



A remote client can use the SGN network to issue orders and receive responses via a remote SLE. The figure below illustrate a remote architecture, with a Valdi Trader or FIS API client accessing a remote SLE via SGN in order to enter orders on Exchange A.



6. Definition of FIS API

FIS APIs are literally an « Application Programming Interface ». They were designed by FIS for customers who have decided to develop their own front-end application for sending orders and/or receiving market feed.

For this reason, FIS has built its own published message-based protocol specifying the communication between FIS server and a client application. Moreover, FIS APIs are ***independent of the technical platform and of the programming language*** used by the clients.

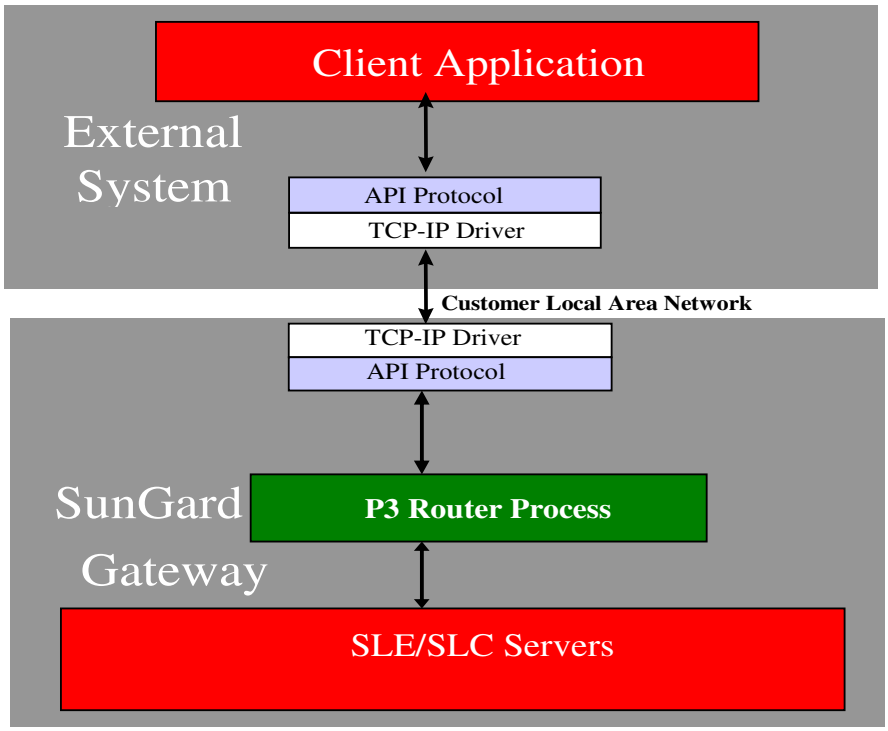
To establish a dialog between FIS server and a Client application, both Servers and Clients have to respect a standardized and precise procedure. This procedure and the management rules associated with it are described below.

7. The Network

7.1 Network protocol

Customers will have to build their applications directly and using ***TCP/IP*** only to take advantage of FIS API. A dedicated process called P3 (or P3 router) is the interface between the Client application and FIS Servers. A unique TCP connection has to be set up by the Client with the P3 process in order to send and route API messages to one or more FIS servers.

The following figure shows how an external Client application using FIS API protocol is connected to FIS Server through P3:



7.2 Connection to FIS Servers

The P3 router process is responsible for establishing and managing communication between the clients (FIS API) and FIS server. To do this, as soon as it is started up, it listens continuously for connections from client processes or FIS server processes.

The server processes connect to the P3 router process. They log on as service providers and "listen" for requests originating from the clients. In the same way, the client processes connect to the P3 router process and request access to a specific service provider. The P3 router process then establishes a logical link between the client process and the server process concerned, on condition that the server process is available. The two processes, once connected, can then communicate and exchange information; the client application can exchange data with FIS server.

To establish a session with FIS Server, the Client application must:

- First, establish an ***IP connection*** with the IP address of FIS Gateway where the P3 process is located (i.e. ISO level 3 connection)
- Second, establish a ***TCP connection*** with the P3 process. P3 is always listening and waiting for connections from any Clients, either VALDI Trader stations or API Programs (i.e. ISO Level 4 connection).

- **Server Connection**:

  When a GL server process connects to the communication handler (also called P3), it identifies itself as a service provider and then listens for requests from clients.

- **Client Connection:**

  When a client process connects to the communication handler, it requests access to a specific server. The communication handler then establishes a logical link between the client and the server (if the server is available) and connects the two processes. Once the two processes are connected, the client can exchange data with the server and vice versa.

8. Encoding type

8.1 FIS coding rule

- ✓ FIS Format

In FIS format, the first byte indicates the length of the field.

The length is calculated as following:

Contains the value + 32 in ASCII

E.g length = 7

First byte = 7 + 32 = 39 = « ‘ »

The rest is the contents of the field coded in ASCII.

E.g value = FTE

First byte = 3 + 32 = 35 = « # »

Rest = « FTE »

So the field = « #FTE »

Warning: This format implies that all the fields have a ***variable length***.

- ✓ ACSII : Data field in ASCII format.
  E.g. 02007 coded « 02007 »

- ✓ Filler : Field only contains spaces: ASCII char(32).

- ✓ GL_C : Fixed length coded avers one byte.
  Contains the value + 32
  E.g. value = 7
  Field = 7+32=39= « ’ »

9. Message Format

9.1 Message Structure

FIS message is a string of variable length in ASCII 8-bit format that must adhere to the following format:

LG	HEADER	DATA	FOOTER

Where :

- LG : Total message length (including LG, Header, Data, Footer)
- Header : General message information
- Data : Message data
- Footer : End of message

9.2 LG

The LG section gives the total length of the message.

It is coded over 2 bytes : LG[0] and LG[1], where :

- LG[0] : LG%256 (remainder of LG/256)
- LG[1] : LG/256 (integer part of LG/256)
- LG = LG[0] + 256 * LG[1]

9.3 Header

The header is of a standard fixed length of 32 bytes.

Remark : The format of the header is the same for SLC V4, SLC V5, SLE V4, SLE V5, P3 and OMS V5

Only the field API version changes.

For server in V4 mode, this field must be filled with « », else it is filled with « 0 ».

Type	Width	*Field*
BINARY	1	STX = 2
ASCII	1	API version
ASCII	5	Request size
ASCII	5	Called logical identifier
FILLER	5	Filler
ASCII	5	Calling Logical identifier
FILLER	2	Filler
ASCII	5	Request Number
FILLER	3	Filler

9.4 Data

The data's are depending of the server and the request. To know the several data's of a request, you need to get the API Exchange documentation available on the Web site.

9.5 Footer

The footer is of a fixed length of 3 bytes and indicates the end of the message. It only contains fillers and binary fields.

Remark: The format of the footer is the same for SLC V4, SLC V5, SLE V4, SLE V5, P3 and OMS V5.

Type	Width	*Field*
FILLER	2	Filler
BINARY	1	ETX = 3

10. Mechanism of an API connection

To communicate with FIS server, you need to follow the following:

1. **IP Connection:**
   The API client and the GL TRADE server communicate by exchanging IP frames, either on the same LAN, or on different LANs connected by (P3) routers. You can use a PING to check this connection.

2. **TCP Connection and Specific synchronisation with P3 :**
   Once started, the router process opens a TCP listening port and waits for the client and server connection requests. Once the TCP session is active, the router is ready for the exchange of client and server messages.

3. **Client confirms its connection to the server:**
   The client API sends an identification message. This consists of a 16 byte message. The P3 does not send a response to this message, but the server is now ready to receive API requests from the client (1100 request).

4. **Logical Connection:**

   The client API must then send a logical connection request (1100 API request). The response from the server can be:

   - Connection established: server returns an 1100 request.
   - Connection refused: either the server returns an error message, or the P3 breaks the TCP session with no error message.
   - No response: if there is no response within a minute, this may indicate network problems.

5 **Exchange API messages:**

   Once the physical and logical connections are established, the client API can send and receive messages.

10.1 IP connection

Both the Client and Server system must communicate by exchanging IP frames, either on the same Local Area Network in the customer site or on different LANs separated by routers.

Remark: The PING standardized program can be used to validate this first step.

10.2 TCP connection

When the P3 process is started and running, it opens a TCP port (see the router configuration file P3V5.INI, section [COMMON], parameter listen_tcp) and waits for a Client connection. The Client application must then open a TCP socket to this port.

Once the TCP session is up, it will stay active permanently to allow Server and Client to exchange API messages.

10.3 Synchronisation with P3

The Client must send a specific identification sequence. The entire string must have a length of 16 bytes and may contain anything.

At this stage, ***no response is returned*** by P3 but the ***P3 process is ready*** to receive API requests from the Client.

10.4 Connection with the server

The Client must therefore send a ***Logical Connection***, or 1100 API Request. The response form the server can be:

- Connection established: the Server then returns 1100 Request.
- Connection refused: even if the Client request respects the FIS API format, some Error Return Codes may be sent back in a 1102 Request. If the Client Request format is wrong (invalid length for instance), the TCP session is simply aborted by P3 and no Return Code is sent back to the Client.
- No response: if there is no response within one minute this may indicate a network problem.

10.5 API messages

The ***physical and logical connections*** are now active. You can start to send API messages for servers connected

11. Workflow Mechanisms

This section presents some examples of the message workflows between API clients and GL TRADE servers.

11.1 Logical Connection

The following 2 diagrams illustrate a successful and an unsuccessful connection.

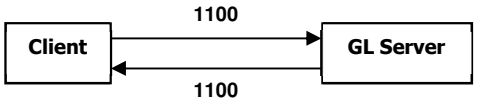


Figure 3 *Successful Logical Connection*

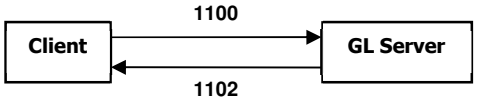


Figure 4 *Unsuccessful Logical Connection*

11.2 Logical Disconnection

The following 2 diagrams illustrate a voluntary and an involuntary disconnection.

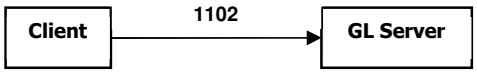


Figure 5 *Voluntary Disconnection*

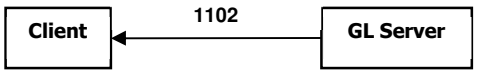


Figure 6 *Involuntary Disconnection*

11.3 Order Book Consultation

The SLE stores all the orders and related market messages in its database. API clients can request information about orders from this database. Use request 2004 to consult a **simple order** in the order book.

> **Note:** A simple order has only one leg.

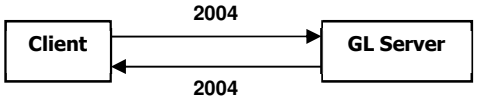


Figure 7 Simple order consultation

11.4 Replies Consultation

The following 3 diagrams illustrate the standard reply mechanisms.

1. The client can consult the replies database using request 2008. The client receives the replies history (2008) for himself and all the users he supervises.

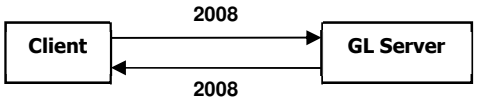


Figure 8 Reply to consultation

2. Once an order is sent, the market automatically sends an update message on this order. In order to receive the update, the API client must subscribe to real-time mode (message 2017), otherwise the client receives message 2008 in snapshot mode.

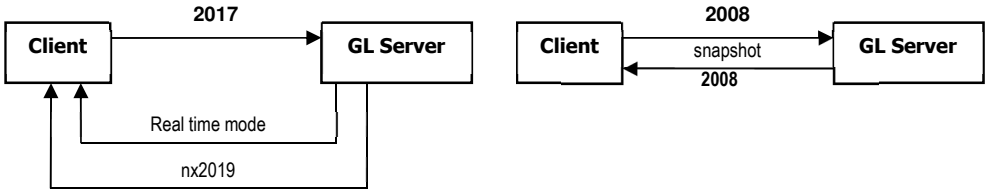
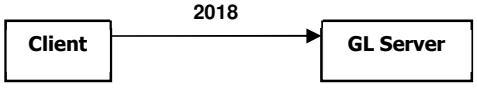


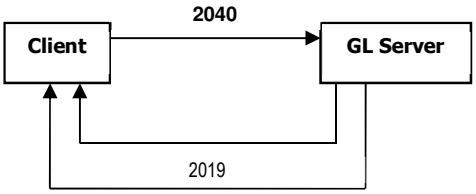
Figure 9 Subscription mechanism

3. Send request 2018 to unsubscribe from real-time mode.



11.5 Sending an Order

Use request 2000 or 2040 to send an order. Before sending the order, you must already have sent a real-time replies request (2017).



11.6 API Tools

This tool allows developers to send and receive all FIS API messages for all servers. This tool with its accompanying documentation is delivered. Documentation updates are also available via gltrade.net.

12. Annex: Sample of C++ coding

This piece of code written in C++ is a basic example of a program showing successful API Login and order sending.

The code snippet can be adapted accordingly and can be used as a tool to resolve any API connection issues.

```
/*****************************************************************************
************************************/

/********************/
/*** INCLUDES ***/
/********************/

#include <stdio.h>
#include <stdarg.h>
#include <unistd.h>
#include <netdb.h>
#include <linux/tcp.h>



/******************/
/*** DEFINES ***/
/******************/

#define FALSE               0
#define TRUE                1
#define LG_LENGTH           2
#define STX                 2
#define TWO_BYTE_FILLER     2
#define ETX                 3
#define FOOTER_LENGTH       3
#define USER_LENGTH         3
#define THREE_BYTE_FILLER   3
#define REQ_SIZE_LENGTH     5
#define REQ_NUM_LENGTH      5
#define CALLING_ID_LENGTH   5
#define FIVE_BYTE_FILLER    5
#define DEST_LENGTH         5
#define PARAM_NUMBER_MIN    6
#define SEVEN_BYTE_FILLER   7
#define MAXFD               8
#define TEN_BYTE_FILLER     10
#define ELEVEN_BYTE_FILLER  11
#define PASS_LENGTH         16
#define CLIENT_NAME_LG      16
#define HEADER_LENGTH       32
#define MAX_BUFFER_SIZE     256
#define MAX_TRACE           1024
#define MAX_BUFFER_READ     32000
#define MAX_MSG_LENGTH      32768
```

```
/*****************************************/
/*** ENUMERATION DECLARATION ***/
/*****************************************/

typedef enum {
NOT_READY               = 0,
PHYSICALLY_CONNECTED    = 1,
LOGICALLY_CONNECTED     = 2,
READY                   = 3
} SERVER_STATUS;



/*********************************/
/*** GLOBAL VARIABLES ***/
/*********************************/

unsigned char *     hostAdress;         /* server adress            */
unsigned short      hostPort;           /* server port              */
FILE *              logFile;            /* main log file            */
FILE *              dumpFile;           /* dump file                */
struct hostent *    hostinfo;           /* host structure info      */
struct sockaddr_in  sas;                /* socket adress structure  */
int                 socketFD;           /* server socket descriptor */
unsigned char *     destinationServer;  /* destination server       */
unsigned int        destLength;         /* destination length       */
unsigned char *     userNumber;         /* user number              */
unsigned int        userLength;         /* user number length       */
unsigned char *     password;           /* password                 */
unsigned int        passLength;         /* password length          */
unsigned char *     callingID;          /* calling logical id       */
unsigned char *     clientName;         /* client application name  */
SERVER_STATUS       serverStatus;       /* server status            */
static fd_set       reaMsk;             /* read mask for select     */
static fd_set       wriMsk;             /* write mask for select    */



/*************************************/
/*** FUNCTIONS DECLARATION ***/
/*************************************/

static int  initParam           (int, char **);
static int  checkParam          (int, char **);
static void printParam          ();
static int  encode1100          (unsigned char *);
static int  encodeHeader        (unsigned char *, unsigned int, unsigned int);
static int  encodeMessageLength ();
static int  init                ();
static int  sendReq1100         ();
```

```
static int  initCommunication   ();
static void exitProcess         ();
static int  decodeHeader        (unsigned char *, unsigned int, int *);
static int  readSocket          ();
static int  sendOrder           ();
static int  sendSubscription    ();
static int  logMessage          (const char *, ...);



/**********************/
/*** MAIN PROC ***/
/**********************/

int main (int argc, char **argv) {
    struct timeval timMsk;
    int selectReturn, run = TRUE;

    /* initialization */
    if (init () == FALSE) {
        exitProcess ();
    }

    /* timeout for select */
    timMsk.tv_sec  = 1;
    timMsk.tv_usec = 0;

    /* parameter initialization */
    if (!initParam (argc, argv)) {
        exitProcess ();
    }

    /* communication initialization */
    if (!initCommunication ()) {
        logMessage ("Unable to establish communication channel\n");
        exitProcess ();
    }

    /* send logical connection */
    if (sendReq1100 () == FALSE) {
        logMessage ("Unable to send logical connection request\n");
        exitProcess ();
    }

    /* event loop */
    while (run == TRUE) {

        /* select call */
        selectReturn = select (MAXFD, &reaMsk, &wriMsk, (fd_set *) 0, &timMsk);

        /* something in select */
        if (selectReturn > 0) {
            readSocket ();
```

```
        }
    }

    /* end of process */
    return 0;
}




/**********************/
/*** FUNCTIONS ***/
/**********************/

/* print required parameters */
static void printParam () {
    logMessage ("\n\nUsage\t:\tclient [host address] [port] [serverDestination]
[userNumber] [userPassword]\n\n");
}

/* check input parameters */
static int checkParam (int argc, char **argv) {

    /* minimum number of argument not reached */
    if (argc < PARAM_NUMBER_MIN) {
        return FALSE;
    }

    /* everything's ok */
    return TRUE;
}

/* parameter initialization */
static int initParam (int argc, char **argv) {
    unsigned int hostLength;

    /* check parameters */
    if (!checkParam (argc, argv)) {
        printParam ();
        return FALSE;
    }

    /* hostname memory allocation */
    hostLength = strlen (argv[1]);

    if ((hostAdress = (unsigned char *) malloc ((hostLength + 1) * sizeof
(unsigned char))) == (unsigned char *) 0) {
        logMessage ("Not enough memory\n");
        return FALSE;
    }

    memset (hostAdress, 0, hostLength + 1);
```

```
    memcpy (hostAdress, argv[1], hostLength);

    /* convert second argument */
    hostPort = atoi (argv[2]);

    /* get back hostname information */
    hostinfo = gethostbyname (hostAdress);
    if (hostinfo == NULL) {
        if ((hostinfo = gethostbyaddr (hostAdress, hostLength, AF_INET)) == NULL)
{
            logMessage ("Unknown host %s\n", hostAdress);
            return FALSE;
        }
    }

    memmove ((char *)&sas.sin_addr, hostinfo->h_addr, hostinfo->h_length);
    sas.sin_port = htons (hostPort);
    sas.sin_family = hostinfo->h_addrtype;

    /* get destination server value */
    destLength = strlen (argv[3]);

    if ((destinationServer = (unsigned char *) malloc ((destLength + 1) * sizeof
(unsigned char))) == (unsigned char *) 0) {
        logMessage ("Not enough memory\n");
        return FALSE;
    }
    memset (destinationServer, 0, destLength + 1);
    memcpy (destinationServer, argv[3], destLength);

    /* get user number value */
    userLength = strlen (argv[4]);

    if ((userNumber = (unsigned char *) malloc ((userLength + 1) * sizeof
(unsigned char))) == (unsigned char *) 0) {
        logMessage ("Not enough memory\n");
        return FALSE;
    }
    memset (userNumber, 0, userLength + 1);
    memcpy (userNumber, argv[4], userLength);

    /* get password value */
    passLength = strlen (argv[5]);

    /* password length check */
    if (passLength > PASS_LENGTH) {
        logMessage ("Password length too long\n");
        return FALSE;
    }

    if ((password = (unsigned char *) malloc ((passLength + 1) * sizeof (unsigned
char))) == (unsigned char *) 0) {
        logMessage ("Not enough memory\n");
```

```
        return FALSE;
    }
    memset (password, 0, passLength + 1);
    memcpy (password, argv[5], passLength);

    /* no error */
    return TRUE;
}

/* first two bytes length encoding function */
static int encodeMessageLength (unsigned char *buf, unsigned int dataLength) {
    int totalLength = LG_LENGTH + HEADER_LENGTH + FOOTER_LENGTH + dataLength;

    /* encode total length */
    if (buf != NULL) {
        buf[0] = totalLength % 256;
        buf[1] = totalLength / 256;

        return TRUE;
    }

    /* no input buffer */
    return FALSE;
}

/* encode header by computing data length */
static int encodeHeader (unsigned char *buf, unsigned int requestSize, unsigned
int request) {
    unsigned int off = LG_LENGTH, idx = 0;
    unsigned char requestSizeString[REQ_SIZE_LENGTH];
    unsigned char requestNumberString[REQ_NUM_LENGTH];

    /* buffer integrity check */
    if (buf == (unsigned char *) 0) {
        logMessage ("Invalid input buffer\n");
        return FALSE;
    }

    /* STX encoding */
    buf[off++] = STX;

    /* API version */
    buf[off++] = ' ';

    /* request size encoding */
    sprintf (requestSizeString, "%05d", requestSize);
    memcpy (&buf[off], requestSizeString, REQ_SIZE_LENGTH);
    off += REQ_SIZE_LENGTH;

    /* called logical identifier encoding */
    while (idx < (DEST_LENGTH - destLength)) {
        buf[off++] = '0';
        idx++;
```

```
    }
    memcpy (&buf[off], destinationServer, destLength);
    off += destLength;

    /* five bytes filler */
    memcpy (&buf[off], "     ", FIVE_BYTE_FILLER);
    off += FIVE_BYTE_FILLER;

    /* calling ID encoding */
    memcpy (&buf[off], callingID, CALLING_ID_LENGTH);
    off += CALLING_ID_LENGTH;

    /* two bytes filler */
    memcpy (&buf[off], "  ", TWO_BYTE_FILLER);
    off += TWO_BYTE_FILLER;

    /* request number encoding */
    sprintf (requestNumberString, "%05d", request);
    memcpy (&buf[off], requestNumberString, REQ_NUM_LENGTH);
    off += REQ_NUM_LENGTH;

    /* three bytes filler */
    memcpy (&buf[off], "   ", THREE_BYTE_FILLER);
    off += THREE_BYTE_FILLER;

    /* no error */
    return TRUE;
}

/* encode header by computing data length */
static int decodeHeader (unsigned char *msg, unsigned int msgLength, int
*requestNumber) {
    unsigned int oct, length, reqSize;
    unsigned char requestNumberString [REQ_NUM_LENGTH];
    unsigned char requestLengthString [REQ_SIZE_LENGTH + 1];

    /* initialization */
    oct = 0;

    /* go to STX */
    while ((msg[oct] != STX) && (oct < MAX_BUFFER_READ)) oct++;
    if (oct == MAX_BUFFER_READ) return FALSE;

    /* get back length */
    length = msg[oct-2] + msg[oct-1] * 256;

    /* skip API version */
    oct += 2;

    /* request size */
    memcpy (requestLengthString, &msg[oct], REQ_SIZE_LENGTH);
    requestLengthString[REQ_SIZE_LENGTH + 1] = 0;
    reqSize = atoi (requestLengthString);
```

```
    oct += REQ_SIZE_LENGTH;

    /* decode called identifier */
    memcpy (callingID, &msg[oct], CALLING_ID_LENGTH);
    oct += CALLING_ID_LENGTH;

    /* five bytes filler */
    oct += FIVE_BYTE_FILLER;

    /* decode calling identifier */
    oct += CALLING_ID_LENGTH;

    /* two bytes filler */
    oct += 2;

    /* decode request number */
    memcpy (requestNumberString, &msg[oct], REQ_NUM_LENGTH);
    *requestNumber = atoi (requestNumberString);

    /* no error */
    return TRUE;
}

/* encoding logical connection request function */
static int encode1100 (unsigned char *buf) {
    unsigned int idx = 0, lg = 0;
    unsigned int off = LG_LENGTH + HEADER_LENGTH;

    /* checking buffer integrity */
    if (buf != (unsigned char *) 0) {

        /* user number encoding */
        while (idx < (USER_LENGTH - userLength)) {
            buf[off++] = '0';
            idx++;
        }
        memcpy (&buf[off], userNumber, userLength);
        off += userLength;
        lg += USER_LENGTH;

        /* password encoding */
        idx = 0;
        memcpy (&buf[off], password, passLength);
        off += passLength;
        while (idx < (PASS_LENGTH - passLength)) {
            buf[off++] = ' ';
            idx++;
        }
        lg += PASS_LENGTH;
    }

    /* seven bytes filler encoding */
    memcpy (&buf[off], "       ", SEVEN_BYTE_FILLER);
```

```
    off += SEVEN_BYTE_FILLER;
    lg += SEVEN_BYTE_FILLER;

    /* footer encoding */
    memcpy (&buf[off], "  ", TWO_BYTE_FILLER);
    off += TWO_BYTE_FILLER;
    buf[off++] = ETX;

    /* header encoding */
    if (encodeHeader (buf, lg + HEADER_LENGTH + FOOTER_LENGTH, 1100) == FALSE) {
        logMessage ("An error occured in 1100 header encoding\n");
        return FALSE;
    }

    /* total length encoding */
    if (encodeMessageLength (buf, lg) == FALSE) {
        logMessage ("An error occured in 1100 total length encoding\n");
        return FALSE;
    }

    /* no error */
    return (LG_LENGTH + HEADER_LENGTH + FOOTER_LENGTH + lg);
}

/* global initialization function */
static int init () {

    /* default log file is standard output */
    logFile = stdout;
    serverStatus = NOT_READY;

    /* memory allocation */
    if ((callingID = (unsigned char *) malloc (CALLING_ID_LENGTH * sizeof
(unsigned char))) == (unsigned char *) 0) {
        logMessage ("Not enough memory\n");
        return FALSE;
    }
    memset(callingID, ' ', CALLING_ID_LENGTH);

    /* dump file opening */
    if ((dumpFile = fopen ("dump.err", "wb+")) == (FILE *) 0) {
        logMessage ("Error in dumpFile opening\n");
        return FALSE;
    }

    /* memory allocation */
    clientName = (unsigned char *) malloc (CLIENT_NAME_LG * sizeof (unsigned
char));

    if (clientName == (unsigned char *) 0) {
        logMessage ("Not enough memory\n");
        return FALSE;
    }
```

```
    memcpy (clientName, "APICLIENT       ", CLIENT_NAME_LG);

    /* no error */
    return TRUE;
}

/* logical connection sending function */
static int sendReq1100 () {
    unsigned int msgLength, lgSent;
    unsigned char msg[MAX_MSG_LENGTH];

    /* request encoding */
    if ((msgLength = encode1100 (msg)) == FALSE) {
        logMessage ("Error in 1100 encoding\n");
        return FALSE;
    }

    /* socket writing */
    logMessage ("Sending request 1100 length %d\n", msgLength);
    if ((lgSent = write (socketFD, msg, msgLength)) != msgLength) {
        logMessage ("Integrity error, part of message lost encoded length = %d,
sent length = %d\n", msgLength, lgSent);
        return FALSE;
    }

    /* no error */
    return TRUE;
}

/* communication initialization */
static int initCommunication () {
    int bufsck, lgSent;
    struct sockaddr_in addr;

    /* build socket */
    socketFD = socket (AF_INET, SOCK_STREAM, 0);
    setsockopt (socketFD, IPPROTO_TCP, TCP_NODELAY, (char *) &bufsck, sizeof
(int));

    /* bind socket */
    memset ((char *)&addr, 0, sizeof (struct sockaddr_in));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl (INADDR_ANY);

    /* bind listening socket */
    if (bind (socketFD, (struct sockaddr *) &addr, sizeof (struct sockaddr_in)) ==
-1) {
        logMessage ("Unable to to bind socket\n");
        return FALSE;
    }

    /* try to connect to socket */
```

```
    if (connect (socketFD, (struct sockaddr *) &sas, sizeof (struct sockaddr_in))
== -1) {
        logMessage ("Unable to connect to address %s\n", inet_ntoa
(sas.sin_addr));
        return FALSE;
    }

    serverStatus = PHYSICALLY_CONNECTED;
    logMessage ("Successfully connected to %s\n", inet_ntoa (sas.sin_addr));

    /* client identification send */
    if ((lgSent = write (socketFD, clientName, CLIENT_NAME_LG)) != CLIENT_NAME_LG)
{
        logMessage ("Integrity error, part of message lost encoded length = %d,
sent length = %d\n", CLIENT_NAME_LG, lgSent);
        return FALSE;
    }

    /* mask initialization */
    FD_ZERO (&reaMsk);
    FD_ZERO (&wriMsk);
    FD_SET  (socketFD, &reaMsk);
    FD_SET  (socketFD, &wriMsk);

    /* no error */
    return TRUE;
}

/* exit function */
static void exitProcess () {
    if (dumpFile != (FILE *) 0) {
        fclose (dumpFile);
    }

    /* exit process */
    exit (1);
}

/* read on socket */
static int readSocket () {
    int size, reqNum;
    unsigned char *buf;

    /* memory allocation */
    if ((buf = (unsigned char *) malloc (MAX_BUFFER_READ * sizeof (unsigned
char))) == (unsigned char *) 0) {
        logMessage ("Memory allocation error\n");
        return FALSE;
    }

    /* get inbound message */
    if ((size = recv (socketFD, (char *) buf, MAX_BUFFER_READ, 0)) > 0) {
        decodeHeader ((unsigned char *) buf, size, &reqNum);
```

```
        switch (reqNum) {
            case 1100   : {
                /* no check on 1100 (TOIMPLEMENT, check on user number, called ID)
*/
                if (serverStatus == PHYSICALLY_CONNECTED) {
                        /* real time subscription */
                        if (sendSubscription () == FALSE) {
                            logMessage ("Unable to send subscription request\n");
                            exitProcess ();
                        }
                    /* no difference between logical connection and server
readyness */
                    serverStatus = READY;
                    logMessage ("Logical connection to server established\n");
                }
                sendOrder ();
                break;
            }
            default : break;
        }
    }

    /* no error */
    return TRUE;
}

/* send an order */
static int sendOrder () {
    unsigned char buf[MAX_MSG_LENGTH];
    unsigned int idx = 0, lg = 0, off = LG_LENGTH + HEADER_LENGTH, messageLength,
lgSent;

    /* checking buffer integrity */
    if (buf != (unsigned char *) 0) {

        /* user number encoding */
        while (idx < (CALLING_ID_LENGTH - userLength)) {
            buf[off++] = '0';
            idx++;
        }
        memcpy (&buf[off], userNumber, userLength);
        off += userLength;
        lg += CALLING_ID_LENGTH;

        /* request category (single order) */
        buf[off++] = 'O';
        lg++;

        /* command (0 for creation) */
        buf[off++] = '0';
        lg++;
```

```
        /* stockcode (FTE in our example) */
        buf[off++] = 35;
        lg++;
        memcpy (&buf[off], "FTE", 3);
        off += 3;
        lg += 3;

        /* 10 byte filler */
        memcpy (&buf[off], "          ", TEN_BYTE_FILLER);
        off += TEN_BYTE_FILLER;
        lg += TEN_BYTE_FILLER;

        /* bitmap encoding by setting GLID only (V3 pos 106, Paris GLID) */
        for (idx = 0; idx < 106; idx++) {
            buf[off++] = ' ';
            lg++;
        }
        buf[off++] = 44; lg++;
        memcpy (&buf[off], "000100000000", 12);
        off += 12;
        lg += 12;
    }

    /* footer encoding */
    memcpy (&buf[off], "  ", TWO_BYTE_FILLER);
    off += TWO_BYTE_FILLER;
    buf[off++] = ETX;

    /* header encoding */
    if (encodeHeader (buf, lg + HEADER_LENGTH + FOOTER_LENGTH, 2000) == FALSE) {
        logMessage ("An error occured in 2000 header encoding\n");
        return FALSE;
    }

    /* send order on socket */
    messageLength = LG_LENGTH + HEADER_LENGTH + FOOTER_LENGTH + lg;

    /* total length encoding */
    if (encodeMessageLength (buf, lg) == FALSE) {
        logMessage ("An error occured in 2000 total length encoding\n");
        return FALSE;
    }

    /* write message on socket */
    if ((lgSent = write (socketFD, buf, messageLength)) != messageLength) {
        logMessage ("Integrity error, part of message lost encoded length = %d,
sent length = %d\n", messageLength, lgSent);
        return FALSE;
    }
    logMessage ("Order sent : encoded length = %d, sent length = %d\n",
messageLength, lgSent);

    /* no error */
```

```
    return (messageLength);
}

/* real-time subscription */
static int sendSubscription () {
    unsigned char buf[MAX_MSG_LENGTH];
    unsigned int idx = 0, lg = 0, off = LG_LENGTH + HEADER_LENGTH, messageLength,
lgSent;

    /* checking buffer integrity */
    if (buf != (unsigned char *) 0) {

        /* ack */
        buf[off++] = '1';
        lg++;

        /* reject */
        buf[off++] = '1';
        lg++;

        /* exchange reject */
        buf[off++] = '1';
        lg++;

        /* trade execution */
        buf[off++] = '0';
        lg++;

        /* exchange message */
        buf[off++] = '0';
        lg++;

        /* default */
        buf[off++] = '0';
        lg++;

        /* inflected message */
        buf[off++] = '0';
        lg++;

        /* 11 bytes filler */
        memcpy (&buf[off], "           ", ELEVEN_BYTE_FILLER);
        off += ELEVEN_BYTE_FILLER;
        lg += ELEVEN_BYTE_FILLER;
    }

    /* footer encoding */
    memcpy (&buf[off], "  ", TWO_BYTE_FILLER);
    off += TWO_BYTE_FILLER;
    buf[off++] = ETX;

    /* header encoding */
    if (encodeHeader (buf, lg + HEADER_LENGTH + FOOTER_LENGTH, 2017) == FALSE) {
```

```
        logMessage ("An error occured in 2017 header encoding\n");
        return FALSE;
    }

    /* total length encoding */
    if (encodeMessageLength (buf, lg) == FALSE) {
        logMessage ("An error occured in 2017 total length encoding\n");
        return FALSE;
    }

    /* send order on socket */
    messageLength = LG_LENGTH + HEADER_LENGTH + FOOTER_LENGTH + lg;

    /* write message on socket */
    if ((lgSent = write (socketFD, buf, messageLength)) != messageLength) {
        logMessage ("Integrity error, part of message lost encoded length = %d,
sent length = %d\n", messageLength, lgSent);
        return FALSE;
    }
    logMessage ("Realtime subscription request sent : encoded length = %d, sent
length = %d\n", messageLength, lgSent);

    /* no error */
    return (messageLength);
}

/* log function */
static int logMessage (const char *message, ...) {
    va_list argPtr;
    char tra[MAX_TRACE];

    /* recuperation des parametres */
    va_start (argPtr, message);
    vsprintf (tra, message, argPtr);
    va_end (argPtr);

    /* test logFile integrity */
    if (logFile != (FILE *) 0)
        fprintf (logFile, tra);

    /* no error */
    return TRUE;
}
```