Marcus
von Cube
Systemsoftware

Janus API

HP NonStop® Server Access

Manual

Date: March 12, 2008

Marcus von Cube
Systemsoftware
Am Hebestumpf 6
61273 Wehrheim
Germany
Phone: +49 6081 59527
Mobile: +49 177 3200614
Fax: +49 6122 56366

marcus@mvcsys.de
www.mvcsys.de

Contents

1 Introduction......1
1.1 History of the the Janus API......1
1.1.1 The beginnings......1
1.1.2 Getting independent......1
1.1.3 Extending the view......1
1.2 Basic functionality......2
1.2.1 Overview......2
1.2.2 What Janus is not meant for......2
1.2.3 Installation......3
1.2.4 Establishing a connection......3
1.2.5 Handling transactions......3
1.2.5.1 Starting and ending a transaction......3
1.2.5.2 Handling automatic transactions......3
1.2.6 Sending messages......3
1.2.6.1 What is a message?......3
1.2.6.2 Destinations......4
1.2.6.3 Timeouts and Nowait I/O......4
1.2.7 Receiving UMS messages......4
1.2.8 Error handling......5
1.2.9 Connection pooling......5
1.2.9.1 Connection pools in Java......5
1.2.9.2 The web based proxy......5
2 API details......6
2.1 The Windows DLL......6
2.1.1 Installation and general remarks......6
2.1.2 Linking the DLL into your application......6
2.1.2.1 Visual Basic......6
2.1.2.2 Delphi Pascal......6
2.1.2.3 C/C++......6
2.1.3 API calls......7
2.1.3.1 Connection Handling......7
2.1.3.2 Transactions......8
2.1.3.3 Sending messages......8
2.1.3.4 Error information......9
2.2 The C API......10
2.2.1 Introduction......10
2.2.2 Distributed files......10
2.2.2.1 Header files......10
2.2.2.2 C source files......10
2.2.2.3 Sample Clients......11
2.2.3 API calls......11
2.2.3.1 Connection handling......11
2.2.3.2 Disconnection......12
2.2.3.3 Errors......12
2.2.3.4 Transactions......12
2.2.3.5 Sending messages......12
2.2.3.6 Server Programming......13
2.2.3.7 Tracing and debugging......14
2.2.3.8 Utility functions......14

1 Introduction

In Roman mythology, Janus (or Ianus) was the god of gates, doors, doorways, beginnings, and endings.

Janus was usually depicted with two faces looking in opposite directions. Janus was frequently used to symbolize change and transitions such as the progression of past to future, of one condition to another, of one vision to another, the growing up of young people, and of one universe to another. He was also known as the figure representing time because he could see into the past with one face and into the future with the other.

http://en.wikipedia.org/wiki/Janus_(mythology))

1.1 History of the the Janus API

1.1.1 The beginnings

The software dates back to the year 1995. Back then, Tandem offered a product named *Remote Server Call, RSC,* that enabled a client written in C to control a Pathway application, i. e. perform transactions and send messages to named server classes. The Product is still available from HP.

A customer wanted to do the same from SAP...

The result was SAP_RSC, a small program running on a system supporting both APIs (SAP's CPI-C and Tandem's RSC) which acts as a gateway between both worlds. With this application, the name Janus came into my mind: Janus is a technology to bridge these two worlds, SAP and Tandem, by looking in each direction with a face of its own.[1]

1.1.2 Getting independent

SAP_RSC has one disadvantage, it depends on two libraries from two vendors which have to be available on the same platform. RSC has features which are not needed in the SAP context. So I decided to create my own implementation to reach the NonStop system via TCP/IP: TCP_TAN.[2] I had to shorten the filename to TCPTAN because of naming restrictions in the Guardian filesystem. The underscore symbolizes the bridge between the two "halves" of each Janus gateway implementation.

SAP_RSC was replaced by SAP_TCP and TCPTAN. SAP_TCP runs somewhere in the SAP landscape and TCPTAN on the NonStop server. TCPTAN does essentially what RSC does but with a totally different API and some differences in functionality and handling.

1.1.3 Extending the view

Once independent from RSC and having created my own TCP/IP based protocol, it was only a small step to extend the audience. The API can be used in applications outside the SAP world. The original C implementation is essentially unchanged from the beginnings of SAP_TCP and TCPTAN, only a few useful additions have been made, like UMS (unsolicited messages) and nowait I/O.

The original API is complicated enough to warrant a simplification: JANUSDLL is a stripped down, Windows only implementation which hides some of the complexity for simple clients. These can even be written in other languages like Delphi or Visual Basic.

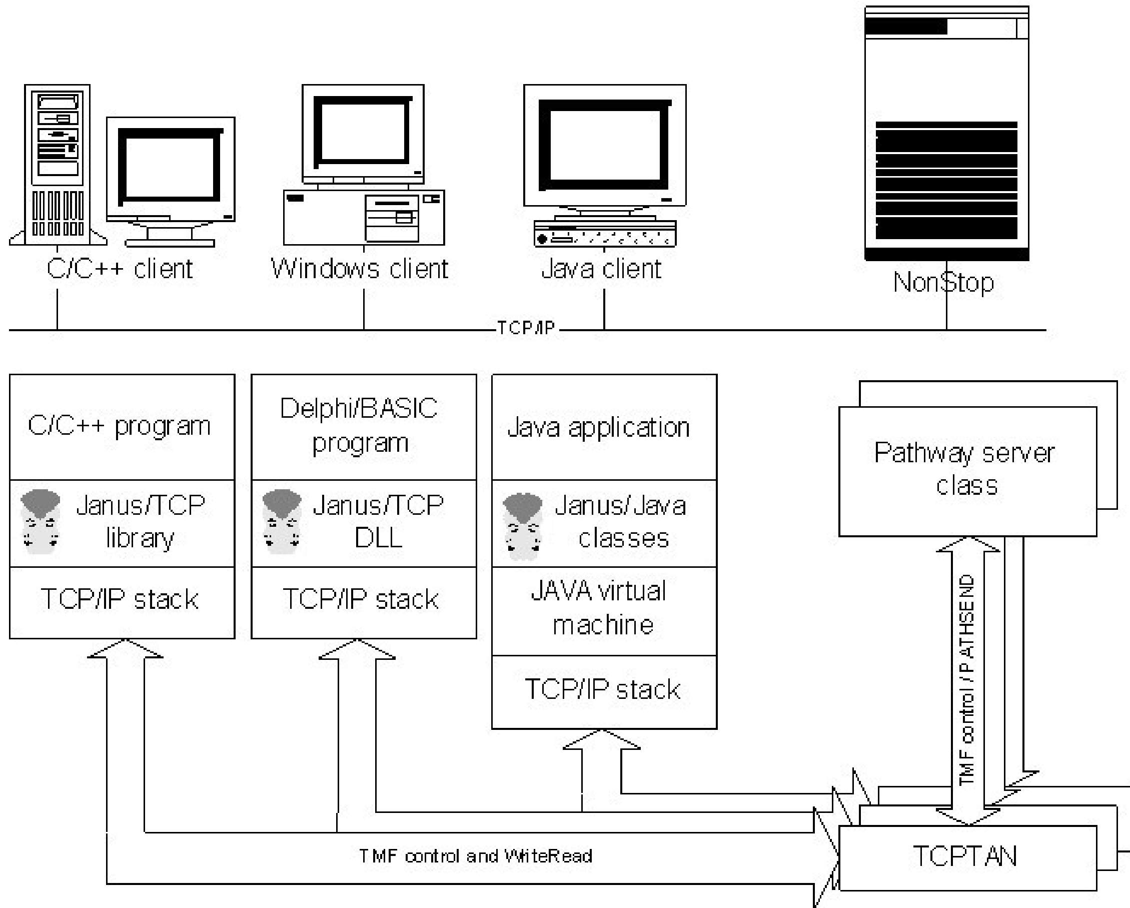
The biggest step forward was a complete reimplementation of the protocol in Java. With Janus/JAVA,connectivity to Pathway servers is totally platform independent. There are no legacy native libraries involved. Janus/JAVAdoes not only the communication, it aids in translating message structures between the Java and the Tandem COBOL world.

1 There is even a version SAP_IBM which connects an R/3 system to a CICS application. It's a customer specific implementation. I can't tell wether the software is still in use.

2 There was an intermediate solution, TCP_RSC, a program running on a Unix platform to bridge between the Janus TCP/IP protocol and the RSC product, but this is history.

1.2 Basic functionality

1.2.1 Overview



1.2.2 What Janus is *not* meant for

Janus does not attempt to replace RSC in every respect, it is simply a different attempt to solve a similar problem, and not a clone.

TCPTAN is configured into the TCP/IP listener on the NonStop system. When a client makes a connection, a separate process is started which handles the communication and owns the TMF transactions. The implications are twofold:

- The installation requires virtually no administration on the NonStop system. There is no process to monitor, configure or restart, except the listener provided by the system.
- TCPTAN isn't multithreaded: Each connection requires its own process. The RSC TDP is one process handling hundreds of simultaneous clients and therefore suitable for direct (PC) client connections. Janus shouldn't be used in such a scenario because of the resources required on the host. It is more suitable for middleware applications which pool connections, or for point to point connections as in the SAP scenario.

A major implication has to do with unsolicited messages, sent from Guardian processes to clients. In RSC, a client is configured as a virtual terminal connected to a central process, the TDP. UMS messages are sent to the TDP and contain a terminal name to address the final destination. To send an UMS message to a Janus client, the process name of the corresponding TCPTAN incarnation must be made public somehow by the application. The API helps to accomplish this.

1.2.3 Installation

Installation is a one time procedure. It requires copying an executable file, TCPTAN, to the Guardian file system and changing the TCP/IP configuration.

1. Use ftp to copy *tcptan.txe* to $SYSTEM.JANUS.TCPTAN,700[1]
2. Add a line to your PORTCONF file:
   `2000 $SYSTEM.JANUS.TCPTAN`
3. Restart your listener.

You can start a separate listener instance if you want to run TCPTAN under a special account or if you can't change the system configuration. The port number, here 2000, can be chosen freely. 2000 is the client default value.

1.2.4 Establishing a connection

To establish a connection, you need to provide a host name, a port number and the name of the default PATHMONprocess where your application resides. There is no login procedure, the messages are sent on behalf of the user who started the TCP/IP listener.[2]

The procedure names differ from implementation to implementation. The C API returns a pointer to a `TCP_HOST` structure which is automatically allocated and contains the details of the connection. The Java API returns a `Connection` object which not only contains all information about the connection but also provides the methods to talk to the host system. The DLL keeps track of connections internally. It was originally designed to allow for just one simultaneous connection but that has been extended recently.

The connection is bound to a TCPTAN process which acts as a gateway. Disconnecting the link stops the TCPTAN process. The name of the process can be queried from the connection object for monitoring purposes or to allow the client to receive UMS messages.

1.2.5 Handling transactions

1.2.5.1 Starting and ending a transaction

Janus provides calls to begin, end or abort (rollback) a transaction. TCPTAN owns the TMF transaction. When a connection is disconnected and a transaction is still in progress, it is rolled back by TMF.

1.2.5.2 Handling automatic transactions

Automatic transaction handling has been borrowed from RSC. The client provides two 16 bit boundaries which are checked against the first two bytes of the result message from the server, interpreted as a reply code. If the reply code is between the boundaries, the transaction is automatically committed by TCPTAN. This saves an additional I/O operation over the network.

When the reply code is outside the given interval, the transaction is left open. The client must then decide whether to abort or to commit it.

1.2.6 Sending messages

1.2.6.1 What is a message?

A message is just a bunch of bytes. Interpretation is up the the server and the client. Janus can handle messages up to 32000 bytes. Typically, messages are defined in the DDL data dictionary. Janus comes with a tool to read this information and convert it to Java objects. For mapping DDL structures to C or other languages, the DDL compiler provides the conversion commands.

1 I can't provide an Itanium object at the moment but a file code 100 RISC version is available on request.

2 You can probably change some file attributes of TCPTAN so that it assumes the identity of the owner of the TCPTAN object file. See the FUP manual for details.

1.2.6.2 Destinations

Messages are sent to specific destinations. A destination can be one of the following:

- a Pathway server class name,
- a Pathway server class name in a specific pathmon,
- a Guardian process name.

The usual case is the first. The message goes to the named server class in the standard pathmon the name of which has been set when the connection was set up. If you need to send a message to a specific pathmon process, just append '@' and the pathmon name to the destination, e. g. `"MY-SERVER@\TD01.$PM2"`. As you can see, Expand networks are fully supported.

Instead of a server class name, you can send a message to a named process. This is done when the name starts with a backslash or dollar sign. The process is opened and closed for each I/O.

1.2.6.3 Timeouts and Nowait I/O

A message can be sent with a timeout value in hundredths of seconds. If the server does not respond within the time given, the I/O is canceled and an error is returned to the client. The TMF transaction is not affected by the timeout, it's up the client to handle the consequences.

A timeout value of -1 disables timeout, the client blocks until the server replies.

A timeout value of zero starts a nowaited I/O operation which returns immediately. Just one simultaneous operation per connection can be started in this manner. To complete the operation, a call to the *awaitio* API function is required. This call has its own timeout value. When the I/O is not completed within the timeout, the operation returns with an error, but is *not* canceled automatically. It must be completed by another call to *awaitio* or canceled by a call to the *cancel* API function.

1.2.7 Receiving UMS messages

A Janus client can receive unsolicited messages from a process on the NonStop system. There are two ways to set up the necessary environment:

- A call to the *AcceptNew* API function establishes a second TCP/IP link to the already running TCPTAN process and returns its control structure to the client. This might be impossible, if there is a firewall between the client and the machine running TCPTAN, because the port number is determined on the fly and the link is established backwards from the NonStop system to the client.
- To overcome the firewall problem, the client can establish a dedicated connection with the string `"UMS"` instead of a pathmon name. This starts another instance of TCPTAN which can only be used for UMS messaging.

In order to handle incoming UMS messages, a separate thread of execution should be created within the client software. This thread can use the *ReceiveData* API function to read an incoming message and reply with the respective *Send* API calls to the sender.

The application must provide a means of broadcasting the Guardian process name of the associated TCPTAN process to the potential UMS senders on the NonStop system, e. g. by placing it in a host database through a dedicated pathway server class.

The format of the UMS message is compatible to RSC. The terminal and alias fields are transmitted to the client and can be freely used.

1.2.8 Error handling

There are mainly two classes of errors that can occur: Communication failures on the network link or problems with the application on the NonStop host. The C API returns an error code and provides a means to get the details of the failure. In case of a host application failure, additional fields are returned. They are filled similar to what RSC would return in the same situation, but TCPTAN is less verbose than RSC. In any case, a short text is returned.

The Java API throws either a `CommException` or a `TandemException`, in case of a failure. The exception object contains the detailed error information.

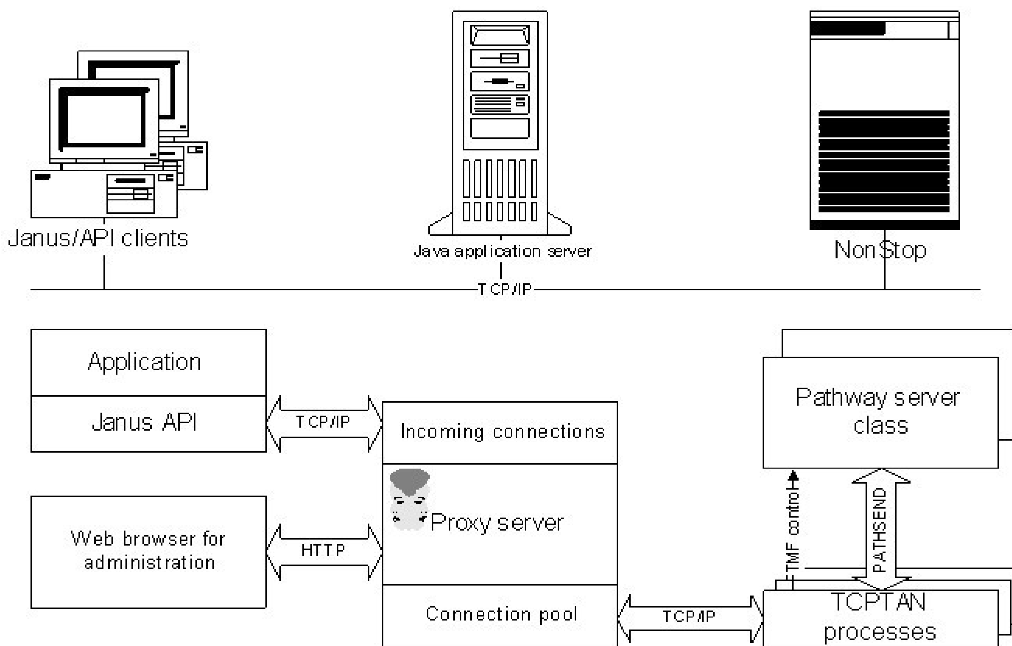
1.2.9 Connection pooling

Pooling connections helps to reduce resource requirements on the NonStop system. Connection pooling is directly available to Java clients and indirectly, through the Janus Web Application, for any other API client.

1.2.9.1 Connection pools in Java

The Java API is fully explained in the JavaDoc documentation available here: http://www.mvcsys.de/doc/javadoc/janus/api/index.html. The comments are in German but that might change in the near future. Look at the `ConnectionPool` class description.

1.2.9.2 The web based proxy



The Janus Web Application can be installed in any servlet container, such as Apache Tomcat, IBM WebSphere or even SAP NetWeaver. It listens on a TCP/IP port for incoming connections and maps them to a pool of connections to the NonStop host. As long as no TMF transaction or nowait I/O is open, the connections are returned to the pool right after the I/O. In case of an open transaction, the connection is reserved until a commit or rollback. This is fully transparent to the client except for one fact: The client cannot set a default pathmon name, because the process name is defined in the connection pool configuration. The `"server@pathmon"` notation is still possible.

2 API details

2.1 The Windows DLL

2.1.1 Installation and general remarks

The Windows DLL is designed for client applications which need to access Pathway servers and manage transactions. The original version allowed just one connection to the NonStop system. It is not necessary to keep track of a connection object after the connection is established. The current version supports juggling with more than one simultaneous connection, if necessary. Unsolicited Messaging is not supported.

The DLL is compiled with Borland C and works with Delphi or Visual Basic. Other languages should work likewise. The file *janusdll.dll* contains everything you need. Just put it somewhere in the path or in the current directory of your application.

2.1.2 Linking the DLL into your application

The way of linking the DLL into your application depends on the programming language. The exported names are all uppercase and use the *stdcall* calling convention.

2.1.2.1 Visual Basic

The following example shows how the call to `JANUSCONNECT` is defined in the .net version of Visual Basic:

```
Protected Declare Ansi Function JANUSCONNECT Lib "JANUSDLL.DLL" _
    (ByVal host As String, ByVal port As Integer, ByVal pathmon As String) _
    As Integer
```

Visual Basic 6 and earlier shouldn't be too different.

Your .net application must be given permission to call native code, otherwise a security exception will be thrown. Look into directory `vb\.net` in the installation archive for a complete implementation and a sample client.

2.1.2.2 Delphi Pascal

The same example as above in Delphi Pascal looks like this:

```
const JANUSDLL = 'JANUSDLL.DLL';

function JanusConnect( pcHost: pchar;
                       nPort: longint;
                       pcPathmon: pchar ): longint;
         stdcall external JANUSDLL name 'JANUSCONNECT';
```

A Pascal unit is available on request.

2.1.2.3 C/C++

For C and C++ programmers, the header file *janusdll.h* contains the necessary definitions. The linker definition file *janusddl.def* lists the exported functions.

For Microsoft compilers, the header file needs to be modified. The version provided uses Borland syntax.

My advice for C programmers: Use the full C API and add the source files to your project. This provides more flexibility and aids in debugging.

2.1.3 API calls

The API calls are shown in C notation, because the DLL is written in C. It shouldn't be too hard to mentally translate the descriptions into your programming language of choice. Strings, like host or server class names, are C strings, i. e. they are terminated by a binary zero character. The above mentioned declarations in Pascal or Basic take care of the conversion automatically.

2.1.3.1 Connection Handling

The DLL can handle a maximum of 10 connections, numbered from 0 to 9. They are maintained in an internal table. The following call selects a specific one:

```
int JANUSSELECT( int connection );
```

The number of the previously selected connection is returned so that you can switch back if needed. If you only need one connection, it's not necessary to call this function at all. Only active connections reserve any resources. Call `JANUSSELECT` before any other API call to direct it to the selected connection! This type of juggling with the connections is *not* thread save! If you want to create a multithreaded application with multiple connections, you must use the full C API.

Sample:

```
oldconn = JANUSSELECT( newconn );
```

In order to establish a connection, you need to call the following function:

```
int JANUSCONNECT( char *host, int port, char *pathmon );
```

The parameters are the TCP/IP host name, port number and the default Pathmon name. The function returns zero if the connection could be established, -1 in case of an error. The most recent call to `JANUSSELECT` determines, which of the possible 10 connections is established.

Example:

```
ret = JANUSCONNECT( "tandem", 2000, "$PM" );
```

This connects to the host "tandem" at port number 2000. The default pathmon name is "$PM".

When you are done or in case of network errors, you should disconnect from the host with the following function:

```
int JANUSDISCONNECT( void );
```

This disconnects the current connection. The return value is always zero.

Example:

```
ret = JANUSDISCONNECT();
```

You can disconnect all active connections with a single call:

```
int JANUSDISCONNECTALL( void );
```

The return value is always zero.

Example:

```
ret = JANUSDISCONNECTALL();
```

2.1.3.2 Transactions

All transaction handling is done through a single API call:

```
int JANUSTRANSACTION( int mode );
```

The type of the call is determined by parameter `mode`. In C, this is a single character. In other languages you might need to convert it to the corresponding ASCII code. A return value of zero indicates a successful call, -1 indicates an error.

Transaction Type	*ASCII Value*	*Symbolic Name*	*Description*
'B'	66	TT_BEGIN	Start a TMF transaction on the NonStop system.
'E'	69	TT_COMMIT	End the current TMF transaction. Commit all database updates.
'R'	82	TT_ROLLBACK	Roll back the current TMF transaction. Cancel all database updates.

The C header file defines the symbolic constants. It is advisable to do the same in your programming language.

Examples:

```
ret = JANUSTRANSACTION( TT_BEGIN );
ret = JANUSTRANSACTION( TT_COMMIT );
ret = JANUSTRANSACTION( TT_ROLLBACK );
```

Automatic transactions as outlined in the previous chapter are not available in the DLL.

2.1.3.3 Sending messages

To send a message to a destination -either a server class name (@pathmon name, if desired), or a named process-, you have to provide two buffers and their respective lengths. The buffers are arrays of bytes. It depends on the programming language how data structures are mapped to these arrays. The NonStop system uses single byte character sets. If your environment uses Unicode, a translation must be done in your application.

A message is sent like this:

```
int ENTRY JANUSSENDDATA( char *server,
                         char *buffer, int len,
                         char *reply,  int replylen );
```

Janus sends `len` bytes to the `server` from the `buffer` and places the answer into `reply`, which is at most `replylen` bytes long. The actual length is returned by the function. If the return value is negative (-1), an error has occurred. `buffer` and `reply` may point to the same memory area.

Example:

```
reply_len = JANUSSENDDATA( "PW823",
                           message, message_len,
                           reply, reply_len );

reply_len = JANUSSENDDATAWITHTIMOUT( "PW823",
                                     message, message_len,
                                     reply, reply_len,
                                     300L );
```

A variant of the API function exists which allows to specify a timeout value in hundredths of seconds:

```
int ENTRY JANUSSENDDATAWITHTIMEOUT( char *server,
                                    char *buffer, int len,
                                    char *reply,  int replylen,
                                    long timeout );
```

A timeout of -1 is treated as an infinite wait, while a timeout of zero starts a nowaited send. In the latter case, `reply` and `replylen` are ignored.

Nowait sends have to be completed or canceled by either one of the following calls:

```
int ENTRY JANUSAWAITIO( long timeout, char *reply, int replylen );
```

```
int ENTRY JANUSCANCEL( void );
```

`JANUSAWAITIO` returns the actual length of the reply, if the call completes normally, `-1` for an error, or -2 in case of a timeout. Timeouts do not automatically terminate the operation, your code has to do it.

Example:

```
ret = JANUSAWAITIO( 300L, reply, replylen );
if ( ret == TIMEOUT ) JANUSCANCEL();
```

The constant `TIMEOUT` is defined in the C header file as `(-2)`.

2.1.3.4 Error information

If any of the API calls returns a value of `-1`, an error has occurred. The following call returns an error message in plain text:

```
int ENTRY JANUSERROR( char *buffer, int len );
```

You provide a buffer for the text and it's maximum length. The returned string is terminated by a binary zero. The return value is the actual string length without the delimiter.

Example:

```
char error_message[ 500 ];
len = JANUSERROR( error_message, 500 );
```

2.2 The C API

2.2.1 Introduction

The C API is the full blown Janus API which is also used in the TCPTAN gateway. This is, in every respect, the reference implementation of the underlying protocol. It is written in standard ANSI C and should compile easily on almost any ANSI compatible system. Because it uses TCP/IP sockets for communication, a matching socket library is required. You *must* declare a global character array of total length 9 with the name `MyName` in your program and store a descriptive short name of your application there, otherwise you get linking errors.

The API supports thread safe client access to the NonStop server. All functions use a handle, which is a pointer to a communication structure. The handle is returned by the `TCP_Connect` API call. Upon disconnect by a call to `TCP_Disconnect`, the communication structure and all associated resources are freed.

Clients can implement UMS services to NonStop processes. On request, the API creates a separate connection with its corresponding communication structure. The client should start a dedicated thread to handle incoming messages.

You can even create a Janus server with the API. It listens on a port and accepts messages in the same way as TCPTAN does. I've used this feature in the past to implement dummy servers to act as mockups for Janus client applications. This has made me independent from a real NonStop system during development.[1]

2.2.2 Distributed files

2.2.2.1 Header files

The API comes with four header files, three of which are always needed: *defs.h*, *util.h* and *tcpcomm.h*. A fourth header, *ums.h*, is only needed for unsolicited messaging.

Header file	***Description***
tcpcomm.h	The main header which includes the next two files. It contains definitions and prototypes for all Janus API calls.
defs.h	Common definitions such as `OK` or `TRUE`, error codes and the transaction types.
util.h	Definitions for to the utility library *util.c*. Its functions are internally used by the API for message formatting, trace file handling, etc.
ums.h	This file should be used on the NonStop system. It contains the necessary definitions to send UMS messages to Janus (or RSC) clients. Its not needed for client programming.

A client needs to include *tcpcomm.h*. C++ clients must surround the `#include` directive with an `extern "C"` declaration.

2.2.2.2 C source files

The API is contained in just two source files, *tcpcomm.c* and *util.c*. They implement the functions defined in their respective header files. Simply add both files to your project or makefile. I've successfully compiled the sources under Windows (with Borland and Microsoft compilers), OS/2, Linux and various Unix systems with the system compilers (ANSI required!) or GCC. They can be compiled on the NonStop system, too.

If *tcpcomm.c* does not compile, the most probable cause is an incompatibility with the socket headers. The file starts with a lengthy section of includes which are grouped by conditional compilation directives. Here is the point where system dependent patches might be necessary.

1 A real life application of this technique is used in Janus/IDOC.

2.2.2.3 Sample Clients

The package contains two more source files, *tclient.c* and *uclient.c*. The former is a sample client which sends a message to a server on the NonStop system and which accepts UMS messaging. The latter is a Guardian client that can send UMS messages to a running instance of *tclient*. Compile it on your NonStop system!

2.2.3 API calls

2.2.3.1 Connection handling

All connection information is stored in a `TCP_HOST` structure:

```
struct {
    int connected;                      /* TRUE while connected */
    int option;                         /* Options, e. g. NOWAIT */
    int socket_recv;                    /* TCP/IP socket for Recieve */
    int socket_send;                    /* TCP/IP socket for Send */
    unsigned bytes_written;             /* Statistics */
    unsigned bytes_received;            /* ditto */
    ERROR_CODE error_code;              /* Last error code */
    ERROR_TYPE error_type;              /* Type of error */
    char error_text[ 544 ];             /* error message text */
    unsigned error_no;                  /* Guardian error */
    unsigned subsystem;                 /* Subsystem */
    unsigned subsystem_error;           /* Subsystem error */
    unsigned extended_error;            /* Extended error */
    unsigned error_class;               /* Error class */
    TRANSACTION_TYPE transaction_type;  /* Transaction type */
    int success_lo, success_hi;         /* Automatic transactions */
    TCP_MESSAGE_TYPE message_type;      /* Type of last message received */
    unsigned user_len;                  /* Length of user data */
    char *user_data;                    /* Pointer to user data */
    char *destination;                  /* Pointer to destination */
    long timeout;                       /* Timeout value x0.01s */
    unsigned io_len;                    /* Total buffer length */
    char io_buffer[ 33000 ];            /* I/O buffer */
};
```

Call one of the following functions to establish a connection:

```
TCP_HOST *TCP_Connect( char *host_name, int port, char *pathmon );
TCP_HOST *TCP_Accept( int argc, char **argv );
TCP_HOST *TCP_AcceptNew( TCP_HOST *host, int port );
```

`TCP_Connect` creates a client connection. The parameters `host_name` and `port` determine the Non-Stop system where TCPTAN runs, `pathmon` is the default pathmon process name.

`TCP_Accept` creates a server connection. The parameters `argc` and `argv` are the parameters to `main` of the C runtime. The possible arguments differ from operating system to operating system. In Unix or Linux, a `"-"` or `"-1"` argument sets up the program to be started from *inetd*. The argument `-P<port>` makes the program listen on the given port number. In this case, only one instance of the server can run on the same port.

`TCP_AcceptNew` creates an UMS connection with an already running instance of TCPTAN. `host` must point to a handle returned by `TCP_Connect`. The port number given may be `-1` or a positive short integer. It is used for a *reverse* TCP connection from TCPTAN to the client. `-1` means that the port number is determined automatically. The returned handle can only be used for UMS messaging. If `TCP_AcceptNew` hangs, there is a firewall in the way. You can then start a dedicated instance of TCPTAN by calling `TCP_Connect` with a pathmon name of `"UMS"`. In any case, the TCPTAN process name can be found in the `user_data` field of the returned handle. Your application must broadcast this name to any potential UMS senders.

2.2.3.2 Disconnection

This call terminates a connection and frees all resources:

```
int TCP_Disconnect( TCP_HOST *host );
```

It always returns `OK`.

2.2.3.3 Errors

If `TCP_Connect` or `TCP_Accept` fail, they nevertheless return a non-`NULL` handle, with the field `connected` set to `FALSE` and the error fields properly filled. Use `TCP_Disconnect` to free the handle! If the returned handle is null, there was a memory allocation problem. `TCP_AcceptNew` sets the error fields in the parent handle instead, and returns a `NULL` handle to the caller, if the connection cannot be established.

All other API functions return either a data length, `OK` (0) or `NOT_OK` (-1). A return value of `NOT_OK` indicates an error and the error fields in the `TCP_HOST` structure pointed to by the handle are filled.

2.2.3.4 Transactions

All transaction handling is done through a single API call:

```
int TCP_Transaction( TCP_HOST *host, TRANSACTION_TYPE mode,
                     int success_lo, int success_hi );
```

The type of the call is determined by parameter `mode`. `TRANSACTION_TYPE` is an enumeration with the following values:

Transaction type	*Value*	*Description*
`TT_NONE`	`'N'`	No transaction is in progress. Normally, this state is only used internally, but you can use it as an argument to `TCP_Transaction` to check if the connection is still functional. TCPTAN treats it as a no-operation.
`TT_BEGIN`	`'B'`	Start a TMF transaction on the NonStop system.
`TT_COMMIT`	`'E'`	End the current TMF transaction. Commit all database updates.
`TT_ROLLBACK`	`'R'`	Roll back the current TMF transaction. Cancel all database updates.
`TT_SINGLE`	`'S'`	TCPTAN starts a transaction and stores the boundary values `success_lo` and `success_hi` for automatic transaction handling (see chapter 1.2.5.2).
`TT_AUTOCOMMIT`	`'A'`	Works just like `TT_SINGLE` but doesn't start a transaction. Both calls just affect the next server message.

2.2.3.5 Sending messages

To send a message to a destination -either a server class name (@pathmon name, if desired), or a named process-, you have to provide two buffers and their respective lengths. The buffers are arrays of bytes which can be mapped to C structures generated by the DDL compiler. Be aware that some architectures have alignment restrictions which are incompatible with the DDL generated definitions!

A message is sent with either of the following routines:

```
int TCP_SendData( TCP_HOST *host,
                  char *destination,
                  void *buffer,
                  int len,
                  void *reply,
                  int reply_len );
```

```
int TCP_SendDataWithTimeout( TCP_HOST *host,
                             char *destination,
                             void *buffer,
                             int len,
                             void *reply,
                             int reply_len,
                             long timeout );
```

Janus sends `len` bytes to the `server` from the `buffer` and places the answer into `reply`, which is at most `reply_len` bytes long. The actual length of the reply is returned by the function. If the return value is `NOT_OK` (-1), an error has occurred. `buffer` and `reply` may point to the same memory area.

The second variant allows for a timeout value in hundredths of seconds. A value of -1 is treated as an infinite wait, while a timeout of zero starts a nowaited send. In the latter case, `reply` and `reply_len` are ignored.

A nowaited I/O has to be completed or canceled by either one of the following calls:

```
int TCP_AwaitIo( TCP_HOST *host,
                 long timeout,
                 void *reply,
                 int reply_maxlen );

int TCP_Cancel( TCP_HOST *host );
```

`TCP_AwaitIo` returns the actual length of the reply, if the call completes normally or `NOT_OK` in case of an error. To determine, if the error was a timeout, check `host->error_no` for a value of 40 which is the corresponding Guardian error code. Timeouts do not automatically terminate the operation, your application has to do it.

2.2.3.6 Server Programming

A server receives and replies to messages. It doesn't matter whether it is an UMS thread within a client or a standalone server process that has called `TCP_Accept`.

Receiving a message means calling the following function in a loop:

```
int TCP_RecvData( TCP_HOST *host );
```

The message contents is stored in `host->user_data`, its length in `host->user_len`. The type of the message is left in the field `host->message_type`. This should be `MT_DATA` for ordinary data messages. Your code is free to handle different types of messages, like transactions, but this is left to you.

If `TCP_RecvData` returns `NOT_OK`, the connection is very likely to be broken and it is advisable to exit the server loop and close the connection.

Your code must reply to all incoming messages with either of the following calls:

```
int TCP_SendReply( TCP_HOST *host,
                   unsigned int reply_len,
                   char *reply );

int TCP_SendAck( TCP_HOST *host, char *text );

int TCP_SendNak( TCP_HOST *host, int err, char *text );
```

`TCP_SendReply` sends a reply message with data back to the sender, `TCP_SendAck` just an empty positive acknowledge with an optional text message. `TCP_SendNack` sends an error message with an appropriate error code back. The error code should resemble some Guardian filesystem error.

2.2.3.7 Tracing and debugging

You can set the environment variables `TRACE` and/or `DEBUG` to `ON`, in order to create a trace file or additional debugging output to the console. The name of the trace file is the name of the application as stored in `MyName` plus the filename extension `.trc`. Trace and log files (created by a call to the `log()` utility function, ending in `.log`) are placed in the current directory or the directory named in the environment variable `LOGDIR`.

2.2.3.8 Utility functions

The *util.c* library contains some useful functions as listed in the English translation of *util.h* below:

```
/*
 *  Data type for time and date information
 */
typedef struct {
    short year,
          month,
          day,
          hour,
          minute,
          sec,
          hsec;
} TIMEBUFF;

/*
 *  Is debugging or tracing turned on by the
 *  environment variables DEBUG and TRACE?
 */
int DebugON( void );
int TraceON( void );

/*
 *  Handle the trace file
 */
void TraceClose( void );
extern FILE *TraceFile;

/*
 *  Call an external script "meldung" which might be used
 *  to send an error message to an operator console
 */
void Meldung( int level, char *text );

#define LVL_INFO    0
#define LVL_WARNING 1
#define LVL_ERROR   2
#define LVL_FATAL   3

/*
 *  Write a message to the application log file
 */
void Log( int level, char *form, ... );

/*
 *  Convert binary values to COBOL-like numeric strings and back
 */
char *CnvIntChar( int number, char *buff, int size );
int  CnvCharInt( char *buff, int size );
char *CnvLongChar( unsigned long number, char *buff, int size );
unsigned long CnvCharLong( char *buff, int size );
int  IsNumeric( char *buff, int size );

/*
 * Convert fixed length character data to zero terminated C strings
 * and back
 */
char *CnvCharStr( char *buffin, char *buffout, int size );
char *CnvStrChar( char *buffin, char *buffout, int size );

/*
 *  ISO8859 <-> EBCDIC Conversions
 */
void CMCNVI_( unsigned char *buff, int len, int *rc );
void CMCNVO_( unsigned char *buff, int len, int *rc );
unsigned char *CnvAsciiEbcdic( char *ascii, unsigned char *ebcdic, int size );
char *         CnvEbcdicAscii( unsigned char *ebcdic, char *ascii, int size );
```

```
/*
 *  Time and date handling
 */
TIMEBUFF *GetDateTime( TIMEBUFF *time_buff );
char *CnvDate( TIMEBUFF *time_buff, char *buff );
char *CnvTime( TIMEBUFF *time_buff, char *buff );
char *CnvTimestampNumeric( TIMEBUFF *time_buff, char *buff, int size,
                           BOOL terminate );
TIMEBUFF *CnvNumericTimestamp( char *buffin, TIMEBUFF *time_buff, int size );

/*
 *  SQL time and date conversions
 *  JJJJ-MM-TT:hh:mm:ss.ffffff"
 */
#define TIMESTAMP_YEAR 0
#define TIMESTAMP_MONTH 1
#define TIMESTAMP_DAY 2
#define TIMESTAMP_HOUR 3
#define TIMESTAMP_MINUTE 4
#define TIMESTAMP_SECOND 5
#define TIMESTAMP_FRACTION 6
char *CnvTimestampSql( TIMEBUFF *time_buff, char *buff,
                       int start, int start_precision,
                       int end, int fraction_precision );
TIMEBUFF *CnvSqlTimestamp( char *buffin, TIMEBUFF *time_buff, int start );

/*
 *  Some macros to simplyfy moves from and to COBOL messages
 */
#define PUT_STR( str, dest ) CnvStrChar( str, dest, (int) sizeof( dest ) )
#define PUT_INT( num, dest ) \
        CnvIntChar( (int) num, dest, (int) sizeof( dest ) )
#define PUT_SINT( num, dest ) \
        CnvIntChar( (int) num, dest, -((int) sizeof( dest )) )
#define PUT_LONG( num, dest ) \
        CnvLongChar( (unsigned long) num, dest, (int) sizeof( dest ) )
#define GET_STR( src, str  ) CnvCharStr( src, str,  (int) sizeof( src  ) )
#define GET_INT( src )       CnvCharInt( src, (int) sizeof( src ) )
#define GET_LONG( src )      CnvCharLong( src, (int) sizeof( src ) )
#define NUMERIC( src )       IsNumeric ( src, (int) sizeof( src ) )
#define TRIM( str )          CnvCharStr( (char *) str, (char *) str, \
                                         (int) strlen( str ) )
#define CLEAR( dest )        memset( &dest, 0, sizeof( dest ) )

#define PUT_EBCDIC( str, dest ) \
        CnvAsciiEbcdic( CnvStrChar( str, dest, (int) sizeof( dest ) ), \
                        dest, (int) sizeof( dest ) )

#define GET_EBCDIC( src, str ) \
        CnvEbcdicAscii( CnvCharStr( src, str,  (int) sizeof( src ) ), \
                        str, (int) strlen( str ) )

#define PUT_TIME( src, dest ) \
    CnvTimestampNumeric( src, (char *) &dest, (int) sizeof( dest ), FALSE )
#define GET_TIME( src, dest ) \
    CnvNumericTimestamp( (char *) &src, dest, (int) sizeof( src ) )

/*
 *  More macros
 */
#ifndef max
#define max(a,b) ((a)>=(b)?(a):(b))
#endif

#ifndef min
#define min(a,b) ((a)<=(b)?(a):(b))
#endif
```

2.3 The Java API

2.3.1 Introduction

When I decided to port Janus to Java I was pretty sure I did *not* want to create a thin Java wrapper around the C libraries, as many some smart companies do with their "Java enabled" drivers. If you go the Java way, go it one hundred percent!

The result is a complete reimplementation of the TCP/IP based communication protocol, 100% pure Java and thus totally independent of the underlying operating system. Janus/JAVAis fully object oriented. Messages are objects as are connections. Errors are handled via exceptions which wrap the detailed error codes.

The package names, `janus.*`, do not follow the standard Java naming conventions.[1] They should, and probably will be named `de.mvcsys.janus.*`. This will introduce incompatible changes to the existing API and I'm a little reluctant to perform the transition right now. Keep in mind, that the API is subject to change.

The full JavaDoc documentation can be found online: http://www.mvcsys.de/doc/javadoc/janus/api.

2.3.2 Installation

Janus comes as a single library, *janus.jar*. The library must be part of the class path of your application. One way to install it is the `jre\lib\ext` directory of your JVM. This way, Janus becomes a system extension and is available to all applications.

For web applications, the right place to put *janus.jar* is `WEB-INF/lib`. You can simply deliver it together with your web application and need not tamper with system directories.

The JVM must be version 1.3 or above.[2]

2.3.3 Generating Messages from DDL

2.3.3.1 Installation of ddl2java.exe

A Windows command line tool, *ddl2java.exe*,[3] converts messages defined in the DDL dictionary to Java classes. It is based on the Remote Enscribe / Remote SQL server which has to be installed on the development NonStop system. This is very similar to the installation of TCPTAN. Just put the RSQLSRV[4] object on your system and add the following line to the listener configuration:

```
3000 $SYSTEM.JANUS.RSQLSRV
```

The port number, 3000, can be chosen freely. Put *ddl2java.exe* somewhere in your path.

2.3.3.2 Starting ddl2java.exe

The program has two modes: messages or constants. Messages are generated from DDL `DEF` objects, Constants from DDL `CONST` objects.

Create a directory where the generated Java files shall reside and switch to it. Now start *ddl2java.exe*:

```
ddl2java.exe <options> <host:port> <dictvol> <item> [<item> ...]
```

1 When I started the development, I hadn't registered my *mvcsys.de* domain, yet.
2 Basic functionality is available for Java 1.1 implementations.
3 Do not confuse it with the jToolkit *ddl2java* tool. The name is the same, the functionality is not!
4 If you want to use the RSQL API, you'll need to SQL-compile the object. For *ddl2java.exe*, this is not necessary.

Common Parameters:

Parameter	*Example*	*Description*
<host:port>	tandem:3000	The host where RSQLSRV is installed.
<dictvol>	$DATA.MYDICT	The DDL dictionary subvolume. Expand names are allowed.
<item>	MY-STRUCT	One or more DDL items to convert or "*" for everything.

Common options:

Option	*Default*	*Description*
-u<user>	no default	Guardian user name.
-p<password>	no default	Guardian password, use -p for an empty password.
-P<package>	empty	Java package name for the generated classes.
-v	not set	Verbose operation.
-d<destdir>	current directory	The destination directory where all files are generated.

-t<nametrans file>	no translation	It is possible that the generated names collide and the Java sources do not compile. A simple text file helps to overcome this situation:
		`# ddl2java name translation for MYDICT`
		`#`
		`OLD-DDL-NAME-1 NEW-DDL-NAME-1`
		`OLD-DDL-NAME-2 NEW-DDL-NAME-2`

Options for generating messages:

Parameter	*Default*	*Description*
-a	not set	Regenerate all dependent definitions. Without this option, only changed or missing definitions are generated.
-texts -notexts	generate all texts	Add textual information from the dictionary to the generated classes. This includes PIC and VALUE clauses, etc.
-l<html link>	not set	If you have created an HTML documentation of your dictionary with *ddl2html.exe,*[1] *ddl2java.exe* can put the matching JavaDoc links into the generated sources. Give the base directory or url.

Options for generating constants:

Parameter	*Default*	*Description*
-c	not set	Create Constants.java, a static class with DDL CONST objects. The item list names constants, not definitions.

Option names and their values are *not* separated by a space!

1 *dd2html* is freely available on my website.

2.3.3.3 Sample script

You should use a script to generate or update the generated messages. Here is an example:

```
setlocal

set prog=c:\bin\ddl2java.exe
set host=tandem:3000
set dict=$DATA.MYDICT
set libdir=lib
set srcdir=src\myapp\msg
set package=myapp.msg
set opts=-uGROUP.USER -pSECRET -v -P%package% -d%srcdir%

if not "%1"=="" goto single

REM create all messages
DEL %srcdir%\*.java
%prog% %opts% -a %host% %dict% *

REM create all constants
%prog% %opts% -c %host% %dict% *
goto compile

:single
REM recreate a single message
%prog% %opts% %host% %dict% %1

:compile
javac -cp %lib%\janus.jar -d classes %srcdir%\*.java
jar cvf %lib%\mydict.jar -C classes *
javadoc -sourcepath src -classpath %lib%\janus.jar -d docs %package%

endlocal
```

The script assumes, that the directories `src\myapp\msg`, `lib` and `docs` exist and that *janus.jar* is placed in the `lib` directory. If you have generated an HTML documentation with *ddl2html.exe* in directory `html`, add the following option to the *ddl2java.exe* command line: `-l../../../html`.

2.3.3.4 What is generated from the dictionary?

Messages are generated as classes extending `janus.util.MessageElement`. Each element is itself an instance of `janus.util.MessageElement` or of a derived type. There a three different cases:

1. The element is a DDL base type: The generated element is an instance of the corresponding specialized class in package `janus.util.ddl`.
2. The element is a structure which is defined inline in the DDL source: The generated element is an inner class inheriting from `janus.util.MessageElement`.
3. The element references another DDL definition in the same dictionary: The generated element is an instance of the corresponding generated class, which is automatically loaded and generated by *ddl2-java.exe*.

An element with an `OCCURS` clause is mapped to an array of the corresponding type.

Each instance of `janus.util.MessageElement` is defined by a `byte[]` buffer and an offset into this buffer. Only the outermost definition owns the physical buffer, the inner components are mapped to this buffer at their respective offsets. All marshaling and unmarshaling is done by the setter and getter methods on the fly. This ensures the maximum possible compatibility with DDL and COBOL semantics. Redefinitions and tables are full supported. A structure can be treated as a single element of character type. Setting the value of the structure modifies all contained elements because the buffers are shared.

Not all DDL types are supported, notably floating point, logical values and SQL date/time literals.

Suppose the following DDL definition:

```
def booking-date heading "Booking Date".
  02 b-year  pic 9(4).
  02 b-month pic 99.
  02 b-day   pic 99.
```

Names are translated to mixed case, hyphens are removed except near a digit where they are converted to an underscore. `booking-date` becomes `BookingDate` with an uppercase "B" because it is a class name.

The generated class can be accessed from Java in the following way:

```
import myapp.msg.*;
```

...

```
BookingDate bookingDate = new BookingDate();
String heading = bookingDate.getHeadingText().toString();
String ddlName = bookingDate.getName();

bookingDate.set( "20080312" );

String dString   = bookingDate.toString();
int bookingYear  = bookingDate.bYear.get();
int bookingMonth = bookingDate.bMonth.get();
int bookingDay   = bookingDate.bDay.get();
```

There is bunch of meta information from the dictionary that is stored together with the data. It can be used in client applications to check values against given ranges, display column headings, etc. `ENUM` values and level 88 items are converted to `static final` members of the generated class.

Constants are all packed in the single class `<package name>.Constants`. Each constant is a member of type `janus.util.DdlText`. DDL-Names are left unchanged, except that hyphens are translated to underscore characters. To access the value of a constant in different formats, use one of the member functions `toString()`, `intValue()` or `longValue()`.

It makes a lot of sense to generate JavaDocs from the generated source files, because *ddl2java.exe* retains the dictionary comments and generates matching JavaDoc comments.