

TCP Communication, Data Exchange Methods: Creating a Messaging System

By Olivier Marolleau, Quality Support Engineer, 4D France.

Technical Note 25-11

Table des matières

Table des matières	2
Abstract.....	3
Introduction	3
Prerequisites	3
Application overview.....	4
Interface overview.....	5
1. Message input	5
2. Message Reception.....	7
3. Preferences Settings.....	9
4. Interaction with the Web Area	11
5. 4D Menu and the Web Area	12
6. Action centralization	14
Communication between users (or nodes).....	14
1. Exchange Contents	15
2. Request Encryption and Decryption	17
3. Message Acquisition.....	17
4. Message sending.....	20
Conclusion	21

Abstract

The TCP (Transmission Control Protocol) protocol plays a key role in network communications, particularly in client-server environments where data exchange reliability is essential.

In a 4D database, it is particularly useful for low-level interactions with automation systems or third-party software. It can also be used to manage communication channels. However, its implementation can be complex.

This technical note, through a simplified application example, aims to demonstrate a communication management approach in the context of inter-user communication.

Introduction

Following the deprecation of TCP commands and the 4D Internet Command plugin, 4D introduces two modern classes:

- **TCPListener** for data reception
- **TCPConnection** for data transmission

This update aims to optimize TCP communications. This technical note presents a practical implementation of these classes through the development of a simplified messaging application.

The objective is to demonstrate how TCPListener and TCPConnection can be used to manage inter-user communications, by providing a concrete example of architecture using the updated TCP Framework.

Prerequisites

This application is designed to work with version 20 R10 minimum in project mode.

It is recommended to have reviewed the documentation on:

- [TCPListener](#)
- [TCPConnection](#)

Application overview

This messaging application is a simplified version, so functions will be limited. For example, we will not handle proxies or network exploration.

Like all messaging applications, it will allow us to send messages, of course. But also, images, videos, and all types of files within a certain size limit.

An encryption key will be defined to secure exchanges.


The potential recipients of these messages will be referenced in a "nodes" section to memorize each one's address.

A channel will then allow designating an active list of message recipients. This will enable filtering of sends.


The username will serve as our base designation. This will allow simplified configuration. For example, if we assume that two users A and B wish to configure our messaging system. Each user launches the application, notes their name and address, communicates this to the other party. Each adds to the nodes list the address and username of the other user.

In the default channel, the only remaining step is to add the referenced user for message sending.

See the illustrated example below.

Communication : 

Address : Port :


User :  Name :

Nodes :


Address	User
192.168.0.20	B

Channels :

Name	Recipients
Default	B

Communication : 

Address : Port :

User :  Name :

Nodes :

Address	User
192.168.0.10	A

Channels :

Name	Recipients
Default	A

Interface overview

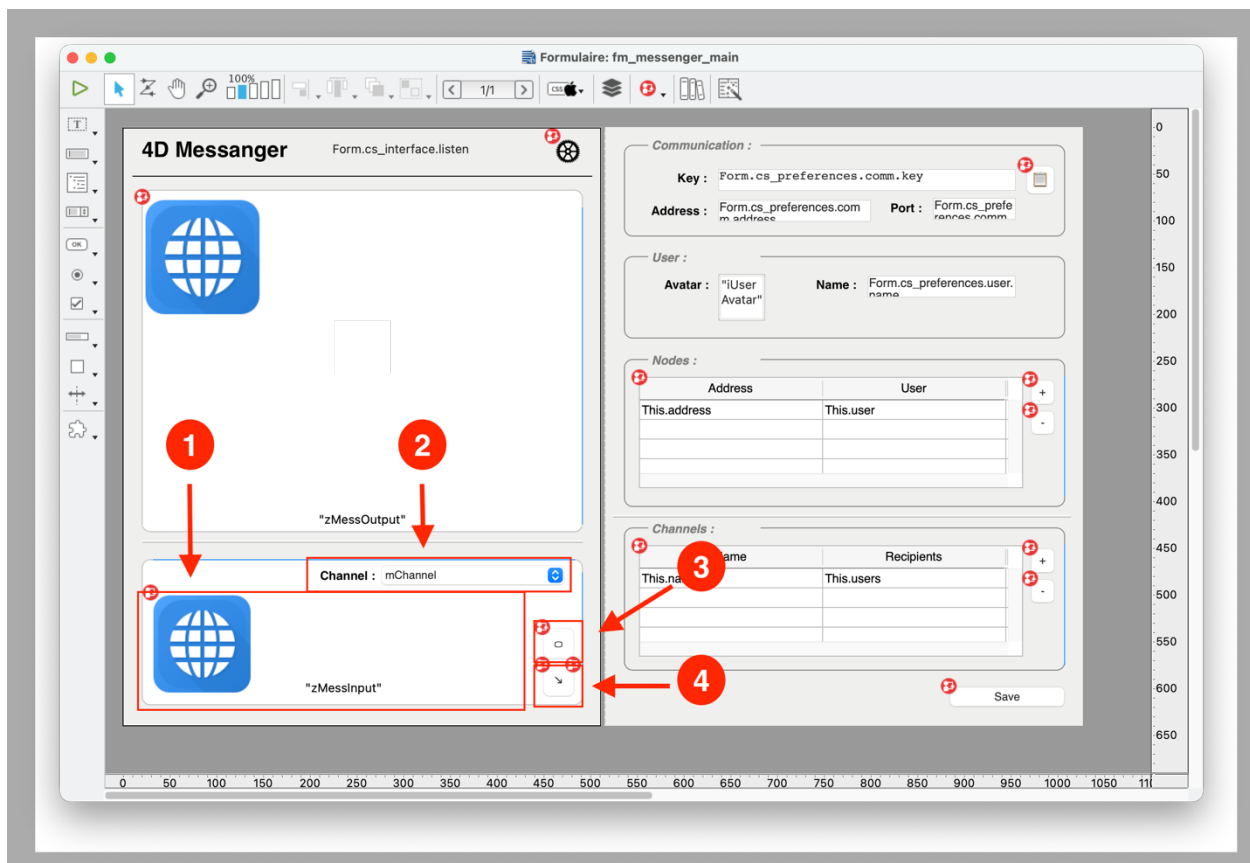
In this part of the technical note, the main features are detailed.

1. Message input

The application must allow text message input.

This becomes complicated when, at the same time, it must also allow input of videos, images, etc.

The only object type that can meet all the display conditions mentioned above is the web area. The application will therefore use the web area "zMessInput" for message input, as shown in the image below:



The input area includes:

1. The web area "zMessInput", which, as mentioned, will serve as the input area for the future message.
2. The popup "mChannel", allows users to choose the broadcast channel.
3. A "bClear" button, which allows the user to clear the message history
4. A "bSend" button, which allows the user to send the message

Sending is performed via the "bSend" button (4):

```
If (FORM Event.code=On Clicked)
```

```
Form.cs_interface.actioSendMessage()
```

```
End if
```

The broadcast channel (2) selects a predefined list of users as the only message recipients.

A message can be entered (1), and elements can be added via drag-and-drop, with a size limit of 64 MB.

This limit is set primarily to simplify development.

Data transmission time is thus optimized (by limiting file size), avoiding the complex management of exchange progress. This also avoids managing users waiting through progress windows.

This value can, however, be modified in the `***input.js***` file located in:

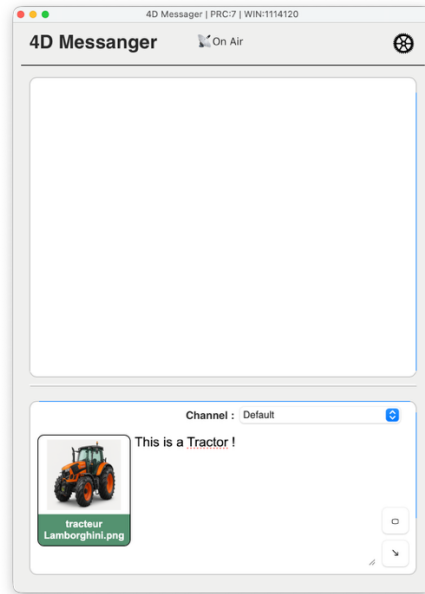
Resources / frames / input.js

The MAX_SIZE value can be modified, as shown below:

```
// mem target
const message_container = document.getElementById('message_container');
const message_cargo = document.getElementById('message_cargo');
const message_text = document.getElementById('message_text');

const MAX_SIZE = 64 * 1024 * 1024;
```

Example below: a text and an image ready to send, additional files can be dragged into the area.

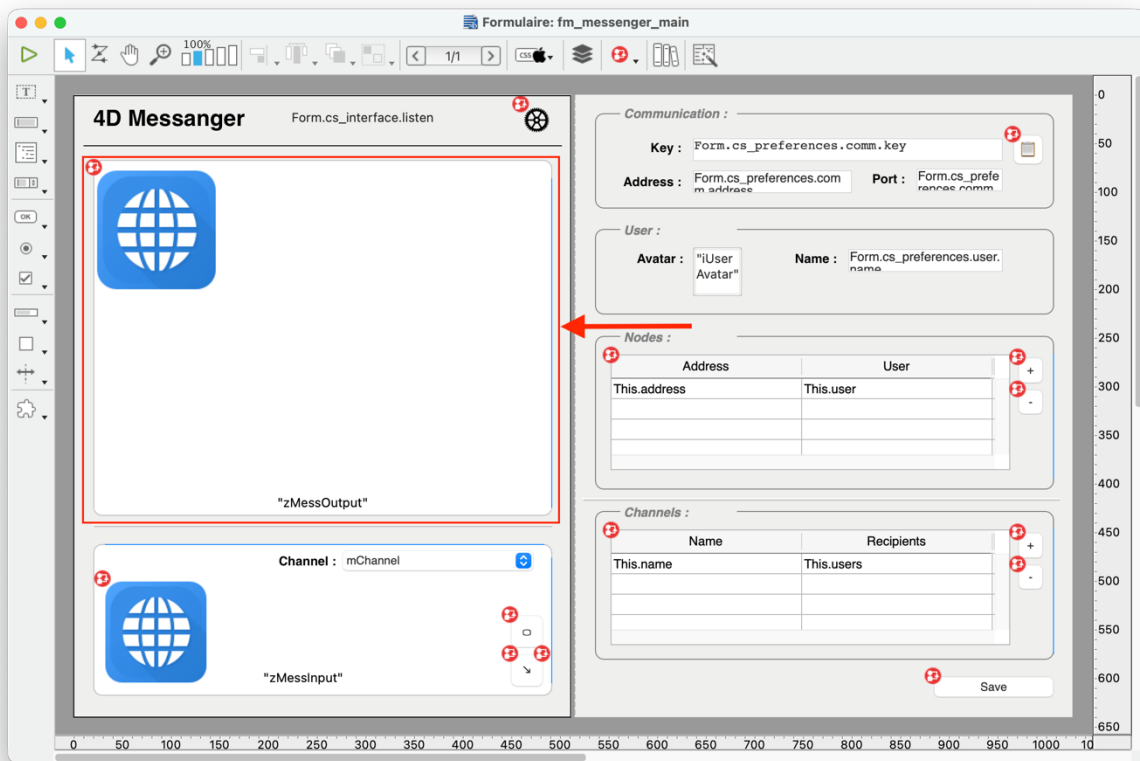


2. Message Reception

Message reception also requires being able to display text, images, videos, etc... In practice, what is displayable in input must be displayable in reception.

Additionally, the application must maintain a reception history in the display, with a clearly distinguishable separation between sent and received messages.

The same principle applies: the `zMessInput` web area serves as the display base, as shown below:



Outgoing and incoming messages are rendered in this area via HTML injection into the web page.

This is performed by invoking **WA EXECUTE JAVASCRIPT FUNCTION** to call the JavaScript function « drawMessage », defined in the following file:

Resources / frames / output.js

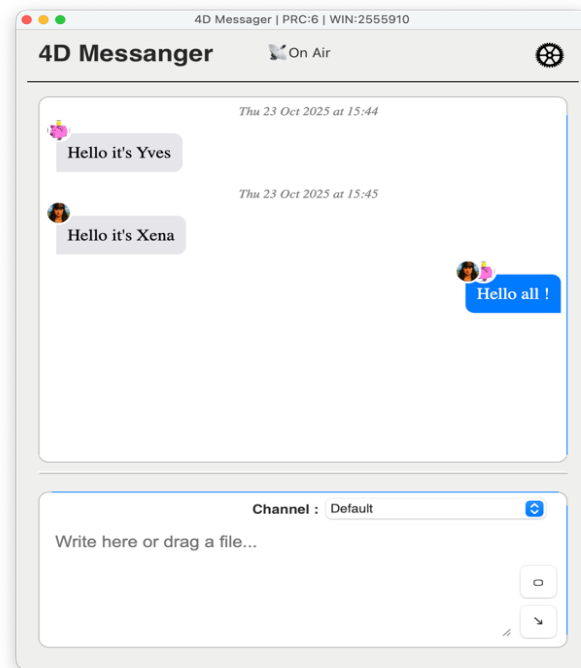
This function constructs the HTML code, with layout handled through assigned CSS classes.

It is defined in the following file:

Resources / frames / output.css

By convention, received messages are aligned to the left, while sent messages are aligned to the right.

See the example below:



3. Preferences Settings

The application must store certain parameters, such as the username and the list of potential recipients, to avoid re-entering these values at each launch.

Parameters are stored as preferences in a JSON file.

By default, the file takes the name of the machine and is located in a ****messenger**** folder within the current data directory. Example:

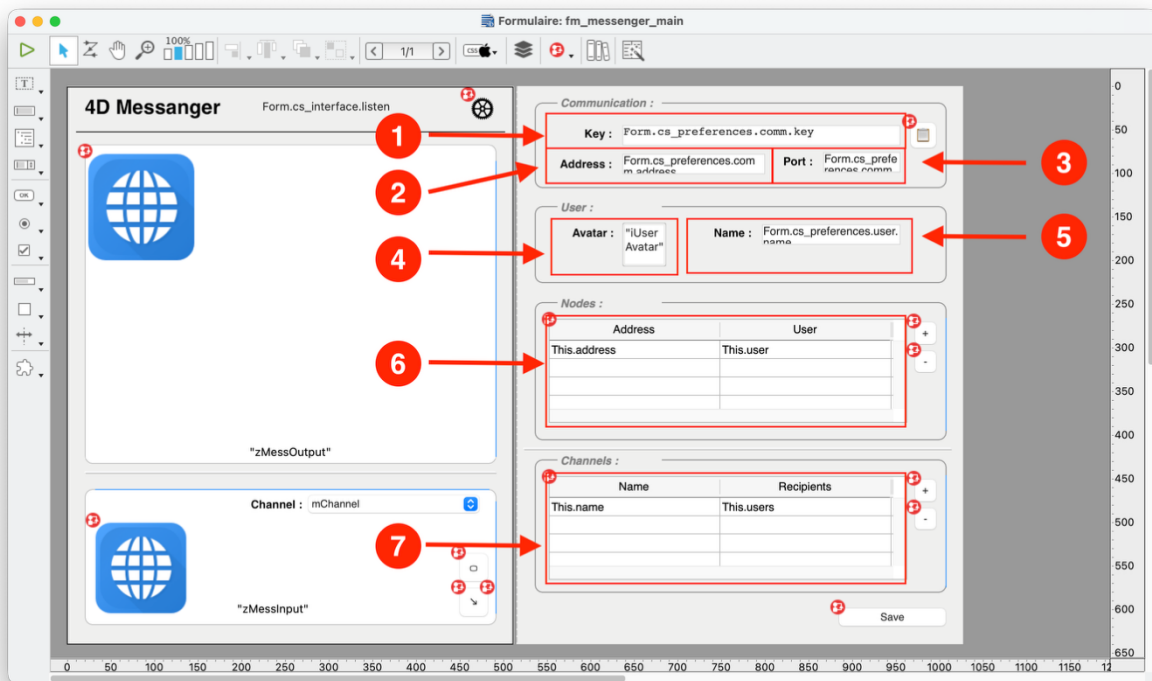
```
Data / Messenger / MyComputerName.json
```

La classe *Lib_Messenger_Preferences* class manages these preferences and is maintained via the *Form.cs_preferences*. Preferences are loaded and initialized at application startup:

```
// init prefs

Form.cs_preferences:=cs.Lib_Messenger_Preferences.new()
Form.cs_preferences.load()
```

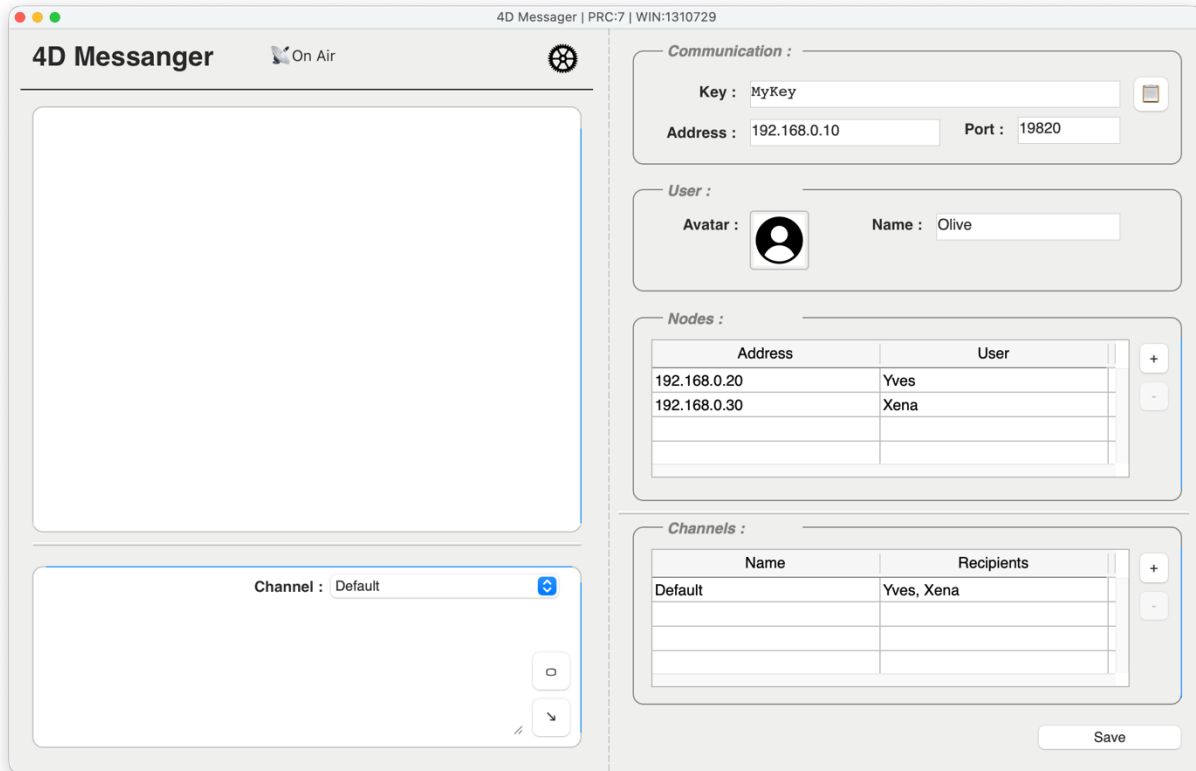
The preferences section is visible below:



These preferences include:

1. **The communication encryption key** (*comm > key*)
Responsible for encrypting data transmitted by the application.
2. **The publication address** (*comm > address*)
The address on which the application will be published, i.e., the address it listens to for incoming messages.
3. **The publication port number** (*comm > port*)
4. **The user's avatar** (*user > avatar*)
5. **The username** (*user > name*)
6. **The list of user nodes** (*comm > nodes*)
Contains the referenced users as well as the IP address of the user node.
To be able to send a message, the user must be duly referenced, the list of all possible recipients must be added before sending messages.
7. **The list of channels** (*comm > channels*)
For each channel, stores the list of recipient users, separated by commas.
8. **The preferences save button**

Configuration example:



4. Interaction with the Web Area

As mentioned, the requirement to display images, videos, text, and other content makes the web area the recipient interface display.

Additionally, using well-designed CSS classes allows the interface to be rendered efficiently and with minimal overhead.

The following CSS files dedicated to html pages, located in the web areas will therefore advantageously handle the layout:

Resources / frames / common.css

Resources / frames / input.css

Resources / frames / output.css

Almost all interactions occur through **WA EXÉCUTER FONCTION JAVASCRIPT**.

Depending on the context, a specific function is invoked to trigger an action on the web page.

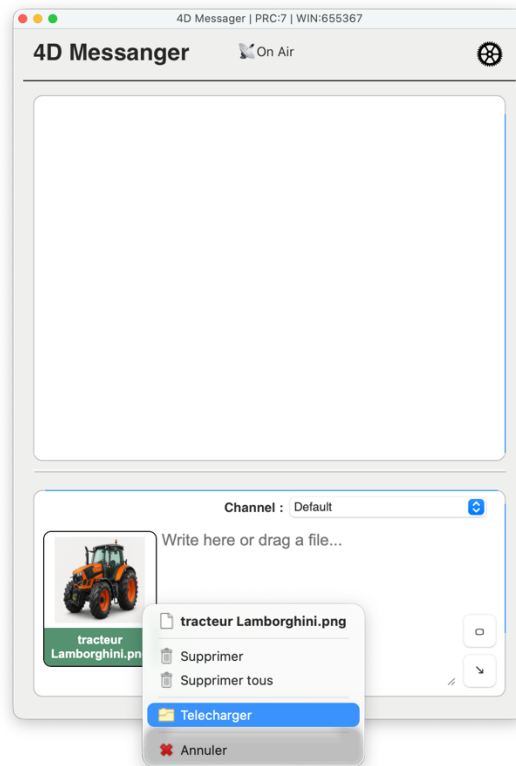
During a click for example on the « *bClear* », a call to `Form.cs_interface.actioClearHistory()` which executes:

```
WA EXECUTE JAVASCRIPT FUNCTION(*; "zMessOutput", "clearOutput"; *)
```

The JavaScript function " `clearOutput` " clears the history on the browser side.

5. 4D Menu and the Web Area

In the web area, for both message history and input, a contextual menu is useful, for example, to download an image.



The menu cannot be managed entirely in JavaScript, as it would be truncated.

The **Dynamic Pop-up Menu** command is used instead.

To interface this command and the web area, certain elements will have predefined links. Like images or files composing a message for example. These links will be intercepted **via WA SET EXTERNAL LINKS FILTERS**.

A « `Lib_Messenger_Interface` » class, maintained in the « `Form.cs_interface` » object, contains the « `callContextMenu` » function, which will choose the menu to display depending on the context.

The links contain the internal description information of the object, allowing a predefined action to be executed in context.

For example, the following url located in a message image:

```
<a data-hash="aaaf16a19f63816f5cc23db905e6ed9c891b9f4d228f57425242e85052cdf7a8"
class="ss_message_crate" href="callContextMenu?{&quot;title&quot;:&quot;tracteur
Lamborghini.png&quot;,&quot;hash&quot;:&quot;aaaf16a19f63816f5cc23db905e6ed9c891b9f
4d228f57425242e85052cdf7a8&quot;}" data-codex="3">$requestRaw : Blob  
    // declare vars  
    var $offset : Integer:=0  
    var $salt; $originSize : Integer  
    var $encrypted : Boolean  
    var $originHash : Text  
    // timestamp  
    This.date:=Current date  
    This.time:=Current time  
  
    // define way  
    INTEGER TO BLOB(This.way; $requestRaw; PC byte ordering; *)  
  
    // define sender  
    TEXT TO BLOB(This.sender; $requestRaw; UTF8 text with length; *)
```

Extract from requestUnPack :

```
Function requestUnPack($key : Text; $requestRaw : Blob)->$success : Boolean  
    // declare vars  
    var $offset : Integer:=0  
    var $salt : Integer  
    var $originSize : Integer  
    var $dateRaw : Text  
    var $originHash : Text  
    var $decrypted : Boolean  
    // primary control  
    $success:=False  
    ASSERT(BLOB size($requestRaw)>37; "4D Messenger : Request bad format")  
  
    // read length request  
    $originSize:=BLOB to longint($requestRaw; PC byte ordering)  
    ASSERT(BLOB size($requestRaw)=$originSize; "4D Messenger : Request corrupted")
```

```
DELETE FROM BLOB($requestRaw; 0; 4)
```

It also manages encryption and data quality validation.

2. Request Encryption and Decryption

The **Encrypt data BLOB**, **Decrypt data BLOB** commands manage data encryption. Although straightforward, these commands present two issues:

If the *passPhrase* is incorrect, there is no way to know that decryption has failed. The command will blindly apply the algorithm, simply decrypting anything. A hash must therefore be added to validate the decrypted data.

The padding added to secure the data is kept after decryption, making verification incorrect.

To address the issue, the structure size is included before encryption, enabling removal of padding and proper restoration of the data.

See below the extract of the resourceUnpack function:

```
SET BLOB SIZE($requestRaw; $originSize+4)  
If ($originHash=Generate digest($requestRaw; MD5 digest))
```

3. Message Acquisition

The classe *Lib_Messenger_Net_Input* class receives the message in the first instance, the latter is built around the following properties:

```
property listener : 4D.TCPLListener  
property window : Integer  
property settings : Object
```

- **listener** : maintains the 4D *TCPLListener* class
- **window** : stores the messaging window reference
- **settings** : stores the listening parameters

The important point in the "*Lib_Messenger_Net_Input*" is the function "onConnection".

```
If (This.settings.comm.nodes.query("address = :1 or address = :2"; $event.address; Replace string($event.address;  
"::ffff."; ""; 1)).length=1)  
    $result:=cs.Lib_Messenger_Stream.new(This)  
End if
```

This function verifies that the address is properly referenced in the preferences, then delegates reception to the "Lib_Messenger_Stream" class:

```
$result:=cs.Lib_Messenger_Stream.new(This)
```

The mechanism concatenates received data in a BLOB until the expected size is reached.

When reception is complete, the data is unencapsulated via "Lib_Messenger_Net_Request" and then routed according to the defined routing.

```
If ($request.way=x Messenger Get user icon)

    DOCUMENT TO BLOB(Get 4D folder(Data folder)+"Messenger"+Folder
separator+Form.cs_preferences.user.name+".png"; $binBuffer)

    INSERT IN BLOB($binBuffer; 0; 1; 0x0002)

    $connection.send($binBuffer)

Else
```

key element in the "Lib_Messenger_Stream" class is the "onData" function. This function not only acquires the data but also sends a one-byte acknowledgment of receipt, enabling the message sender to update the interface. See below:

```
$request.way:=x Messenger Receive
SET BLOB SIZE($binBuffer; 0)
INSERT IN BLOB($binBuffer; 0; 1; 0x0001)
$connection.send($binBuffer)
```

The reception codes are as follows:

Code	Description
0x1	Message received successfully
0x2	Icon request received successfully

Further within the « *OnData* » function of the « *Lib_Messenger_Stream* » class, the following block retrieves missing user icons on each node. Once cached, the icons enable interface updates:

```

$requestDraw:=cs.Lib_Messenger_Net_Request.new()
$requestDraw.sender:=Form.cs_preferences.user.name

For each ($user; Split string($request.recipient_list+","+ $request.sender; ","; sk ignore empty strings))

    $requestDraw.recipient_list:=$user

    $userIconPath:=Get 4D folder(Data folder)+"Messenger"+Folder separator+$user+".png"
    If (Test path name($userIconPath)#Is a document)

        $requestDraw.way:=x Messenger Get user icon
        $requestDraw.recipient:=$user
        $requestPack:=$requestDraw.requestPack(Form.cs_preferences.comm.key)

        $sender:=cs.Lib_Messenger_Net_Output.new(Form.cs_preferences)

        If ($user=Form.cs_preferences.user.name)
            $sender.connect(Form.cs_preferences.comm.address)
            $sender.send(OB Copy($requestDraw); $requestPack)
        Else
            If (Form.cs_preferences.comm.nodes.query("user = :1"; $user).length>0)

                $sender.connect(Form.cs_preferences.comm.nodes.query("user = :1";
$user)[0].address)

                $sender.send(OB Copy($requestDraw); $requestPack)
            End if
        End if
    End if
End for each
// update interface
CALL FORM(This.listener.window; "app_messenger_main"; False; $request)

```

4. Message sending

The Lib_Messenger_Net_Output class handles message sending. It is built around the following properties:

```
This.connection:=Null  
This.settings:=$settings  
This.requestDraw:=Create object()  
This.requestPack:=$bin
```

- **connection** : maintains the TCPConnection class
- **settings** : stores the listening parameters
- **requestDraw** : contains the message in object format (destination web area)
- **requestPack** : contains the message in binary format to send (destination TCP)

Bellow the code of the "onData" function:

```
// var  
var $bin : Blob  
// blob contain somthings  
If (BLOB size($event.data)>=1)  
    $bin:=$event.data  
    Case of  
        : ($bin{0}=0x0001)  
            // update interface  
            CALL FORM(Current form window; "app_messenger_main"; False; This.requestDraw)  
            WA EXECUTE JAVASCRIPT FUNCTION(*; "zMessInput"; "clearInput"; *)  
            GOTO OBJECT(*; "zMessInput")  
  
        : ($bin{0}=0x0002)  
            DELETE FROM BLOB($bin; 0; 1)  
            BLOB TO DOCUMENT(Get 4D folder(Data folder)+"Messenger"+Folder  
separator+Form.cs_preferences.comm.nodes.query("address = :1"; $connection.address)[0].user+".png"; $bin)  
  
    End case  
    SET BLOB SIZE($bin; 0)  
End if
```

This part handles the reception code returned by the target:

- 0x1 → message received, triggers interface update
- 0x2 → positive response to an icon request, retrieves the icon.

Conclusion

The presented application exemplifies the implementation of inter-user communication leveraging the newly introduced “CPListener” and “CPConnection” classes in 4D. By adopting a peer-to-peer architecture, each user simultaneously operates as both sender and receiver, removing dependency on a central server and thereby enhancing system flexibility, resilience, and deployment efficiency.

The dedicated classes (“Lib_Messenger_Net_Input”, “Lib_Messenger_Net_Output”, “Lib_Messenger_Net_Request”, etc.) enable structured and maintainable code, enforce secure data exchanges, and support seamless system evolution.

Overall, this technical note underscores a systematic methodology for designing sophisticated real-time messaging systems within a modern 4D environment, fully capitalizing on the capabilities and robustness of the TCP protocol.