

Dyalog™ for Windows

# Conga User Guide

**Version 2.2.0**

Dyalog Limited

Minchens Court  
Minchens Lane  
Bramley  
Hampshire  
RG26 5BH  
United Kingdom

tel: National (01256) 830030  
International +44 1256 830030  
fax: +44 (0)1256 830031  
email: [support@dyalog.com](mailto:support@dyalog.com)  
<http://www.dyalog.com>

Dyalog is a trademark of Dyalog Limited  
Copyright © 1982-2011



*Copyright © 2012 by Dyalog Limited.  
All rights reserved.*

*Version 2.2.0*

***First Edition March 2012***

*No part of this publication may be reproduced in any form by any means without the prior written permission of Dyalog Limited, Minchens Court, Minchens Lane, Bramley, Hampshire, RG26 5BH, United Kingdom.*

*Dyalog Limited makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Dyalog Limited reserves the right to revise this publication without notification.*

*All other trademarks and copyrights are acknowledged.*

# Contents

INTRODUCTION .....	1
0. RELEASE NOTES .....	2
New Functionality .....	2
Changed Functionality .....	2
Licensing Information .....	2
1. CONGA FUNDAMENTALS .....	3
1.1 A Simple Conga Client .....	3
1.2. A Simple Server .....	4
1.3 Command Mode .....	6
1.4 Parallel Commands .....	8
1.5 More on Multi-Threading .....	8
1.6 Conga versus TCPSocket objects .....	9
1.7 Deflate and HTTP Compression .....	10
2. SECURE SOCKETS .....	12
2.1 CA Certificates .....	12
2.2 Client and Server Certificates .....	13
2.3 Creating a Secure Client .....	14
2.3 Creating a Secure Server .....	16
2.4 Using the <code>DRC.X509Cert</code> Class .....	16
3. SAMPLES .....	18
3.1 Overview .....	18
3.2 The Samples Namespace .....	19
3.3 Web Client .....	19
3.4 Web Server .....	21
3.5 RPC Client and Server .....	22
3.6 FTP Client .....	23
3.7 Telnet Server .....	24
3.8 Telnet Client .....	25
3.9 TODServer .....	25
APPENDIX A: FUNCTION REFERENCE .....	26
Checking DRC Return Codes .....	26
APPENDIX B: CREATING AND CONVERTING CERTIFICATES .....	43
APPENDIX C: TLS FLAGS .....	45
APPENDIX D: ERROR CODES .....	46
APPENDIX E: UPGRADING TO CONGA V2.2 .....	47
APPENDIX F: CHANGE HISTORY .....	48
v2.0 New Functionality .....	48
v2.0 Changed Functionality .....	48
New Functionality .....	49
Changed Functionality .....	49



# Introduction

Conga<sup>1</sup>, also known as the Dyalog Remote Communicator, is a tool for communication between applications. Conga can be used to transmit APL arrays between two Dyalog applications which are both using Conga, and it can be used to exchange messages with many other applications, like HTTP servers (also known as “web servers”), web browsers, or any other web clients and servers including Telnet, SMTP, POP3 and so forth.

Uses of Conga include, but are not limited to the following:

- Retrieving information from – or “posting” data to – the internet.
- Accessing internet-based services like FTP, SMTP, or Telnet
- Writing an APL application that acts as a Web (HTTP) Server, mail server or any other kind of service available over an intra- or the internet.
- APL “Remote Procedure Call” (RPC) servers which receive APL arrays from client applications, process data, and return APL arrays as the result.

From version 12.0 of Dyalog, Conga replaces the use of `TCPsocket` objects as the recommended mechanism for managing TCP-based communications from Dyalog. (Although Conga currently only uses the TCP protocol, the interface to Conga is at a level where other communication mechanisms could be added in the future.) The programming model for Conga is significantly simpler, and supports multi-threaded applications more easily than do `TCPsocket` objects. Conga also supports secure communication using TLS (Transport Layer Security), which is the successor to SSL (Secure Sockets Layer).

Conga is implemented as a Windows Dynamic Link Library or a Unix/Linux Shared Library. The library is loaded and accessed through the companion namespace named `DRC`, found in the distributed workspace named `Conga`. The `Conga` workspace also contains a number of sample applications, which illustrate its use, and are discussed in this document.

---

<sup>1</sup> Named after one of the earliest known forms of long distance messaging, the Conga drum.

# 0. Release Notes

## Version 2.2

Conga version 2.2 is an incremental update to Conga version 2.1. While released as a companion to Dyalog APL version 13.1, Conga v2.2 can be used with earlier versions of Dyalog APL as well.

### New Functionality

- **Deflate Compression:** Conga v2.2 includes a new class, `#.DRC.flate`, which implements the Deflate compression scheme using the zlib open source compression library. For more information on zlib, please refer to <http://zlib.net>. Deflate compression is one of several content encoding schemes used by all major web servers and browsers to optimize the flow of data across networks.
- **DRC.Version function:** Returns the current version of Conga. This is useful if you write code that dynamically loads Conga.

### Changed Functionality

- **DRC.Send function:** Allows a new value, 2, for the close connection parameter. This is used to send a command without expecting a response. On the client side, the command is disposed of after sending. On the server side, the command is disposed of after receipt, thereby preventing the server from doing a subsequent **DRC.Respond**.
- **DRC.Version function:** The GnuTLS version is reported.
- **Samples.HTTPGet function:** Now supports deflate HTTP compression.

### Licensing Information

- If you redistribute code that uses Conga, please refer to the "Licenses for third-party components" document found in the help directory of the Dyalog APL installation.

# 1. Conga fundamentals

This chapter introduces Conga Client and Server objects, and demonstrates their use through simple examples.

## 1.1 A Simple Conga Client

A Conga client is used to establish contact with a service which is already running and “listening” on a pre-determined “port” at a known TCP “address”. The service might be an APL application which has created a Conga server, but it can also be any application or service which provides services through TCP sockets. For example, most Unix systems, and many Windows servers, provide a set of simple services like the Time of Day (TOD) service, or the Quote of the Day (QOTD) service, both of which respond with a text message as soon as a connection is made to them. Once the message has been sent, they immediately close the connection.

The interface to Conga is provided in a namespace called DRC (for Dyalog Remote Communicator<sup>2</sup>). Before using any DRC functions, we need to initialize the system by loading the Windows DLL or Unix Shared Library. To do this, we need to load the Conga workspace or copy the DRC namespace from it, and call `DRC.Init`:

```
)COPY Conga DRC
...Conga saved... some time on some day...
DRC.Init ''
0 Conga loaded from: ...\\bin\\Conga20Uni
```

The function `DRC.Clt` is used to create a Conga client. In the following example, we provide an argument with five elements, which are:

- the name that we want to use to refer to the client object (`C1`)
- the IP address or name of the server machine providing the service (`localhost`),
- the port on which the service is listening (`13` – the “Time of Day” service),
- the type of socket (`Text`), and finally
- the size of the buffer which should be created to receive data (`1000`).

```
DRC.Clt 'C1' 'localhost' 13 'Text' 1000
1111 ERR_CONNECT_DATA /* Could not connect ... */
```

In the event of an error, the first element of the result of all DRC functions is a return code, the second is an error name, and in some cases the third element contains more information about the error. You should not assume a fixed length for the result; additional information may be included in future versions.

The above is the most likely result if you try the example under Windows; there is not usually a TOD service running. Under some versions of Windows, you can go to Control Panel|Programs|Turn Windows features on or off, and enable Simple TCP/IP services.

---

<sup>2</sup> Or the Democratic Republic of Congo, the home of many Congas.

If you got the 1111 error, you can either enable the services on your machine, write your own (see the next section), or use the address of a host which does provide a TOD service – for example:

```
DRC.Clt 'C1' 'myLinuxBox' 13 'Text' 0
0
```

The result code of zero indicates that the client was successfully created. To receive incoming data, call `DRC.Wait` with the name of the object on which to wait:

```
DISP DRC.Wait 'C1'
```

0	C1	Block	09:58:40 13-02-2008(crlf)
---	----	-------	---------------------------

The elements of the result are the return code (0), the name of the object (C1), the type of event (Block), and data associated with the event.

Finally, we should close the client object:

```
DRC.Close 'C1'
0
```

The above illustrates the simplest possible use of a Conga client (for a further example, see the function `Samples.TestSimpleServices` in the Conga workspace). Most uses of a client would also require the use of the function `DRC.Send` to transmit data to the service before receiving a result – and an understanding of a few more possible return codes and event types from `DRC.Wait`. We'll take a look at a few examples of this later on, but first we'll take a look at the simplest imaginable Conga server:

## 1.2. A Simple Server

The *Time of Day* service used in the previous example is a very simple server, and can be implemented using a handful of calls to Conga to create a server object. The following function is provided under the name `TODServer.Run` in the Conga workspace:



```

▽ Run port;wait;data;event;obj;rc;r
[1] A Time of Day Server Example (use port 13 by default)
[2]
[3] ##.DRC.Init '' ◇ DONE←0 A DONE is used to stop service
[4] :If 0≠1⇒r←##.DRC.Srv 'TOD' '' port 'Text'
[5]   □←'Unable to start TOD server: ',⌘r
[6] :Else
[7]   □←'TOD Server started on port ',⌘port
[8]   :While ~DONE
[9]     rc obj event data←4↑wait←##.DRC.Wait 'TOD' 1000
                                A Time out every second
[10]     :Select rc
[11]     :Case 0
[12]       :Select event
[13]       :Case 'Connect'
[14]         r←(, 'ZI2,<:>,ZI2,<:>,ZI2,< >,ZI2,<->,ZI2,<->,ZI4'
                □FMT 1 6ρ□TS[4 5 6 3 2 1]),□AV[4 3]
                {}##.DRC.Send obj r 1 A 1=Close connection
[15]         {}##.DRC.Close obj A Anything unexpected
[16]       :Else
[17]         {}##.DRC.Close obj A Anything unexpected
[18]       :EndSelect
[19]     :Case 100 A Time out - Housekeeping Here
[20]     :Else
[21]       □←'Error in Wait: ',⌘wait ◇ DONE←1
[22]     :EndSelect
[23]   :EndWhile
[24]   {}##.DRC.Close 'TOD' ◇ □←'TOD Server terminated.'
[25] :EndIf

```

This function enters a loop where it waits for connections. Therefore, if we want to be able experiment with using this service without starting a second APL session, we start it in using the spawn operator (&) so that it runs in a separate thread:

```

TODServer.Run&13
TOD Server started on port 13

```

The right argument is the port number: If your machine is already running a TOD service on port 13, you will probably get socket error number 10048, and you will need to use a different port for the new service. The following examples assume that port 13 was available:

```

DRC.Clt 'C1' 'localhost' 13 'Text'
0
DRC.Wait 'C1'
0 C1 Block 14:36:23 06-05-2007
DRC.Close 'C1'
0

```

Note that the above service is a completely normal TOD service, in the sense that it could be used by any program which is written to use a TOD service – not only Dyalog applications using Conga. We can stop the server as follows (it may take a second for the Server to time out and discover that it has been asked to shut down):

```

TODServer.DONE←1
TOD Server terminated.

```

The function Run works as follows:

- [3] Call DRC.Init and set global flag DONE to zero.
- [4] Create a Server object named TOD on selected port in Text mode.

- [ 8 ] Repeat the following until `DONE` is equal to 1:
- [ 9 ] Wait for any event and split the result into `rc` (return code), `obj` (object name), `event` and `data`. `obj` will be a string identifying a “child object” of `TOD`, with a name like `'TOD.CON00000000'`.
- [14] If return code was 0 and the event was `Connect`, format the time of day
- [15] Send the time of day to `obj`. The 1 in the 3rd element of the argument to `Send` instructs `Conga` to close the object as soon as the data has been sent.
- [17] For any other event, we simply close the connection. (We are a very simple service.)
- [19] If you want the service to periodically do housekeeping tasks, we will arrive here every 1000 milliseconds (specified in the argument to `Wait` on line 9).
- [21] Any return code from `Wait` other than 0 or 100 will cause a shutdown of the service.
- [24] When we are done, close the server object

### 1.3 Command Mode

As we have seen in the preceding sections, we can use `Conga` as a client to connect to an existing server and make requests, or as a server to wait for connection from clients and provide a service.

In the above examples, we used `Text` connections, which are appropriate for most web applications. Even when remote procedure calls are made over the internet, with arguments and results containing arguments which are not simply text strings, the parameters are usually encoded using `SOAP/XML`, which is a text-based encoding.

`Conga` clients and servers support three different connection types:

- Text** Allows transmission of character strings, which *must* consist of characters with Unicode code points less than 256. To transmit characters outside this range, it is recommended that you UTF-8 encode the data (see `UCS`).
- Raw** Essentially the same as a `Text` connection, except that data is represented as integers in the range 0 to 255 (for coding simplicity, negative integers `-128` to `-1` are also accepted and mapped to `128-255`).
- Command** Each transmission consists of a complete APL object in a binary format.

`Text` and `Raw` connections are essentially equivalent, and are typically used when only one end of the connection is an APL application.

`Command` connections are designed to make it easy for APL clients and servers to communicate with each other. The internal representation is the binary format used by APL itself, it is more compact than a textual representation, and numbers do not need to be formatted and interpreted in order to be transmitted. No buffer size needs to be declared, and `DRC.Wait` only reports incoming data when an entire APL array has arrived. For connections between APL clients and servers, `Command` mode is therefore more convenient.

When using `Text` and `Raw` connections, `Wait` will report incoming data each time a TCP packet arrives, or the receive buffer is full. The recipient may need to buffer

incoming data in the workspace and analyze it to determine whether a complete message has arrived.

We could produce a Command Mode Time-of-Day server for use by APL clients only, which returns the time as a 7-element array in `⌈TS` format.

To do this, we need to make the following changes to `TODServer.Run`: Remove `'Text'` from the end of line `Run[4]` (Command mode is the default). Lines [13-15] can be replaced by the following:

```
[13] :Case 'Connect' A Ignore
[14] :Case 'Receive'
[15]      {}##.DRC.Respond obj ⌈TS
```

In Command mode, all communication on a connection is synchronous and broken up into “commands”, each consisting of a request from the client followed by a response from the server. Unlike the text mode TOD service, a server in Command mode cannot initiate the transmission of data when the connection is made, but has to wait for the client to send a request to which it can respond. If our TOD server wanted to record connections, it could use the `Connect` case statement for this, but we will ignore this for now and simply respond with the current timestamp regardless of the content of the request.

Note that, in Command mode, the function `DRC.Respond` is used in place of `DRC.Send`. A function called `DRC.Progress` can be used to send progress messages while the server is processing a command, to allow the client to show the user a progress bar or other status information.

We can now start the modified server – ideally on some other port than 13, so that it is not confused with a “normal” TOD server. We could run both at the same time, in different threads, if we so desire:

```
TODServer.Run&913
TOD Server started on port 913
```

A Dyalog client can now retrieve a numeric timestamp from the server, as follows:

```
DRC.Clt 'C1' 'localhost' 913
0 C1
DRC.Send 'C1' ''
0 C1.Auto00000000
```

The first element of the argument to `Send` is a command name. If the name of the connection is used instead, Conga will generate a command name automatically, in this case `C1.Auto00000000`. The command name is always returned in the second element of the result.

Dyalog client can now retrieve a numeric timestamp from the server, as follows:

```
DRC.Wait 'C1'
0 C1.Auto00000000 Receive 2008 2 13 10 41 39 585
```

Element 4 of the result is now a 7-element integer vector rather than a formatted timestamp, which is more useful to an APL client. However, the server is now unusable by other TCP client programs, if they are expecting a Text mode TOD server. For this reason, it would be unwise to run the command mode service as a listener on port 13.

Note that the Command mode server also does not close the connection after sending a timestamp, so we can ask for the time of day again if we like (this time we will supply a maximum wait time of 5 seconds to `DRC.Wait`):

```

    DRC.Send 'C1' ''
0 C1.Auto00000001
    DRC.Wait 'C1' 5000
0 C1.Auto00000001 Receive 2008 2 14 21 20 8 169

```

## 1.4 Parallel Commands

Although the command mode protocol is synchronous, you can have more than one command active at the same time, and it is not necessary to wait for the response to one command before the next is sent. You can start multiple commands and retrieve the results in any order that you like. In the above examples, the command name was automatically generated, but you can also specify command names if you prefer:

```

    DRC.Send 'C1.TS1' ''
0 C1.TS1
    DRC.Send 'C1.TS2' ''
0 C1.TS2
    DRC.Wait 'C1.TS2' 1000
0 C1.TS2 Receive 2008 2 14 21 52 17 48
    DRC.Wait 'C1.TS1' 1000
0 C1.TS1 Receive 2008 2 14 21 52 14 873

```

Note that the timestamp shows that the `TS1` command was executed first, even though the result was retrieved last.

The command mode protocol allows multiple threads to work independently. Unlike `TCPsocket` objects, which can only be “dequeued” by the thread which created them, *any* thread can wait for the result of a command, so long as it knows the name (for predictable results, only one thread should wait for each command). Multiple threads can share the same sever connection, so a thread can send a command and then dispatch a new thread to wait for and process the result of a command, while the main thread continues with other work. For example:

```

    DRC.Send 'C1.TS1' ''
0 C1.TS1
    DRC.Send 'C1.TS2' ''
0 C1.TS2
    {[]TID,DRC.Wait ω 1000}&'' 'C1.TS1' 'C1.TS2'
29 0 C1.TS1 Receive 2008 2 14 21 55 39 465
30 0 C1.TS2 Receive 2008 2 14 21 55 39 553

```

The above expression asynchronously runs a dynamic function – each in a separate thread - for each command. Each function call returns the thread number and the result of `Wait`. Calls to `Wait` are “thread switching points”, which means that APL will suspend a waiting thread, and allow other threads continue working. Also note that command names can be reused as soon as the result has been received – but not before.

## 1.5 More on Multi-Threading

Conga is specifically designed to support multi-threaded applications. In particular, the ability to have a program work as both client and server simultaneously, without blocking other threads, has been a key design goal. All calls to Conga are implemented as asynchronous calls to an external Windows DLL or Unix/Linux Shared Library. Threads waiting on a result from Conga are suspended, but all other threads can continue execution.

For example, the `RPCServer` namespace contains an example of a server working in Command mode. This server is able to execute APL statements in the server workspace and return results to client applications. The function `Samples.TestRPCServer` starts the RPC server and then exercises it by making a number of calls. Each client call is made in a separate thread. On the server side, the function `RPCServer.Process` is dispatched in a new thread to handle each request. (Keep an eye on the thread count at the bottom of the session as you run this function.)

If this server needed to know the time, we could safely add a call to a Time-Of-Day service accessed through Conga to the function which processes client requests, simply by adding a couple of lines to the beginning of the function `RPCServer.Process`:

```
[2.1] tod←2>##.DRC.Clt ' ' 'localhost' 13 'Text' 1000
[2.2] time←4>##.DRC.Wait tod 1000 ⋄ ##.DRC.Close tod
```

(Adding error checking and localization of `tod` and `time` is left as an exercise for the reader ☺.)

The TOD service could be external, but it could also safely run in the same workspace as everything else – so long as it was launched as a separate thread as shown in section 1.2. Outside the APL session, Conga (which is written in C) uses multiple operating system threads to handle TCP communications. It will handle communications independently of what the interpreter is doing, and return each result to the APL thread which is waiting, as appropriate.

The application developer only needs to take care that there is an APL thread waiting on each server object that has been created. (Otherwise requests will not be serviced.) Having more than one thread waiting on the same object is not recommended – it can lead to unpredictable behaviour.

**Tip** If you experiment with adding the above functionality and everything seems to lock up, try using the `Threads|Resume all Threads` menu item. By default, all threads are paused on error and resuming execution of a suspended function does not restart other threads by default.

## 1.6 Conga versus TCPsocket objects

Experienced Dyalog users will recognize that the functionality provided by Conga is similar to that provided by the Dyalog `TCPsocket` component, and wonder why Dyalog is introducing a second mechanism to address essentially the same requirements.

The `TCPsocket` object is implemented as a GUI object and closely models the underlying TCP socket which it is covering. Although this approach is very flexible, experience has shown that most applications fall into a small handful of usage patterns, and that many APL programmers struggle to manage correctly all the issues related to initialising sockets, handling errors, and – last but definitely not easiest – closing sockets. In addition, because events on TCP socket objects are “received” using the system function `⌈DQ`, which is also used to handle GUI events, `TCPsocket` objects are often a little tricky to use to implement remote-calling mechanisms that will be used inside – or in parallel with – callback functions in a GUI application. Multi-threaded and multi-tier applications can be quite difficult to implement using this model.

Conga is designed to make it easy for APL developers to embed client or server components in APL applications. Conga hides many of the details of TCP socket

handling, notifies the application of incoming data, connection events and errors – but the application does not need to do anything other than handle the data which arrives. Conga makes it straightforward to make remote calls in a multi-threaded client environment.

Finally, because Conga hides most of the details of TCP sockets, it can be ported to work on top of other communications mechanisms at some point in the future.

## 1.7 Deflate and HTTP Compression

The `DRC.flate` namespace contains functions to provide support for "deflate" data compression. Deflate is one of several content encoding schemes that can be used to implement HTTP compression. All major web browsers and web servers support deflate.

The first step to using deflate is to make sure the deflate compression library is loaded.

```
DRC.flate.IsAvailable
1
```

While deflate can be used as a general data compression utility, we will focus on its use to provide HTTP compression. HTTP compression is a technique to improve throughput between HTTP clients and servers by transmitting less data across the network. Typically a client will be a web browser like Microsoft Internet Explorer, Google Chrome, Mozilla Firefox, and Apple Safari and a server will be a web server like Microsoft IIS, Apache, or IBM WebSphere. `DRC.flate` enables Conga-based clients and servers to implement and support HTTP compression.

### How HTTP Compression Works

In order for HTTP compression to work, both the client and server need to support the same content encoding scheme. The client informs the server what compression schemes it supports via the `Accept-Encoding` HTTP header that is sent with the request. `Accept-Encoding` is a comma-delimited list of compression schemes that the client can process. Gzip and deflate are, at present, the two predominant compression schemes in use.

The server examines at the `Accept-Encoding` header received from the client and if it supports one or more of the listed schemes, it *may* choose to encode the body of its response using one of the schemes. The server is not required to use the content encoding scheme, but if it does, it informs the client of the content encoding scheme it used via the `Content-Encoding` HTTP header.

Finally, when the client receives the response, it checks the `Content-Encoding` HTTP header and, if found, the body of the response using that scheme.

The open source library, zlib, used to implement deflate compression prepends a 2-byte header, normally `120 156`, to the beginning of the compressed data. Most web browsers handle this without difficulty, however Microsoft's Internet Explorer does not properly process these bytes, and as such, web servers tend to strip them off. This requires some consideration for both client and server sides.

One choice you will need to make is whether to use `'text'` or `'raw'` mode with clients and servers that will use deflate. `'text'` mode is more convenient for the parsing and processing of the HTTP message wrapper, whereas `'raw'` mode is better suited for passing data to `DRC.flate.Deflate` and `DRC.flate.Inflate`.

### Client Side Considerations

If you plan to implement a client that uses deflate compression:

- Add the HTTP header:  
`Accept-Encoding: deflate`  
to the HTTP headers sent with the client request.
- `DRC.flate.Inflate` takes as its right argument an integer vector of values in the range 0-255. As such, it is best suited to be used with `DRC.Clt 'raw'` mode. However, if you use 'text' mode, you should use  
`'UTF-8' DR ω: 256|83 DR data`  
to convert the data to a form suitable for `DRC.flate.Inflate`.
- Check the `Content-Encoding` HTTP header in the response, and if it contains 'deflate', prepend `120 156` to the response before calling `DRC.flate.Inflate`
- See the function `Samples.HTTPGet` for an example of the use of client-side HTTP compression.

### Server Side Considerations

If you need to implement a server which supports deflate compression:

- The first choice is to decide when to use HTTP compression. For small responses, the CPU overhead to perform the compression may outweigh the gains of transmitting less data. Also, there is no benefit to compressing data which is already in a compressed format. These would include compressed files such as .zip and .gz files and many graphics formats such as .jpg/.jpeg and .gif.
- When a request from a client is received, check for the existence of the `Content-Encoding` HTTP header and, if found, that it contains 'deflate'.
- When you do use deflate to compress your response, add the HTTP header:  
`Accept-Encoding: deflate`  
to the HTTP headers sent with the server response.
- `DRC.flate.Deflate` takes as its right argument an integer vector of values in the range 0-255. However, responses, such as web pages, are best assembled as text. As such, you'll probably need to perform some conversion. The following is a slightly modified sample of code from `MiServer`, an APL-based web server. Note how the 2-byte header is dropped from the result.

```

▽ (rc raw)←Compress buf;toutf8
:Implements Method ContentEncoder.Compress
toutf8←(3=10|DR ω: 256|ω ◊ 'UTF-8' DR ω)
:Trap 0
  ⚡ drop of 789C header (IE cannot process it)
  raw←((2×120 156≡2↑ω)↓ω)#.DRC.flate.Deflate toutf8 buf
  rc←0
:Else
  (rc raw)←1 DR ω
:EndTrap
▽

```

## 2. Secure Sockets

Conga supports secure connections using SSL/TLS protocols. If you do not intend to use secure communications, you can safely skip to the next chapter which discusses the samples which are included with Conga. Secure connections allow client and server applications to:

1. Verify the identity of the partner that they are connected to.
2. Encrypt messages so that the contents cannot be deciphered by a third party, even when using text or raw mode connections.
3. Ensure that messages have not been tampered with by a third party, during transmission.

SSL/TLS is a generic term for a set of related protocols used to add confidentiality and authentication to communications channels such as sockets. TLS, which stands for “Transport Layer Security” is the successor to SSL, the “Secure Socket Layer” protocol V3 designed by Netscape. TLS is defined by the IETF and described in RFC 2246. There are only minor differences between the two protocols, so their names are often used interchangeably. A good discussion of SSL/TLS may be found at [http://technet.microsoft.com/en-us/library/cc784450\(WS.10\).aspx](http://technet.microsoft.com/en-us/library/cc784450(WS.10).aspx)

A good overview of the public key cryptography techniques used in SSL/TLS can be found at:

[http://developer.mozilla.org/en/docs/Introduction\\_to\\_Public-Key\\_Cryptography](http://developer.mozilla.org/en/docs/Introduction_to_Public-Key_Cryptography)

The sections on the SSL protocol, and CA (certificate authority) certificates are recommended reading for anyone who would like to make use of secure communications. The page <http://en.wikipedia.org/wiki/X.509> also contains an introduction to how X.509 certificates and how CAs (Certificate Authorities) are used to establish trust.

To use TLS/SSL, Conga simply needs to be passed the necessary Certificate and Public Key files, when Client and Server objects are created. Once a secure connection is established, the same functions are used to send and receive data – and with the same arguments – as when using a non-secure connection.

### 2.1 CA Certificates

CAs (Certificate Authorities) are trusted third parties that sign certificates to indicate that a certificate belongs to whom it claims to belong to. Assuming that you trust the CA that signed a certificate that some third party presents to you, and the CA certificate is still valid, you can use the certificate to verify the identity of your communications partner, or “Peer”. To check the CAs signature on a certificate, you need to have access to the CAs public certificate (often called a root certificate).

Conga can be used to secure many different types of system, which may require different (and sometimes private) root certificates. Therefore, you may need to obtain the root certificates from several CAs. All public root certificates that you wish to use with Conga need to be placed in a root certificate directory. Conga needs to be



informed about the location of the root certificates with a call to the function `SetProp`. For example, you should be able to use the sample root certificates by typing:

```
DRC.SetProp '.' 'RootCertDir' (Samples.CertPath, 'ca')
```

`Samples.CertPath` is a function which returns the location of the *TestCertificates* folder (if it can find it).

You may be lucky enough to have a system administrator who provides you with all the necessary certificates, but in case you do not, fairly recent copies of the most common certificates are shipped with Conga, and you can download the latest certificates from the CAs websites.

The following table lists the download pages for root certificates for the most widely used CAs, and whether their main root certificates are shipped with Conga (in which case you can find them in the folder *PublicCACerts* below the main Dyalog program folder). Please note that *PublicCACerts* contains only the CA main root certificates. Most CAs have additional certificates available for download, some that may be application specific.

Authority	Included	Download root certificates from
VeriSign, Geotrust & Thawte	✓	<a href="http://www.verisign.com/support/roots.html">http://www.verisign.com/support/roots.html</a>
Comodo	✓	<a href="http://www.comodo.com/repository/">http://www.comodo.com/repository/</a>
GoDaddy & ValiCert	✓	<a href="https://certs.godaddy.com/Repository.go">https://certs.godaddy.com/Repository.go</a>
Cybertrust	✓	<a href="http://cybertrust.omniroot.com/support/sureserver/rootcert_ap.cfm">http://cybertrust.omniroot.com/support/sureserver/rootcert_ap.cfm</a>
Entrust	✓	<a href="http://www.entrust.net/developer/index.cfm">http://www.entrust.net/developer/index.cfm</a>
CAcert	✓	<a href="http://www.cacert.org/index.php?id=3">http://www.cacert.org/index.php?id=3</a>
GlobalSign	✓	<a href="https://www.globalsign.com/support/root-certificate/osroot.htm">https://www.globalsign.com/support/root-certificate/osroot.htm</a>
IPS Servidores	×	<a href="http://www.ips.es/Declaraciones/NuevasCAS/NuevasCAS.html">http://www.ips.es/Declaraciones/NuevasCAS/NuevasCAS.html</a>

Note that some organizations use root certificates generated within the company, in which case you may be using root certificates generated by your own system administrators rather than one of the above authorities.

Conga accepts certificates in files with one of the extensions `.cer`, `.pem` or `.der` files. These files must contain data in either PEM or DER format. See Appendix B for instructions on how to create certificate files.

## 2.2 Client and Server Certificates

These certificates are files used to identify the machines at each end of a secure connection, so that a peer can decide whether or not they are who they claim to be. Conga uses X.509 certificates to establish the identity of the peer in a TLS/SSL connection. A X.509 certificate contains the public portions of a certificate, including details of the public key algorithm and signing certificates signature to validate the contents of the certificate

The Dyalog installation includes a set of test certificates which can be used to test SSL support, and are used by the functions with names beginning with `TestSecure` in the `Samples` namespace. The test certificates are found in the folder `TestCertificates`, which has 3 subfolders called `ca`, `Server` and `Client`. These certificates can be used for testing your own code, but should *never* be used in production code.

`TestCertificates/ca/ca-key.pem`: The private key for the test CA, which was used to sign the client/server & CA certificates. As this is distributed with Conga no certificate that relies on this can be considered truly secure.

`TestCertificates/ca/ca-cert.pem`: The public certificate for the test CA. Used to authenticate the client/server certificates.

`TestCertificates/ca/DyalogCaPublic.pem`: The public certificate for the test CA for <https://ssltest.dyalog.com/> (which is used in the function `TestSecureWebClient`). Note that this is again a self-signed certificate, but using a different CA key to the one for `ca-cert.pem`.

`TestCertificates/client/client-cert.pem` and `client-key.pem`: The certificate/key pair used for sample clients.

`TestCertificates/server/server-cert.pem` and `server-key.pem`: The certificate/key pair used for sample servers.

### Certificate Stores

Certificates can also be stored in common repository known as a certificate store. Conga v2.1 supports the ability to read certificates from the MicroSoft certificate store under Windows.

### Revocation Lists

Conga does not currently support the use of Certificate Revocation Lists. However this may be added in future versions if required.

## 2.3 Creating a Secure Client

Secure Conga Clients are created by passing certificate and key information to the `Clt` function. Beginning with Conga v2.1, the recommended way to do this is to use the `DRC.X509Cert`.

```
cert<->DRC.X509Cert.ReadCertFromFile
    path,'client/client-cert.pem'
cert.KeyOrigin<'DER' (path,'client/client-key.pem)
certs<('X509' cert)('SSLValidation' 16)
DRC.Clt 'C1' 'localhost' 713 'Text', certs
```

The first line uses the `X509Cert` class method `ReadCertFromFile` to read the certificate and create an instance that contains all of the information for the certificate. The second line specifies the location of the file containing the private key.

Another technique, which is similar to that used in Conga v2.0, is to specify the locations of the certificate and key files.

```

certs←('PublicCertFile' ('DER' (path,'client/client-
                             cert.pem')))
certs,←('PrivateKeyFile' ('DER' (path,'client/client-
                             key.pem')))
certs,←('SSLValidation' 16)
DRC.Clt 'C1' 'localhost' 713 'Text', certs

```

`PublicCertFile` and `PrivateKeyFile` identify the files containing the public certificate and private key files.

In fact, a certificate is not required in order to create a secure client; many secure servers accept connections from clients without certificates. In this case, the server cannot verify the identity of the client, but the connection is still encrypted and safe from tampering. Most web commerce sites use this type of connection to protect sensitive used data transmitted over the internet without requiring that customers have a digital signature. To create a secure connection with no certificate, you may create an empty certificate as in (`'X509' (NEW DRC.X509Cert)`).

`SSLValidation` contains the sum of TLS flags (see Appendix C for a complete list). A typical flag value used for a client connection would be 16 (accept the server certificate even if its hostname does not match the one we asked to connect to), or 32 (accept without validating). The latter can be useful to determine the reason why a connection is failing. For example, if we try to connect to a secure site, and have not set `RootCertDir` to point to the required CA certificates, all attempts to make secure connections will fail:

```

args←' ' 'ssltest.dyalog.com' 443 'Text' 100000
args,←('X509' (NEW DRC.X509Cert ' '))

DRC.SetProp '.' 'RootCertDir' 'c:\wrong_folder'
DRC.Clt args
1202 ERR_INVALID_PEER_CERTIFICATE
/* The peers certificate is not valid */

```

Without access to the necessary CA certificate, validation fails. But we can connect if we disable validation:

```

DRC.Clt args, ←'SSLValidation' 32
0 CLT00000051

```

Having connected without validation, we can retrieve the certificate information and use this to decide whether we wish to proceed with the conversation with this server (output above has been adjusted to increase readability):

```

rc cert←DRC.GetProp 'CLT00000051' 'PeerCert'
,[1.5]1>cert.Formatted.(ValidFrom ValidTo Issuer Subject)
Mon Dec 17 15:56:44 2007
Sat Dec 15 15:56:44 2012
C=UK,ST=Hampshire,L=Bramley,O=Dyalog ltd.,CN=Dyalog Ltd. Test
Root CA,EMAIL=jonathan@dyalog.com
C=UK,ST=Hampshire,L=Bramley,O=Dyalog ltd.,CN=Dyalog Ltd. Test
Root CA,EMAIL=jonathan@dyalog.com

```

We can also correct the problem by pointing to the root certificates:

```

DRC.SetProp '.' 'RootCertDir' 'C:\..\TestCertificates\ca'
0
DRC.Clt args, ←'SSLValidation' 0
0 CLT00000052

```

(The final element above is redundant; 0 is the default value.) Once a Secure Client has been created, the rest of the communication works in exactly the same way as for a non-secure client.

## 2.3 Creating a Secure Server

Secure Servers are created using the same additional arguments as for Clients (see the previous section on creating secure clients for an explanation). Unlike a secure client, a secure server *must* have a certificate.

When a client has connected to a secure server, it is possible to use `GetProp` on the new connection to retrieve information about the client certificate. However, since client certificates are not *required*, information about client certificates is only transmitted to the server if this had been requested through the use of one of the flags `RequestClientCertificate` (64) or `RequireClientCertificate` (128). The former allows connections without client certificates and fetches information *if* the client has a certificate, the latter will only allow connections from clients which do have a certificate. If no client certificate has been requested, or no certificate exists, the certificate information will have zero rows when queried.

Note that validation of client certificates requires access to root certificates, so you must first have used `SetProp` to identify the folder containing these certificates.

The flags controlling certificates have the same meaning for a server as for a client, except that in the case of a server they are applied each time a new connection is made, rather than on creation of the server object. Note that connections which are rejected due to certificate validation failure do *not* generate events on the server that application code will need to handle.

## 2.4 Using the DRC.X509Cert Class

Conga v2.1 introduces a new class, `DRC.X509Cert`, to encapsulate certificate handling. `DRC.X509Cert` has methods to read certificates from files, folders, and Microsoft certificate stores. A complete description of the class can be found in the Reference section of this document.

Read one or more certificates from file. (Statements have been formatted for readability.)

```

path←Samples.CertPath
file←path,'client/client-cert.pem'
mycert←DRC.X509Class.ReadCertFromFile file
1

```

Since there's only a single certificate, we'll peel away the outermost layer of nesting.

```

←mycert←mycert
#.DRC.X509Cert.[X509Cert]
mycert.[NL] 2 A examine its properties
Cert CertOrigin Elements Extended Formatted KeyOrigin
LDRC ParentCert UseMSStoreAPI

```

`Elements`, `Extended`, and `Formatted` contain specific information about the certificate. `Elements` contains the information in a basic format while `Formatted` and `Extended` have the same elements in a more human readable format, and `Extended` may, in some instances, contain more information.

```

mycert.Elements.\nl -2
AlgorithmID AlgorithmParams Description EnhancedKeyUsage
Extensions FriendlyName Issuer IssuerID Key KeyContainer
KeyHex KeyID KeyLength KeyParams KeyProvider
KeyProviderType SerialNo Subject SubjectID ValidFrom
ValidTo Version

mycert.Elements.(ValidFrom ValidTo)
2008 2 15 11 19 50 0 2018 2 12 11 20 4 0
mycert.Formatted.(ValidFrom ValidTo)
Fri Feb 15 11:19:50 2008
Mon Feb 12 11:20:04 2018
mycert.Extended.(ValidFrom ValidTo)
Fri Feb 15 11:19:50 2008
Mon Feb 12 11:20:04 2018

mycert.\nl -3 A examine methods
AsArg Chain IsCert ReadCertFromFile ReadCertFromFolder
ReadCertFromStore

mycert.IsCert A my certificate is indeed a certificate!
1

```

**Certificate Chains**

A certificate chain, also known as a certification path, is a list of certificates used for authentication. The chain starts with the certificate for an entity. Each certificate in the chain is signed by the issuer which is the next entity in the chain. This continues until a root CA certificate is reached.

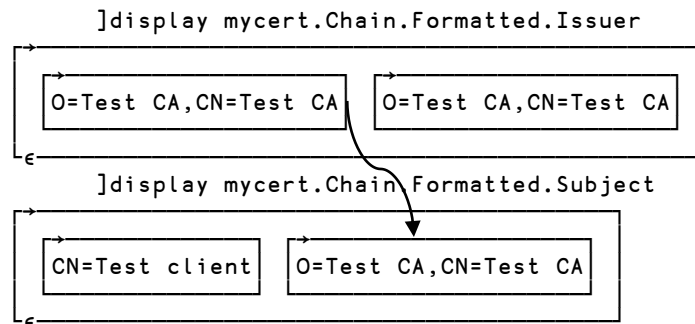
In Conga v2.0 you could specify a certificate chain by supplying a list of comma separated file names

```

mycert.Chain
#.DRC.X509Cert.[X509Cert] #.DRC.X509Cert.[X509Cert]

```

Here we have the certificate chain for the test client certificate which consists of the client certificate and the test CA certificate. You can verify this by seeing that is issuer for the lower certificate is the same at the subject for the higher one. You might also notice that the Test CA certificate is self-signed, meaning the subject and issuer are the same.



## 3. Samples

The distributed workspace **Conga** contains a number of working examples, which are intended to demonstrate how to use most of the capabilities of **Conga**. Although they are simple, many of the samples have enough functionality to be used as the starting point for communicating **Dyalog** applications.

For a complete list of **Conga** functions, see Appendix A.

### 3.1 Overview

The **Samples** workspace contains the following classes and namespaces:

<b>DRC</b>	The <b>Conga</b> interface functions – see Appendix A for a complete function reference
<b>FTPClient</b>	A class which implements a “Passive Mode” FTP Client, exposing functions to <b>List</b> the contents of a folder on an FTP Server, <b>Get</b> and <b>Put</b> in binary and text mode
<b>HTTPUtils</b>	A collection of utilities useful for manipulating HTTP headers
<b>Parser</b>	A utility which is used by the <b>TelnetClient</b> class to parse constructor options
<b>RPCServer</b>	A framework for a Remote Procedure Server based on Command-mode clients for communication between APL systems
<b>Samples</b>	A collection of functions which demonstrate and test everything else in the workspace. The function <b>Samples.HTTPGet</b> is a tool for extracting the contents of any web page
<b>TelnetClient</b>	A class which allows you to control Telnet sessions (log on to a remote computer, collect session output)
<b>TelnetServer</b>	An oversimplified Telnet server, which (unlike the <b>TelnetClient</b> ) does not properly support Telnet session option negotiation
<b>TODServer</b>	The simple “Time Of day” service, discussed in the introductory chapter
<b>WebServer</b>	A simple (but functional) HTTP Server, which can be used to provide simple Web Services

## 3.2 The Samples Namespace

The `Samples` namespace contains a number of functions with names beginning with `Test`. Between them, these examples should show examples of most of the different ways that the above components can be used. In alphabetical order, the test functions supplied with Conga v2.1 are:

<code>TestAll</code> , <code>TestAllSecure</code>	Cover-functions which run several other <code>Test</code> functions
<code>TestFTPClient</code>	Uses the <code>FTPClient</code> class to connect to <code>ftp.mirrorservice.org</code> and downloads the file <code>pub/readme.txt</code>
<code>TestSecureConnection</code>	Creates a secure server and connects a secure client to it, sends one transaction back and forth.
<code>TestSecureWebClient</code>	Secure version of <code>TestWebClient</code> .
<code>TestSecureWebServer</code>	Secure version of <code>TestWebServer</code> .
<code>TestSecureTelnetServer</code>	Secure version of <code>TestSecureTelnetServer</code> .
<code>TestRPCServer</code>	Starts an <code>RPCServer</code> on port 5050 and then starts a number of threads which make several remote procedure calls to functions <code>Foo</code> and <code>Goo</code> in the <code>RPCServer</code> namespace
<code>TestSimpleServices</code>	Tries to connect to and use the TOD (Time of Day) and QOTD (Quote of the Day) services on a named host
<code>TestTelnetServer</code>	Starts the <code>TelnetServer</code> sample and logs two sessions on to it (the <code>TelnetServer</code> example is oversimplified and should be reworked)
<code>TestWebClient</code>	Exercises the <code>HTTPGet</code> function
<code>TestWebFunctionServer</code>	Starts the <code>WebServer</code> example in the mode where it calls a user-defined function <code>WebServer.TimeServer</code> to handle all requests: illustrates how you can write APL code to provide “virtual” web pages
<code>TestWebServer</code>	Start the <code>WebServer</code> to serve pages from the <code>asp.net</code> samples folder, and start a bunch of threads which each use <code>HTTPGet</code> to request the text of a page from this folder.
<code>TestX509Certs</code>	Read a number of certificate and key files to build certificate variables used by the secure tests.

The following sections discuss most of the above examples in more detail, starting with Web Clients and Servers.

## 3.3 Web Client

The function `Samples.HTTPGet` shows how Conga can be used to retrieve the contents of a web page from an internet site. For example:

```
z←Samples.HTTPGet 'http://www.dyalog.com/news.htm'
```

The function returns 3 elements containing return code, HTTP headers, and data:

```
1→z
0
```

A return code of 0 indicates success; if the value is anything other than zero, the request has failed.

```
2→z
http/1.1 200 ok
date           Mon, 10 Dec 2007 09:10:28 GMT
server         Apache/2.2.4 (Ubuntu) PHP/...etc...
last-modified  Fri, 30 Nov 2007 00:15:32 GMT
etag           "228c056-50eb-5426d100"
accept-ranges  bytes
content-length 20715
content-type   text/html; charset=utf-8
```

The HTTP headers (above) are returned as a 2-column matrix of attribute names and values. Browsers use this information to know how to encode or decode data, and provide other functionality to the end user. The third element contains the data, as a character vector:

```
ρ3→z
20715
60†3→z
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//E
```

If the content-type header specified `charset=utf-8`, `HTTPGet` will also have performed the necessary decoding from UTF-8.

The function also supports the retrieval of web pages protected using “basic authentication”, if you supply a URL containing user id and password as follows:

```
z←Samples.HTTPGet 'http://user:pass@www.secret.com'
```

`HTTPGet` can also retrieve secure web pages and will make a secure connection if a left argument is provided or the URL begins with `https`. In this case, the 4<sup>th</sup> element of the result will contain certificate information for the server. If the server requires a client certificate, a left argument *must* be provided, containing the `CertFile`, `KeyFile` and optionally `SSLValidation` parameters for the connection.



The `HTTPGet` sample uses Conga as follows:

- [7] Call `DRC.Init` to ensure DRC is initialised
- [20] Arguments are constructed for use in constructing a secure client, if necessary.
- [25] A Basic Authentication header is added if a user id and password have been supplied.
- [30] `HTTPUtils.HostPort` is used to detect a trailing port number (for example `:8080`) if one was supplied, otherwise the default HTTP port of `80` (or `443` for SSL) is used
- [32] A Text-mode client is created, with a buffer size of 100k
- [33] An HTTP “GET” command is sent to the Web Server
- [37] We wait for blocks of data. As soon as some data has arrived, we call `HTTPUtils.DecodeHeader` to detect and extract the HTTP header
- [42] HTTP headers are decoded (if the entire header has been received).
- [45] We look for a content-length field in the header, as this will allow us to know how much data to expect
- [53–58] Various logic is used to determine whether all data has arrived. If the declared Content-Length has been delivered, if the server delivered data and closed the socket (`BlockLast` event), or if the text contains an ending `</html>` tag (the latter is perhaps questionable).
- [63] UTF-8 conversion of data if required.
- [67] Retrieval of the server certificate.
- [72] We close the client object

### 3.4 Web Server

The namespace `WebServer` contains an ultra-simple implementation of a web server. This does not provide any of the services that sophisticated web servers provide. Nonetheless, it shows just how little code is required to implement a web server that interfaces to a web browser. The `WebServer` example supports serving real files from the file system, or using APL functions to intercept requests and manufacture virtual pages on request. A much more complete example of a web server called the *MildServer* is available as an “Open Source” project on the APL Wiki.

The function `WebServer.Run` is used to start a web server; it takes an argument with three elements: The first is either a path to the root of the file system to be served up, or the name of a function in the active workspace which will intercept requests and manufacture output. The second and third elements are always the port number on which the server will listen, and the name of the Conga server object to create.

`Samples.TestWebServer` launches a web server which serves up pages from the `asp.net` tutorial folder. This is perhaps a little odd, as the sample web server does not support ASP.Net scripting, so any attempt to load `.aspx` or `.asmx` pages through the server will only result in the source of the pages being delivered to the client without any attempt to process the scripts contained within.

`Samples.TestWebFunctionServer` starts a web server which uses the function `WebServer.TimeServer` to generate output in response to each request. This simple function returns an echo of the request with a timestamp.

Both test functions start a number of threads and use `HTTPGet` to request pages and thus test that the web server that has been started is responding as expected.

The `WebServer.Run` function uses Conga as follows;

- [4] Call `DRC.Init` to ensure DRC is initialised.
- [13] Create a Raw-mode server. (Same as Text-mode except it returns byte numbers in the range 0-255.)
- [24] Loop on `Wait`, timing out every 10 seconds.
- [27] Switch on the first element of the result of `Wait`.
- [28] Code 0: `Wait` returned an application event.
- [31] The event was an error on the socket, which will have been closed. Clean up the data namespace for the client. (`SpaceName` generates a namespace name based on the IP address and port number of the client.)
- [38] Data arrived. Find the client namespace and call `HandleRequest` in a new thread, passing the object name and input data to it. `HandleRequest` will eventually call `DRC.Send` to send the answer to the client.
- [46] If the client closed the connection, expunge the namespace.
- [50] Insert code here to react to `Connect` events.
- [56] If the return code was 100, nothing happened for 10 seconds. Insert timekeeping code here. (For a busy web server, we will need to do housekeeping even if we don't have any timeouts.)
- [60] If `Wait` returned 1010 (Object not found), this means that the server object has disappeared. This probably means that another thread has closed it. Many of the test functions do this once they have completed client tests.
- [72] If we get to the end of the while loop, this means that we shut down because some component of the server set stop to 1. We close the server object ourselves.

The `WebServer.HandleRequest` function manufactures the response to each web request. In "file server" mode, it calls `GetAnswer`, which tries to read a file. Otherwise, it calls the function nominated when the server was created. Finally, it formats an HTTP response with status information, a Content-Length header, and the content, and used `DRC.Send` to transmit the response to the client.

### 3.5 RPC Client and Server

The RPC Server is similar in structure to the web server discussed in the previous section, except that Command sockets are used to transmit Remote Procedure Calls to the server, which validates and executes them, and returns the array result to the client. Both client and server need to be Conga users. A Command-mode client application for use by other languages is possible; similar tools have been built in the past. However, most non-APL clients already support SOAP/XML for remote

procedure calls in the form of Web Services. In April 2011, Dyalog released SAWS, the Stand Alone Web Services framework, as a means to enable users to provide and consume Web Services from APL.

`RPCServer.Run` is very similar to `WebServer.Run`:

- [4] Call `DRC.Init` to ensure DRC is initialised.
- [6-12] In order to be able to return an error if it is unable to start the server, `Run` first creates the Command-mode server on line [7], and only starts a new handling thread on line [9] by calling itself recursively with a left argument of 0 if the server could be created. The handler continues execution from the `:While` on line [15].
- [16] Loop on `Wait`, timing out every 5 seconds.
- [18] Switch on the first element of the result of `Wait`.
- [19] Code 0: `Wait` returned an application event:
- [21] The event was an error. If the object in error was the server itself, we close it and stop running. Otherwise, the error is ignored. (We might do some housekeeping if we were tracking client sessions.)
- [27] Data arrived. Validate the format of the incoming array, and confirm the first element names a function that may be called. If all is OK, run `Process` in a new thread, passing the object name and input data to it. `Process` will eventually call `DRC.Progress` and subsequently `DRC.Respond` to first signal progress and finally send the answer to the client.
- [38] We ignore connection events.
- [44, 46] Codes 100 and 1010: See `WebService.Run` in the previous section.
- [72] If we get to the end of the while loop, this means that we shut down because some component of the server set stop to 1. We close the server object ourselves.

### 3.6 FTP Client

The `FTPClient` class implements a basic Passive Mode FTP client. The code is essentially the same as the FTP workspace which was distributed with versions 11.0 and earlier, rewritten to use Conga, and cast in the form of a class. The function `Samples.TestFTPClient` shows an example of its use, by listing the contents of the `pub` folder at `ftp.mirror.service.org`, and retrieving the `readme.txt` file from this folder.

The use of Conga by the class is as follows:

- `Open[4]` Call `DRC.Init` to ensure DRC is initialised.
- `Open[6]` Create a Text-mode client for issuing commands to the FTP server.
- `Open[11-13]` Use the function `Do` to enter user ID and password and check for the expected responses from the server.

<code>Do[5]</code>	Send a command to the server. <code>Do</code> returns the FTP state code following the command.
<code>ReadReply[106]</code>	Wait for a response, called by <code>Do</code> .
<code>GetData[3]</code>	Execute the PASV command to prepare for passive-mode data transfer; server returns a dataport that it has opened.
<code>GetData[4]</code>	Connect a Text- or Raw-mode client to the dataport identified by PASV.
<code>GetData[5-6]</code>	Set ASCII or Binary mode and issue the command which will return data.
<code>GetData[9-11]</code>	Wait and collect output response until the server closes connection.
<code>GetData[14]</code>	Confirm the server thinks transfer was completed.
<code>PutData[6-8]</code>	Same as <code>GetData[4-6]</code>
<code>PutData[10]</code>	Send all data in a single call to <code>DRC.Send</code> ; Conga will break the data up into TCP Packets.

The FTP client protocol is surprisingly easy to implement.

### 3.7 Telnet Server

These examples were developed by Dyalog for internal use in testing. They are provided as examples but not documented to the same degree as other examples. The server does not really deserve to be called a 'Telnet Server', as it does not support feature negotiation. However, it is perhaps mildly entertaining. If you start the server by typing:

```
TelnetServer.Run&@
Dyalog Timesharing System 'TELNETSRV' started on port 23
```

You can now start a Telnet session and connect to your own APL timesharing system. Start a command shell and enter "Telnet localhost" to connect to the server. If your machine is already running a Telnet server, you will need to modify the line in `TelnetServer` which sets the variable `port`, and modify the command to `Telnet localhost:nnnn` where `nnnn` is the new number you picked. You can also connect from another machine on the network if you replace `localhost` by the network name or IP address of the machine on which you started the server:

```

Administrator: Command Prompt
Please login to the Dyalog Timesharing System!

User: mkrom
Password: secret
Welcome to the Dyalog Timesharing System!
  1 2 3+4 5 6
5 7 9
  A 1 2 3 4 5 6
  B 100 200 300
  #NL 2

A
B
  A,B
1 2 3 4 5 6 100 200 300
+/A,B
621
)END
Dyalog Timesharing System Shutting Down...

Connection to host lost.

C:\Users\mkrom>

```

It is a ‘real’ multi-user system, using a namespace to contain the ‘private workspace’ for each session. Two substitutions of input characters are made: use `_` for `←` and `#` for `⊠`. Modify `TelnetServer.Process` to extend its functionality. If you find a terminal program which supports UTF-8, and modify the server code to translate it, you can support entry of APL symbols.

**Watch Out!** You might be in violation of your Dyalog licence if you allow anyone else to use it. And don’t leave it running for long in this form; it opens up a rather large back door which will allow the knowledgeable hacker to wreak havoc with your machine.

### 3.8 Telnet Client

This class implements a Telnet Client which supports Telnet feature negotiation and can be used to log in to systems which provide Telnet access. It has been developed by Dyalog for use in driving automated quality-assurance scripts and is provided without documentation. The login sections have been tweaked to work with the Unix systems that we need to use, it may need further work to connect to new servers.

### 3.9 TODServer

This example was discussed in detail in Chapter 1.

# Appendix A: Function Reference

This section documents the functions in the DRC namespace which are intended for use by applications, in alphabetical order. Any additional functions found in the namespace are for internal use and should not be called by application code.

## Checking DRC Return Codes

The first element of the result returned by many of the functions in the DRC namespace is a return code. If this element is 0, no error occurred and the rest of the result is as described in the function documentation. If the element is not 0, the result returned by the function is a three element vector consisting of [1] return code, [2] error name, [3] error description, if available.

Therefore, it is recommended that you first check the return code in the result before attempting to process the other elements of the result.

```
rc stores←DRC.Certs arg      A not recommended,
                             LENGTH ERROR could result
```

The following code snippet demonstrates one technique to properly check the return code.

```
:if 0≠1↑res←DRC.Certs arg
  rc err desc←res
  ...                               A error processing
:else
  rc stores←res
  ...                               A normal processing
```

### DRC.Certs

**Purpose** Provide low level interface to read certificates. `DRC.Certs` is documented here for completeness. It is strongly recommended that the `DRC.X509Cert` class be used for certificate processing.

**Syntax**

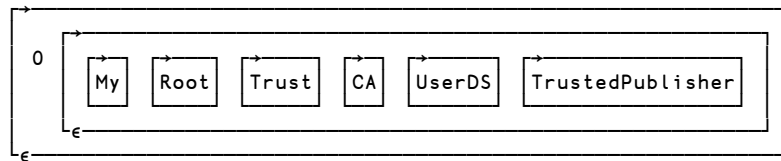
```
rc stores ← DRC.Certs 'ListMSStore'
rc certs ← DRC.Certs 'MSStore' storename
rc certs ← DRC.Certs 'DER' filename
```

<b>rc</b>	return code	0	0 if no error, non-zero if an error occurred (see Checking DRC Return Codes at the beginning of the Function Reference)
<b>stores</b>	store names	'My' 'Root'	'ListMSStore' will return a vector of character vectors each containing a Microsoft certificate store name.
<b>certs</b>	raw certificate data		A vector of integer vectors each

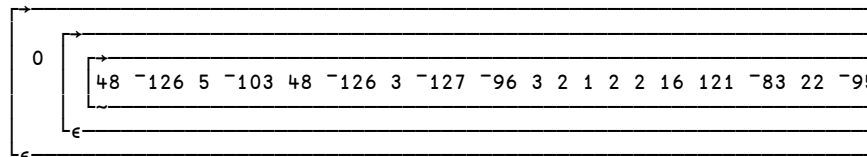
			containing raw certificate information.
<code>storename</code>	store name	<code>'CA'</code>	<code>'MSStore'</code> <code>storename</code> will return all of the certificates in the named store.
<code>filename</code>	folder and filename of the certificate file		<code>'DER'</code> <code>filename</code> will return all of the certificates in the named file. Files should be of type <code>.pem</code> or <code>.cer</code> .

Examples

```
]display DRC.Cert 'ListMSStore'
```



```
]display DRC.Cert 'MSStore' 'Root'
```



(display has been truncated, but you get the idea)

## DRC.Cl t

**Purpose** Creates a Conga Client.

**Syntax** `rc name←DRC.Cl t Name Address Port Mode BufferSize SSLValidation EOM IgnoreCase Protocol X509`

**Note that parameters can be named from Conga v2.0.**

<code>rc</code>	return code	<code>0</code>	<code>0</code> if no error, non-zero if an error occurred (see Checking DRC Return Codes at the beginning of the Function Reference)
<code>name</code>	client name	<code>'C1'</code>	If empty, will be generated (and returned in result).
<code>Address</code>	IP address or name	<code>'192.168.1.1'</code> <code>'www.dyalog.com'</code>	Address of server to connect to.
<code>Port</code>	service port	<code>80</code>	Port that server is listening on.
<code>Mode</code>	type of client	<code>'Command'</code>   <code>'Raw'</code>   <code>'Text'</code>	See <code>DRC.Srv</code> for discussion.
<code>MaxSize</code>	buffer size	<code>10000</code>	Maximum data size

<code>CertFiles</code>	public certificate	<code>mycert.pem</code>	File names, sep. by commas, of the public certificate and any files in the signing chain.
<code>KeyFile</code>	private key	<code>mykey.pem</code>	File name
<code>SSLValidation</code>	SSL flags	32	See appendix C
<code>EOM</code>	End of message	( <code>␣UCS 13 10 13 10</code> )	Termination String(s) – a simple character vector or a vector of vectors.  Applies to Text/Raw mode only.
<code>IgnoreCase</code>	Re EOM	1	1 to ignore case when looking for EOM.
<code>Protocol</code>	IPv4 or -v6	IPv4	IPv4 or IPv6: Default is IP – which means try IPv6 then -v4.
<code>X509</code>	X509Cert instance		Refer to the X509Cert documentation elsewhere in the reference.

Examples

```
DRC.Clt 'C1' '192.168.1.1' 5050
```

(Command-mode client of server at port 5050 at address 192.168.1.1)

```
DRC.Clt '' 'localhost' 13 'Text' 1000 ('EOM'(␣UCS 13 10))
```

(Text-mode client with an auto-generated name, connected to server on port 13 on the same machine, with a maximum buffer size of 1000 characters, and termination sequence of CRLF)

```
myCert↔DRC.X509Cert.ReadCertFromFile 'client-cert.pem'
DRC.Clt 'C1' '192.168.1.1' 5050, c('X509' myCert)
```

(Secure command-mode client of server at port 5050 at address 192.168.1.1)

Typical Errors

- 1009 Object name already in use
- 1110, 1111 Nothing seems to be listening on the port

Plus all the TLS handshaking errors listed in Appendix C.

**DRC.Close**

**Purpose** Closes any Conga object

**Syntax** `rc ← DRC.Close name`



<code>rc</code>	return code	0	0 if no error, non-zero if an error occurred (see Checking DRC Return Codes at the beginning of the Function Reference)
<code>name</code>	object name	'C1'	Should be the name of an existing Conga object

Example

```
DRC.Close 'C1'
0
```

Typical Errors

1010 Object name does not exist

**DRC.Describe**

**Purpose** Return a description for a Conga object.

**Syntax** `rc desc ← DRC.Describe name`

<code>rc</code>	return code	0	0 if no error, non-zero if an error occurred (see Checking DRC Return Codes at the beginning of the Function Reference)
<code>desc</code>	object description		Varies by object
<code>name</code>	object name	'.'   'C1'	Name of the object to describe

`DRC.Describe` is similar to `DRC.Tree` except that a) it returns information for only the named object and not its children as well, and b) the descriptions are textual rather than numeric codes.

For all objects except the root, the first three elements of the object description are: [1] Name, [2] Type, [3] State.

The root object has [1] Name, [2] Version, [3] State, [4] Thread count.

Commands and messages (object types 4 and 5) have the additional elements [4] expected size, and [5] received size. These are useful check when receiving large command result. If the client times out, you can use these elements to determine if the server is still sending information.

**Examples**

```
DRC.Describe '.'
0 [DRC] Conga Dynamic Link Library 0.0.8590.0 Copyright
(C) 2004-2011 Dyalog Ltd. built Mar 24 2011 09:56:16
State=RootInit Threads=2
```

```
DRC.Describe 'C1'
0 CLT00000000 Client Connected
```

See also `DRC.Tree` and `DRC.Names`

**DRC.Error**

**Purpose** Converts an error number into a textual identification or description of the error.

**Syntax** (no name desc) ← DRC.Error no

no	error number	1201
name	name of the error	ERR_TLSHANDSHAKE
desc	description (optional)	/* unable to complete a TLS handshake with the peer */

Example

```
DRC.Error 1009
1009 ERR_NAME_IN_USE
```

**DRC.Exists**

**Purpose** Tests the existence of an object.

**Syntax** bool ← DRC.Exists name

bool	result	1	1 if the named object exists, else 0
name	object name	'C1'	an object name

Example

```
DRC.Exists 'C1'
1
```

**DRC.flate.Deflate**

**Purpose** Compresses data using the deflate compression scheme. Used to implement server-side HTTP compression.

**Syntax** comp ← DRC.flate.Deflate data

data	data to be compressed	Integer vector with values in the range 0-255. If converting from character data, use 'UTF-8' UCS data
comp	compressed data	Integer vector with values in the range 0-255. The first 2 elements comprise a header for the zlib wrapper for the compressed data. This header should generally be stripped off before sending to the client.

Example

```
DRC.flate.Deflate 256|83 [DR 2000p'this is a test'
120 156 43 201 200 44 86 0 162 68 133 146 212 226 146 146 81
222 40 111 148 55 202 27 229 141 242 70 121 67 144 7 0 17 217
213 243
```

**DRC.flate.Inflate**

**Purpose** Uncompresses data that has been compressed using the deflate compression scheme. Used to implement client-side HTTP compression.

**Syntax** data ← DRC.flate.Inflate comp

comp	compressed data	Integer vector with values in the range 0-255. The first 2 elements should be a header, 120 156, for the zlib wrapper for the compressed data. If the header is not present, it should be prepended before Deflate is called.
data	uncompressed data	Integer vector with values in the range 0-255. To convert

Example

```
tmp←DRC.flate.Deflate 256|83 □DR 'this is a test'
'UTF-8' □UCS DRC.flate.Inflate 256|83 □DR tmp
this is a test
```

**DRC.flate.IsAvailable**

**Purpose** Tests if the deflate compression library is loaded

**Syntax** bool ← DRC.flate.IsAvailable name

bool	result	1	1 if the deflate compression library is loaded, else 0
------	--------	---	--

Example

```
DRC.flate.IsAvailable
1
```

**DRC.GetProp**

**Purpose** Retrieving properties from a Conga object.

**Syntax** rc res ← DRC.GetProp obj property

rc	return code	0 if no error, non-zero if an error occurred (see Checking DRC Return Codes at the beginning of the Function Reference)
res	properties	Depends on the object and the property requested – see table below.
obj		Name of the Conga object
property		Name of the property

Examples:

```
DRC.GetProp '.' 'PropList'
0 PropList Protocol ReadyStrategy RootCertDir
DRC.GetProp '.' 'RootCertDir'
0 '/usr/rootcerts'
```

Supported Properties:

Name	Objects	Description
<code>LocalAddr</code>	Client, Server, Connection	Your own address: [1] Protocol, [2] Address Name, [3] Address Bytes, [4] Port
<code>OwnCert</code>	Client, Server, Connection	<code>X509Cert</code> object containing information about your own certificate.
<code>PeerAddr</code>	Client, Connection	Address of peer – same format as <code>LocalAddr</code> .
<code>PeerCert</code>	Client, Connection	<code>X509Cert</code> object containing information about the certificate used at the other end of the connection.
<code>PropList</code>	All	Returns the list of property names
<code>Protocol</code>	Root ('.')	The protocol to use ( <code>IPv4   IPv6   IP</code> ). The Root property is inherited if none other is specified when making a connection. If set to <code>IP</code> (the default), Conga will try IPv6 and then IPv4 if that fails.
<code>ReadyStrategy</code>	Server	Strategy to use to decide which connection to report as ready when more than one connection has received data. 1="Round Robin" in the object tree, 2="Oldest First" This is the default strategy and is considered to be the "fairest" but consumes slightly more CPU than 1. There is also a strategy of 0="Use First" which will use the first object in the tree. This is potentially dangerous as other object in the tree may not be serviced.
<code>RootCertDir</code>	Root ('.')	Name of the folder containing Certificate Authority root certificates (should be set using <code>SetProp</code> )

**More about Certificate Information:**

Certificate information is returned as an `X509Cert` object. The specific content may vary from certificate to certificate, but will normally include the certificate issuer, subject, public key algorithm, certificate format version, serial number, valid from & to dates. If no certificate exists, or in the case of a `Server` object no certificate information has been requested (see the section on TLS Flags in Appendix C), the result will be an empty vector.

Certificate information can be used to validate a peer certificate in combination with flags such as `CertAcceptWithoutValidating` (see the TLS Flags section). It can also allow a `Server` to confirm the identity of a `Client` without requiring a login.

**DRC.Init**

**Purpose** Loads and initializes, or reinitializes the Conga DLL or Shared Library

**Syntax** `rc ← {reset} DRC.Init ''`

<b>rc</b>	return code	0	0 if no error, non-zero if an error occurred (see Checking DRC Return Codes at the beginning of the Function Reference)
<b>reset</b>	reset code	0	<p>If Conga has been loaded previously, subsequent calls to DRC.Init will have the following effect:</p> <p>If <b>reset</b> is not supplied or is not either 1 or -1, return a message indicating that Conga has already been loaded.</p> <p>If <b>reset</b> = 1, close any existing Conga objects.</p> <p>If <b>reset</b> = -1, reload the Conga DLL or Shared Library.</p>

The right argument is currently unused but is reserved for future extensions.

**Example**

```
DRC.Init ''
0 Conge loaded from...
```

**Typical Errors**

1000 Unable to load the library

**DRC.Names**

**Purpose** Returns names of existing objects

**Syntax** `names ← DRC.Names root`

<b>names</b>	object names	Names of children of root
<b>root</b>	root object	'' 'C1'

**Example**

```
DRC.Names ''
C1 C2 C3
DRC.(Close''Names '')
0 0 0
```

See also DRC.Describe and RC.Tree

**DRC.Progress**

**Purpose** Send any APL array as a progress report to a client waiting on the named Command. A server can call Progress any number of times before Respond.

**Syntax** `rc ← DRC.Progress name data`

<b>rc</b>	return code	0	0 if no error, non-zero if an error occurred (see Checking DRC Return Codes at the beginning of the Function Reference)
-----------	-------------	---	---

**name**          command name      'C1.CMD1'  
**data**          any array

**Example**

```
DRC.Progress (2>waitresult) 'Task 50% completed'
```

If the result of `Wait` on a `Command` is in `waitresult`, the above expression will send a progress report to the client.

## DRC.Respond

**Purpose**    Send any APL array as the response to a command.

**Syntax**    `rc ← DRC.Respond name data`

**rc**          return code            0                    0 if no error, non-zero if an error occurred (see Checking DRC Return Codes at the beginning of the Function Reference)

**name**        command name            'C1.CMD1'

**data**        any array

**Example**

```
DRC.Respond (2>waitresult) (Process 4>waitresult)
```

If the result of data received using `Wait` on a `Command-mode` server is in `waitresult`, the above expression will call the function `Process` on the data which accompanied the most recent command and send the result to the client.

See also `DRC.Progress` and `DRC.Wait`

## DRC.Send

**Purpose**    Send data to partner.

**Syntax**    `rc [command] ← DRC.Send name data [close]`

**rc**          return code            0                    0 if no error, non-zero if an error occurred (see Checking DRC Return Codes at the beginning of the Function Reference)

**command**    command name            'C1.CMD1'            Generated from Client name if necessary (in command mode)

**name**        client / command        'C1'  
   'C1.CMD1'            Client or Command name. In command mode, if a client name is supplied, command name is generated automatically.

**data**        any array



			Function Reference)
<b>Name</b>	server name	'S1'	If empty, will be generated (and returned in result).
<b>Address</b>	IP address or name	'192.168.1.1' 'localhost'	If empty, server will listen on all addresses that the machine has.
<b>Port</b>	service port	80	Port number to listen on.
<b>Mode</b>	type of client	'Command'   'Raw'   'Text'	See the discussion on Mode and EOM following this table.
<b>MaxSize</b>	buffer size	16384	Maximum data size
<b>SSLValidate</b>	SSL flags	32	See appendix C
<b>EOM</b>	End of message	([]UCS 13 10 13 10)	Termination String(s) – a simple character vector or a vector of vectors.
<b>IgnoreCase</b>	Re EOM	1	1 to ignore case when looking for EOM.
<b>Protocol</b>	IP Protocol to use	IPv4	IPv4, IPv6 or IP: Default is to use both if available (IP)
<b>X509</b>	X509Cert instance		Refer to the X509Cert documentation elsewhere in the reference.

**Mode and EOM**

In Command mode, each transmission consists of an entire APL array; the `DRC.Wait` function will only terminate when the entire APL array reaches its destination. In Raw or Text modes, byte streams are transmitted. In Text mode these are translated to a character vector on receipt; in Raw mode, integers between 0 and 255 are returned.

In Text and Raw modes, you can set the End-of-message parameter, in which case `DRC.Wait` will terminate on receipt of the specified termination string. If an empty termination string is specified, `DRC.Wait` will terminate when the buffer contains `size` bytes.

If no stop criteria are specified, `DRC.Wait` will return data each time a TCP packet is received. If the packet is larger than `size` the data will be returned in chunks of the specified size.

**Examples**

```
DRC.Srv 'APLRPC' '' 5050 'Command'
0 APLRPC
```

Creates a Command mode server listening on port 5050.



```
DRC.Srv ' ' ' 23 'Text' 1000 ('EOM' (␣UCS 13))
0 SRV00000001
```

Creates a Text-mode server listening on port 23, with a maximum buffer size of 1000 characters, and termination sequence of CR, with an auto-generated name.

```
cert↔DRC.X509Class.ReadCertFromFile 'server-cert.pem'
cert.KeyOrigin←'DER' 'server-key.pem'
certs←('X509' cert)('SSLValidation' 64)
DRC.Srv 'APLRPC' ' ' 5050 'Command',certs
0 APLRPC
```

Creates a secure Command server on port 5050 of the local machine using the named certificate and key files, and the SSL Validationflag value 64 (RequestClientCertificate).

**Typical Errors**

- 1009                                   Object name already in use
- 1110, 1111                          Nothing seems to be listening

Plus all the TLS-specific error listed in Appendix C.

**DRC.Tree**

**Purpose**   Return state information about an object and all of its “children”.

**Syntax**   rc tree ← DRC.Tree root

<b>rc</b>	return code	0	0 if no error, non-zero if an error occurred (see Checking DRC Return Codes at the beginning of the Function Reference)
<b>tree</b>	tree (see below)		Description of the object and its child nodes
<b>root</b>	root object	'.'   'C1'	Root object

A tree is a 2-element vector. The first element describes the root object, and the second element is a vector of trees describing each of its children, empty if the object has no children. `DRC.Tree '.'` returns a complete tree of all existing objects.

For all objects, the first three elements of the object description are: [1] Name, [2] Type, [3] State. The root object also has additional information: [4] Version, [5] Semaphore count. Commands and messages (object types 4 and 5) have the additional information [4] expected size, and [5] received size. These are useful to check when receiving large command result. If the client times out, you can use these elements to determine if the server is still sending information.

**Object Type codes (2nd element)**

- 0       Root
- 1       Server
- 2       Client
- 3       Connection
- 4       Command
- 5       Message

**State codes (3rd element)**

- 0 New
- 1 Incoming
- 2 RootInit
- 3 Listen
- 4 Connected
- 5 APL
- 6 ReadyToSend
- 7 Sending
- 8 Processing
- 9 ReadyToRecv
- 10 Receiving
- 11 Finished
- 12 MarkedForDeletion
- 13 Error
- 14 DoNotChange
- 15 Shutdown
- 16 SocketClosed
- 17 APPLast
- 18 SSL

**Example**

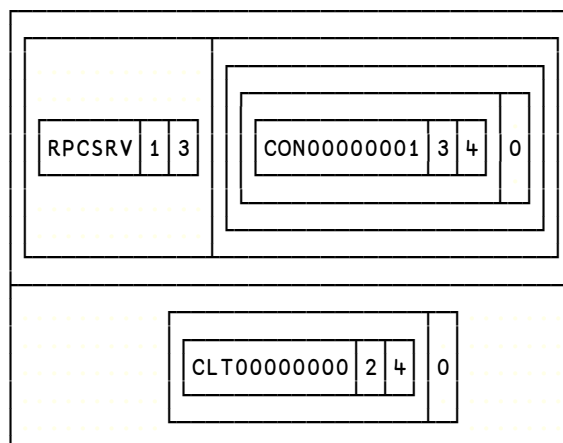
```
(rc (root subtree))+DRC.Tree '.'
rc
0
```

DISP root

0	2	Conga.Dynamic.Link.Library.1.1.14.0.Copyright...	1
---	---	--	---

**Interpretation** The root object name is empty, it is an object of type 0 (Root), the state is 2 (RootInit). The number of semaphores currently in use for thread synchronization is 1.

DISP 2 1psubtree



**Interpretation** There are two children of the root, named RPCSRV and CLT00000000 (this example was ‘shot’ while running Samples.TestRPCserver). RPCSRV is of type 1 (Server) and in state 3 (Listen). It has a child named CON00000001 which is a Connection (3) in state Connected (4). CLT00000000 is a client (2) in state Connected (4).

See also `DRC.Describe` and `RC.Names`

**DRC.Version**

**Purpose** Return the Conga version

**Syntax** `ver ← DRC.Version`

```
ver      version      2 2 11167  [1] major version
                                     [2] minor version
                                     [3] build number
```

Example

```
      DRC.Version
2 2 11167
```

**DRC.Wait**

**Purpose** Wait for an event to occur.

**Syntax** `rc obj event data ← DRC.Wait name [timeout]`

<code>rc</code>	return code	0 100	0 means data was received 100 means timeout other non-zero values indicate an error occurred (see Checking DRC Return Codes at the beginning of the Function Reference)
<code>obj</code>	object name	'C1.CMD1'	Object on which an event occurred
<code>event</code>	event name	'block'	See below
<code>data</code>	received data		
<code>name</code>	root name	'C1'	If a server or connection name is used, Wait will report events on the named object or any any of its children. If a command mode client wishes to wait on a specific command, the full command name can be given.
<code>timeout</code>	how long to wait	1000	Optional: Number of milliseconds to wait before timing out. If not specified, defaults to 1000 milliseconds.

**Events**

The following events can be reported in the 3rd element of the result

- Block** A block of data was received (Text or Raw mode) and the connection is still open.
- BlockLast** A block of data was received and the connection was closed; no more data can be expected. If the connection is closed while it is inactive, a **BlockLast** event will be reported with empty data.
- Connect** `obj` is a newly created connection.

<b>Error</b>	An error occurred (data will contain an error message)
<b>Progress</b>	Command mode client only: The server transmitted data using the <code>DRC.Progress</code> function
<b>Receive</b>	Command mode only: data received

**Command-mode server examples:**

```
DRC.Wait 'RPCSRV' 5000
0 RPCSRV.CON00000007 Connect 0
```

A client connected to the server, the connection name is `RPCSRV.CON00000007`.

```
DRC.Wait 'RPCSRV' 1000
0 RPCSRV.CON00000002.Auto00000000 Foo 1
DRC.Respond 'RPCSRV.CON00000002.Auto00000000' (Foo 1)
```

A command arrived from a command mode client, and we responded to it with the result of executing `(Foo 1)`.

**DRC.X509Cert class**

**Purpose** Provide a container to encapsulate X.509 style certificates and is the recommended technique to use when providing secure communications.

**Methods** `certs` ← `DRC.X509Cert.ReadCertFromFile filename`  
`certs` ← `DRC.X509Cert.ReadCertFromFolder pathname`  
`certs` ← `DRC.X509Cert.ReadCertFromStore storename`

The shared methods listed above are all used to read certificates from various sources. The instance methods are listed below.

- `bool` ← `#.DRC.X509Cert.[X509Cert].IsCert`  
(verifies that the object is a structurally valid certificate)
- `certs` ← `#.DRC.X509Cert.[X509Cert].Chain`  
(returns the certificate chain for this certificate)
- `args` ← `#.DRC.X509Cert.[X509Cert].AsArg`  
(formats the certificate for use with `Cl t` and `Srv`. This is an internal routine and need not be used by the Conga user but is documented here for completeness.)

<b>certs</b>	vector of certificate instances		Each element is of type <code>DRC.X509Cert</code>
<b>filename</b>	character vector certificate file name	<code>'server-cert.pem'</code>	This is a single file name. The file may contain multiple certificates.
<b>pathname</b>	character vector path name	<code>'c:\mycerts\*.pem'</code>	Wildcarding is allowed. All of the filenames found need to be valid certificate files.
<b>storename</b>	Microsoft certificate store	<code>'Root'</code> <code>'CA'</code>	Returns all of the certificates in the specified certificate store.

	<code>name</code>		<code>DRC.Certs 'ListMSStore'</code> will return the list of certificate store names.
<code>bool</code>	Boolean value	<code>1   0</code>	Returns 1 if the certificate is of valid structure.
	<code>args</code>		Used internally by <code>Cl t</code> and <code>Srv</code>

**Properties** Each instance of `DRC.X509Cert` has the following properties.

<code>Cert</code>	Raw certificate data in integer vector form.
<code>CertOrigin</code>	'MSStore' <code>storename</code> for certificates read from a certificate store. 'DER' <code>filename</code> for certificates read from a file
<code>Elements</code>	Namespace containing unformatted certificate elements. See discussion of <code>Elements</code> , <code>Extended</code> , and <code>Formatted</code> below.
<code>Extended</code>	Namespace containing extended certificate elements. See discussion of <code>Elements</code> , <code>Extended</code> , and <code>Formatted</code> below.
<code>Formatted</code>	Namespace containing formatted certificate elements. See discussion of <code>Elements</code> , <code>Extended</code> , and <code>Formatted</code> below.
<code>KeyOrigin</code>	'MSStore' <code>storename</code> for keys read from a certificate store. 'DER' <code>filename</code> for keys read from a file
<code>LDRC</code>	Reference to the local DRC namespace. Used internally within the class.
<code>ParentCert</code>	If this certificate is part of a certificate chain and has a parent, <code>ParentCert</code> is an instance of the parent certificate.
<code>UseMSStoreAPI</code>	Boolean to indicate whether to use the Microsoft certificate store API (1) or the gnuTLS API (0).

The `Elements`, `Extended`, and `Formatted` namespaces contain certificate elements in different formats. Not all certificates will have values for all elements. Some elements are more useful than others; `ValidFrom` and `ValidTo` for instance, are useful for determining if the certificate has expired. This section is not intended to be a comprehensive reference of X.509 certificate structure, but is here to provide the user with a basic understanding of the data with a certificate. Some of the more interesting elements are listed below.

<code>AlgorithmID</code>	The cryptographic algorithm used to generate the signature.
<code>Description</code>	Certificate description.
<code>Issuer</code>	The issuer of the certificate. This is useful for validating certificate chains. The issuer of a certificate should match the subject of the parent certificate. Self-signed certificates have identical <code>Issuer</code> and <code>Subject</code> elements.
<code>Key</code>	For certificates that include a key, this is the key. Keys should be carefully secured and not shared.

<b>KeyHex</b>	The key in hexadecimal format.
<b>KeyID</b>	The type of key.
<b>KeyLength</b>	Length of the key.
<b>SerialNo</b>	The certificate serial number. This number is unique for a given CA.
<b>Subject</b>	The subject of the certificate. Used in conjunction with the <b>Issuer</b> element
<b>ValidFrom, ValidTo</b>	Together , these elements determine the period of validity for the certificate.
<b>Version</b>	The X.509 version number of the certificate. As of the writing of this document, version 3 is the most current.

## Appendix B: Creating and Converting Certificates

There are many different file formats that can be used for storing X.509 certificates including PEM, CER, DER, PFX, P7C and P12. The popularity of the formats varies from one platform to the next. Conga supports the most popular formats, PEM and DER format files, with file extensions .pem or .cer or .der - on all platforms. To use certificates supplied in other formats, these must first be converted using freely available tools such as GnuTLS and OpenSSL (a guide to converting between formats using OpenSSL is available at:

<http://gagravarr.org/writing/openssl-certs/general.shtml>).

PEM files begin and end with:

```
-----BEGIN CERTIFICATE-----  
-----END CERTIFICATE-----
```

And contain a base 64 encoded version of the certificates.

Some .cer files contain spaces as line separators, but the gnutls tools used by conga only supports unix or windows standard line endings. You can check a .cer file by opening it in notepad with wordwrap turned off. If you can see a space after the -----BEGIN CERTIFICATE----- header you will need to replace all spaces in the base64 encoded portion of the file with newlines before the file will be recognised by Conga.

GnuTLS (see <http://www.gnu.org/software/gnutls/>) comes with a command line tool called certtool that can be used for creating certificates, keys & certificate requests in pem format files. Its documentation can be found at

[http://www.gnu.org/software/gnutls/manual/html\\_node/Invoking-certtool.html](http://www.gnu.org/software/gnutls/manual/html_node/Invoking-certtool.html)

The following examples show the most common commands for creating private keys, certificates, and certificate requests. Working through the samples in order will generate a self signed CA certificate/key pair and a certificate/key pair that can be used by a client or server (assuming their peer has a copy of the ca-cert.pem file).

### Create a self signed test CA certificate & key:

```
certtool --generate-privkey --outfile ca-key.pem  
certtool --generate-self-signed --load-privkey /  
ca-key.pem --outfile ca-cert.pem
```

These self-signed files can be used to run an in house CA, or for signing test certificates.

### Create a private key & certificate request

```
certtool --generate-privkey --outfile client-key.pem  
certtool --generate-request --load-privkey client-key.pem /  
--outfile client-request.pem
```

This certificate request can now be turned into a certificate by a CA such as Verisign, or your own in house CA using files generated in the “Create a self-signed test CA” section above.

**Generate & Sign a certificate request**

```
certtool --generate-certificate --load-request /
client-request.pem --outfile client-cert.pem /
--load-ca-certificate ca-cert.pem --load-ca-privkey ca-
key.pem
```

This creates a certificate that can be used by your client applications with the key file that was used to generate the certificate request.



## Appendix C: TLS Flags

### TLS Flags

**Purpose** These flags can be added together and passed to `ClT` or `Svr` to control the certificate checking process. If you do not require any of these flags simply pass 0 as the flags parameter of these functions.

<i>Code</i>	<i>Name</i>	<i>Description</i>
1	<code>CertAcceptIfIssuerUnknown</code>	Accept the peer certificate even if the issuer (root certificate) can't be found.
2	<code>CertAcceptIfSignerNotCA</code>	Accept the peer certificate even if it has been signed by a certificate not in the trusted root certificates directory.
4	<code>CertAcceptIfNotActivated</code>	Accept the peer certificate even if it is not yet valid (according to its valid from information).
8	<code>CertAcceptIfExpired</code>	Accept the peer certificate even if it has expired (according to its valid to information).
16	<code>CertAcceptIfIncorrectHostName</code>	Accept the peer certificate even if its hostname does not match the one we asked to connect to.
32	<code>CertAcceptWithoutValidating</code>	Accept the peer certificate without checking it. This is useful if you want to check the certificate manually (see <code>GetProp</code> ).
64	<code>RequestClientCertificate</code>	Only valid for a server, this asks the client for a certificate, but will still allow connections if the client does not provide one.
128	<code>RequireClientCertificate</code>	Only valid for a server, this asks the client for a certificate and refuses the connection if a valid certificate (subject to any other flags) is not provided by the client.

## Appendix D: Error Codes

### TLS Related error codes

**Purpose** This table lists the TLS specific error messages that can be returned when dealing with TLS connections.

<i>Code</i>	<i>Message</i>	<i>Description</i>
1201	ERR_TLSHANDSHAKE ERR_SECSOCK	The handshake process sets up a secure connection between the client and server before the certificates are exchanged. This error indicates that process is failing. The most likely causes are that the systems can't agree an encryption protocol, or that you are connecting to a server/client that is not using SSL/TLS.
1202	ERR_INVALID_PEER_CERTIFICATE	The certificate supplied by the peer is not valid. If you would like to make the connection anyway and examine the certificate manually please supply the CertAcceptWithoutValidating flag to <code>DRC.Svr</code> or <code>DRC.Clt</code> . Once the connection has been made you can examine the peer certificate by calling <code>GetProp</code>
1203	ERR_COULD_NOT_LOAD_CERTIFICATE_FILES	One or more of the specified certificate files could not be loaded. Either the file specified does not exist, can't be read or is not a valid PEM or DER file. Please check the filenames that are being passed to the <code>DRC.Svr</code> & <code>DRC.Clt</code> functions.
1204	ERR_INITIALISING_TLS	There was an error setting up the TLS libraries. This could be due to missing or invalid GnutTLS files.
1205	ERR_NO_SSL_SUPPORT	There is no SSL support in the available conga library. Please contact Dyalog support for a version with SSL support.

## Appendix E: Upgrading to Conga v2.2

Conga version 2.2 is released coincident with Dyalog APL version 13.1. If you have existing applications using Conga you have a few options.

### **Use your existing Conga v2.0 or v2.1 application with Dyalog APL v13.1 or later.**

If you choose to continue to use Conga v2.0 or v2.1, you will need to copy the their.dll files (under Windows) or .so files (under Unix variants) into the directory where Dyalog v13.1 is installed.

### **Upgrade your existing Conga v2.0/2.1 application to use Conga v2.2 with Dyalog APL v13.1 or later.**

If you're not using secure communications, you can simply replace the elements of Conga v2.0/2.1 in your application with their v2.2 counterparts.

If you are using secure communications with Conga v2.0, you will need to modify how certificate information is used. The affected functions are `DRC.Cl t`, `DRC.Srv`, `DRC.GetProp`, `WebServer.HttpsRun`, and `Samples.HTTPGet`.

### **Use Conga v2.2 with an earlier version 11, 12.0, or 12.1 of Dyalog APL.**

You will need to copy the Conga v2.2 .dll files (under Windows) or .so files (under Unix variants) into the directory where Dyalog APL is installed. You should then be able to use the functions in Conga v2.2.

### **Still using Conga v1.0 and want use Dyalog APL v13.1 or later?**

If you have an application using Conga v1.0 and want to use Dyalog APL v13.1 or later, you can:

- Copy the DRC namespace into your application and modify your code to use `DRC.Cl t` and/or `DRC.Srv`. This will, in effect, convert your application to Conga v2.1.
- Copy the DRC namespace into your application and use the cover functions provided in the `DRCv1` namespace. You should be able to use your existing application code, though it is strongly recommended that you upgrade to use the newer syntax in Conga v2.1 and later.

# Appendix F: Change History

This appendix contains the documented changes from prior versions of Conga.

## Version 2.0

Conga version 2.0 is the first major update to Conga. In addition to new features, some “tidying up” has been performed, to simplify applications built on Conga. Note that this means that *minor modifications may need to be made to existing applications* in order to switch to Conga v2.

### v2.0 New Functionality

- **IPv6:** Support for the IPv6 protocol, which will replace the existing “IPv4” in the years to come, as the world runs out of IP addresses. A property of the Root object defines the default protocol, but the protocol can also be selected for each Client or Server object.
- **Named Parameters:** Functions for creating client and server objects now support named arguments (previously, all arguments were “positional”).
- **Select Server IP Address:** When you create a Server on a machine with several addresses, you can specify which address it should listen on. Previously, Conga servers would accept connections to any address that the machine had.
- **Improved Scheduling:** A flaw in the scheduling algorithm has been fixed, avoiding problems which would favour certain connections and potentially “starve” others.

### v2.0 Changed Functionality

- **New Srv and Clt functions:** The functions `SecureServer` and `Server` have been merged into a function called `Srv`. `SecureClient` and `Client` have become `Clt`. A set of cover functions which implement the old functionality are supplied in a namespace called `DRCv1`, but it is recommended that you migrate away from the old functions at your earliest convenience.
- **Received event removed:** The `Received` event, which was issued when that a message was received AND the connection had also been closed, has been removed. In version 2.0, a `Receive` event is issued regardless of whether the connection is still open.
- **Peer Address removed from Connection Names:** Incoming connections no longer have names which contain an encoding of the peer address and port number. New properties called `LocalAddr` and `PeerAddr` allow you to retrieve the addresses of both ends of a connection.
- **Stop Criteria changed:** The format for specifying Stop Criteria has been changed. A new property `EOM` allows you to set one or more termination strings, which will trigger new `Receive` events. Messages are always cut at the END of a

termination string. A property called `IgnoreCase` controls how the comparison is done.

- **Enhanced HTTPGet sample:** The function `Samples.HTTPGet` has been enhanced to handle secure connections (`HTTPSGet` has been withdrawn), basic authentication and automatic UTF-8 conversion.

## Version 2.1

Conga version 2.1 is an incremental update to Conga version 2.0. While released as a companion to Dyalog APL version 13.0, it can be used with earlier versions of Dyalog APL as well. The primary feature in this update modifies how certificates are used to facilitate secure communications. This has resulted in changes to the functions `DRC.Srv` and `DRC.Clt` when using certificates. *Existing Conga v2.0 applications that use certificates will require minor modification to use Conga v2.1.*

### New Functionality

- **DRC.X509Cert Class:** This class encapsulates the structure and function necessary to use X.509 certificates with Conga. This is the recommended method for providing certificate information to `DRC.Clt` and `DRC.Srv`.
- **DRC.Certs function:** Provides the underlying functionality used by `DRC.X509Cert` to read and decode certificates.
- **Microsoft Certificate Store support:** Certificates located in Certificate Stores under Windows can now be read and used with Conga.

### Changed Functionality

- **DRC.Srv and DRC.Clt functions:** The syntax to pass certificate information for both functions has changed.
- **DRC.Init function:** Now accepts a left argument of `-1` to cause Conga to reload its underlying drivers.
- **Samples.HTTPGet function:** The optional left hand argument is now an `X509Cert` object.
- **WebServer.HttpsRun function:** Now accepts an `X509Cert` object argument.
- **DRC.GetProp:** Now returns an `X509Cert` object for `'PeerCert'`