Conga
User Guide

Conga version 3.7



DYALOG

The tool of thought for software solutions

Dyalog is a trademark of Dyalog Limited
Copyright © 1982-2026 by Dyalog Limited
All rights reserved.

Conga User Guide

Conga version 3.7
Document Revision: 20260907_370

Unless stated otherwise, all examples in this document assume that `⎕IO ⎕ML ← 1`

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.

email: support@dyalog.com
https://www.dyalog.com

TRADEMARKS:

Raspberry Pi is a trademark of the Raspberry Pi Foundation.
Oracle® , MySQL, and Java™ are registered trademarks of Oracle and/or its affiliates.
JavaScript™ is a trademark of Oracle Corporation.
Unicode is a registered trademarks of Unicode, Inc. in the United States and other countries.
UNIX® is a registered trademark in the U.S. and other countries, licensed exclusively through X/Open Company Limited.
Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.
Windows® is a registered trademark of Microsoft Corporation in the U.S. and other countries.
macOS® and OS X® (operating system software) are registered trademarks of Apple Inc. in the U.S. and other countries.

All other trademarks and copyrights are acknowledged.

Contents

1 About This Document **1**
1.1 Audience 1
1.2 Conventions 1
2 Introduction **3**
3 Installation **5**
3.1 Compatibility 5
3.1.1 Compatibility with Applications Using Conga Version 2.x 5
3.2 Initialisation 6
4 Getting Started **8**
4.1 Conga Objects 8
4.1.1 Conga Object Types 8
4.1.2 Conga Object States 10
4.1.3 Conga Object Modes 13
4.2 Root Objects 16
4.3 A Simple Conga Client 17
4.4 A Simple Conga Server 19
4.5 Command Mode 20
4.6 Parallel Commands 22
4.6.1 Multi-threading 23
5 Secure Connections **25**
5.1 CA Certificates 26
5.2 Client and Server Certificates 27
5.2.1 Certificate Stores 28
5.2.2 Revocation Lists 28
5.3 Creating a Secure Client 28
5.3.1 Upgrading to a Secure Client 31
5.4 Creating a Secure Server 31
5.4.1 Upgrading to a Secure Connection 32
5.5 Using the DRC.X509Cert Class 33
5.5.1 Certificate Chains 34
6 HTTP Mode **36**
6.1 Receiving HTTP Messages 36
6.2 HTTP Mode Events 37
6.2.1 Event: HTTPHeader 38
6.2.2 Event: HTTPBody 39
6.2.3 Event: HTTPChunk 39
6.2.4 Event: HTTPTrailer 39
6.2.5 Event: HTTPFail 40
6.2.6 Event: HTTPError 40

6.3 Limiting HTTP Message Size 40
6.4 Sending HTTP Messages 41
6.5 WebSocket Protocol 42
6.5.1 Client-side WebSocket Upgrade 42
6.5.2 Server-side WebSocket Upgrade 46
6.5.3 Transmitting WebSocket Data 47
6.5.4 Receiving WebSocket Data 48
6.5.5 Secure WebSockets 48
7 The Conga Workspace 50
8 Utilities and Samples 51
8.1 Utilities 51
8.2 Samples 51
9 Advanced Usage 53
9.1 ConnectionOnly Property 53
9.2 Sending Files 53
9.3 Compression Level 55
9.4 Temporarily Prevent New Connections 56
9.5 Allow/Deny Connections from Specific Address Ranges 56
9.6 Timeout and Closed Events 57
9.7 Sent Event 58
9.8 Options Parameter 58
9.9 Magic Parameter 60
9.10 HTTP/1.1 Tunnelling 61
A Technical Reference 63
A.1 Return Codes 63
A.2 Conga Object Properties 64
A.3 Function: Conga.Init 73
A.4 Function: Conga.Magic 75
A.5 Function: Conga.New 75
A.6 Function: Conga.RootNames 76
A.7 Function: DRC.Certs 77
A.8 Function: DRC.ClientAuth 78
A.9 Function: DRC.Close 79
A.10 Function: DRC.Clt 79
A.11 Function: DRC.DecodeOptions 82
A.12 Function: DRC.Describe 82
A.13 Function: DRC.Error 83
A.14 Function: DRC.Exists 84
A.15 Function: DRC.GetProp 84
A.16 Function: DRC.Init 85
A.17 Function: DRC.Names 86
A.18 Namespace: DRC.Options 87
A.19 Function: DRC.Progress 87
A.20 Function: DRC.ReadCertFromFile 88

A.21 Function: DRC.ReadCertFromFolder ... 89
A.22 Function: DRC.ReadCertFromStore ... 89
A.23 Function: DRC.ReadCertUrls ... 90
A.24 Function: DRC.Respond ... 91
A.25 Function: DRC.RootName ... 91
A.26 Function: DRC.Send ... 92
A.27 Function: DRC.ServerAuth ... 97
A.28 Function: DRC.SetProp ... 97
A.29 Function: DRC.Srv ... 98
A.30 Function: DRC.Tree ... 101
A.31 Function: DRC.Version ... 103
A.32 Function: DRC.Wait ... 103
A.33 Class: DRC.X509Cert ... 106
A.33.1 Instances of the DRC.X509Cert Class ... 108
B Certificates ... 111
B.1 PEM File Format ... 111
B.2 Generating Certificates and Keys ... 112
C TLS Flags ... 116
D Conga Libraries ... 118
E Error Codes ... 119
F Change History ... 122
F.1 Version 3.7 ... 122
F.2 Version 3.6 ... 122
F.3 Version 3.5 ... 122
F.4 Version 3.4 ... 123
F.5 Version 3.3 ... 123
F.6 Version 3.2 ... 124
F.7 Version 3.1 ... 124
F.8 Version 3.0 ... 124
F.9 Version 2.7 ... 126
F.10 Version 2.6 ... 126
F.11 Version 2.5 ... 127
F.12 Version 2.4 ... 127
F.13 Version 2.3 ... 127
F.14 Version 2.2 ... 128
F.15 Version 2.1 ... 128
Index ... 130

1 About This Document

This document is a complete guide to Conga, Dyalog's framework for TCP/IP communications. It describes the tools with which Conga can be used to create a variety of clients and servers using protocols based on TCP/IP, including HTTP, HTTPS, FTP, Telnet and SMTP. It covers Conga support for secure communications (using SSL/TLS) and communication between APL processes (allowing them to exchange native APL data directly). It also introduces the Conga workspace, various samples/utilities and contains a technical reference of the namespaces, classes and functions provided with Conga.

1.1 Audience

It is assumed that the reader has a reasonable understanding of Dyalog and server/client connection protocols; a working knowledge of HTTP/FTP/SMTP is needed to understand the samples provided.

For information on the resources available to help develop your Dyalog knowledge, see https://www.dyalog.com/introduction.htm.

1.2 Conventions

Unless explicitly stated otherwise, all examples in Dyalog documentation assume that ⎕IO and ⎕ML are both 1.

Various icons are used in this document to emphasise specific material.

General note icons, and the type of material that they are used to emphasise, include:


Hints, tips, best practice and recommendations from Dyalog Ltd.


Material of particular significance or relevance.



Legacy information pertaining to behaviour in earlier releases of Dyalog or to functionality that still exists but has been superseded and is no longer recommended.



Warnings about actions that can impact the behaviour of Dyalog or have unforeseen consequences.

A full list of the platforms on which Dyalog version 21.0 is supported is available at https://www.dyalog.com/dyalog/current-platforms.htm. Although the Dyalog programming language is identical on all platforms, differences do exist in the way some functionality is implemented and in the tools and interfaces that are available. Differences in behaviour between operating systems are identified with the following icons (representing macOS, Linux, Microsoft Windows and UNIX respectively):



2 Introduction

Conga (also known as the Dyalog Remote Communicator) is a tool for communication between applications. It can transmit APL arrays between two Dyalog applications that both use Conga and it can exchange messages with many other applications, for example, HTTP servers (also known as web servers), web browsers and other web clients/servers including Telnet, SMTP and POP3.

Uses of Conga include, but are not limited to:

- retrieving information from – or uploading 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 intranet or the internet.
- implementing APL Remote Procedure Call (RPC) servers; these receive APL arrays from client applications, process data and return APL arrays as the result.

Conga supports secure communication using TLS (Transport Layer Security), which is the successor to SSL (Secure Sockets Layer). Conga makes it easy for APL developers to embed client or server components in APL applications and simplifies the process of making remote calls in a multi-threaded client environment.

Although Conga currently only uses the TCP protocol, other communication mechanisms could be added in the future. Conga hides many of the details of TCP socket handling and notifies the application of incoming data, connection events and errors so that all the application needs to do is handle the data that arrives. Dyalog Ltd recommends Conga as the mechanism for handling TCP-based communications in preference to the now outdated TCPSocket object.

Conga is used in many Dyalog tools including MiServer (Dyalog's APL-based web server), Jarvis (the stand-alone web service framework), HTTPCommand, the Dyalog File Server, and isolates.



If you redistribute code that uses Conga, please see the *Licences for third-party components* document in the **[DYALOG]/Help** directory of your installation.

3 Installation

Conga is implemented as a library. The library is loaded and accessed through by a set of API functions.

Conga is installed with Dyalog and no additional installation is required.

3.1 Compatibility

All versions of Conga are:

- compatible with all supported Dyalog versions.
- compatible with each other.

When the server and client are both Conga objects, they:

- do not have to be running the same version of Conga.
- do not have to be running the same version of Dyalog.

However, standard Dyalog interoperability rules apply – any specific functionality that is required (in Conga or Dyalog) must be available at both ends of the connection. For details of the changes made in each Conga version, see *Appendix F*.

For compression to work, the client and server both need to support the same level of compression (see *Section 9.3*).

3.1.1 Compatibility with Applications Using Conga Version 2.x

Conga version 2.x was initialised using the `DRC` namespace – this contains all of the API functions that are necessary to access the Conga library. Conga version 3.x takes a different approach and is initialised by creating an instance of the `Conga.LIB` class. This provides APIs that are syntactically identical to those in the `DRC` namespace and enables multiple Conga Root objects to be run.

For backwards compatibility, the DRC namespace is still included in the conga workspace – this means that existing applications will continue to work without requiring any modification. However, Dyalog Ltd recommends that you update any legacy applications to use the `Conga.LIB` approach and that any new applications also use this approach.

3.2 Initialisation

Before using any of the Conga API functions, the system needs to be initialised by loading the library.

To initialise the system

1. Access the `Conga` namespace in the appropriate way:
   - If you are experimenting with functionality before adding Conga to an application, then load the `conga` workspace:

     ```
           )LOAD conga
     ...\ws\conga.dws saved Fri Feb 22 17:21:04 2019
     ```

   - If you are writing an application whose behaviour should remain unchanged irrespective of future changes to Conga, then copy the `Conga` namespace from the `conga` workspace into your application's workspace and then save the application workspace. This is a common scenario when writing a stand-alone application that will be packaged and distributed:

     ```
           'Conga' ⎕CY 'conga'
     ```

   - If you are writing an application and want the current version of Conga to be loaded dynamically, include the following statement in your application's start-up code (this is a common scenario when writing an application that acts as a server):

     ```
           'Conga' ⎕CY 'conga'
     ```

     If you intend to ship an application that includes Conga, then the relevant libraries will also need to be shipped. For more information, see *Appendix D*.

2. Initialise an instance of the Conga root object:

   ```
   DRC←Conga.Init ''
   ```

This connects to the Conga root object named DEFAULT, which is shared by all applications that use an empty right argument to `Conga.Init`. By naming the instantiation DRC, existing application code that refers to the DRC namespace should continue to work without modification.



If you have a Conga version 2.x application, then replacing `DRC.Init ''` with `DRC←Conga.Init ''` and ensuring that the `Conga` namespace rather than the DRC namespace is loaded into your workspace, means that the Conga version 2.x application should run unchanged.

The examples throughout this document assume that the system has already been initialised, that is, the above steps have already been followed.



Throughout this document, unless explicitly noted otherwise, references to DRC represent a Conga library instance initialised using `DRC←Conga.Init ''` rather than the legacy DRC namespace.

4 Getting Started

This chapter introduces Conga client and server objects and demonstrates their use through simple examples.

The purpose and syntax of the functions and methods used in this chapter are detailed in *Appendix A*.

4.1 Conga Objects

A *Conga object* is a named object created inside Conga but outside the workspace. Each Conga object has specific properties that can be queried and/or set (some properties are read-only).

Conga object names cannot exceed 32 characters in length and must not contain null characters.

The Conga root object is referred to as '`.`'.

A name can be assigned to Conga server and client objects when they are created using the `DRC.Srv` and `DRV.Clt` functions respectively; if an empty vector is specified (`''`) then Conga will generate a name. Subsequent uses of the server or client objects (for example, to send a message using `DRC.Send` or listen for messages using `DRC.Wait`) require their name.

Each Conga object has a type, state and mode.

4.1.1 Conga Object Types

There are six possible *types* of Conga object – these are listed with their identifying object type code in *Table 4-1* (object type codes are used by several of the functions in Conga – see *Appendix A*).

Table 4-1: *Conga object types*

Code	Object Type	Description
0	Root	The highest level object. The root object contains information about the Conga installation; it is created by the `DRC.Init` or `Conga.Init` function.
1	Server*	A server object listens for connections from clients – the client can be any TCP/IP client and does not have to be a Conga object. It also receives, processes and responds to requests from connections.
2	Client*	A client object connects to a server, sends requests to it and receives responses from it; the server can be any TCP/IP server and does not have to be a Conga object.
3	Connection	A server object can respond to multiple client connections; a connection object maintains information pertaining to each client connection.
4	Command	A command object represents an individual request (from a client) or response (from a server). Command objects only exists for servers and clients in *Command* mode (see *Section 4.1.3*).
5	Message	Message objects are created by servers using the `DRC.Progress` function and sent to client objects. Message objects only exist for servers and clients in *Command* mode (see *Section 4.1.3*).

* A client is said to be *using Conga* if it is communicating through a client instance that was set up using the `DRC.Clt` function (see *Section A.10*); a server is said to be using Conga if it is communicating through a server instance that was set up using the `DRC.Srv` function (see *Section A.29*). The other end of the connection can also be using Conga or it can be a non-Conga object that understands TCP/IP (for example, a web browser or web server).

The ERD (Entity-Relationship Diagram) in *Figure 4-1* shows how servers and clients are related to each other and to other Conga objects. At least one of the server side or client side must be using Conga.

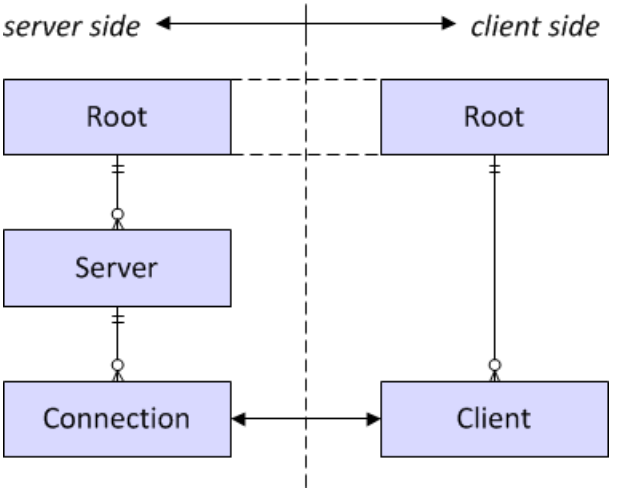


Figure 4-1: *Conga object ERD (standard crow's foot notation)*

Communication between a server's connection object and a client object depend on the connection mode (see *Section 4.1.3*).

The ERD in *Figure 4-2* shows how the remaining two Conga object types relate to the connection and client objects – this is only valid when both the server and client are in *Command* mode. The sequence of events starts when the client sends a command through the connection to the server; optionally, the server sends messages through the connection to the client before sending a final response to the command.

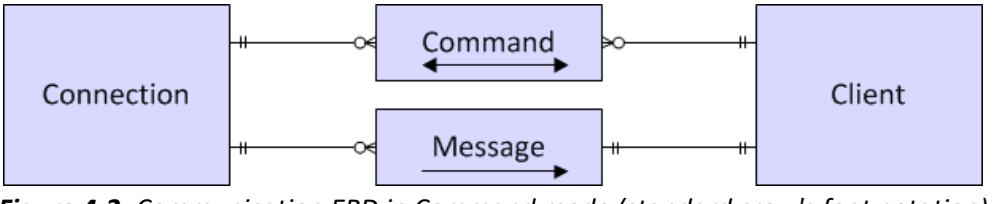


Figure 4-2: *Communication ERD in Command mode (standard crow's foot notation)*

For more information on connection modes, see *Section 4.1.3* and *Section 4.5*.

4.1.2 Conga Object States

Each Conga object has a *state*, that is, the temporary condition in which it exists as it progresses through its cycle from creation to deletion. The possible states and the Conga object types (*Section 4.1.1*) that can exist in each of these states are detailed in *Table 4-2*.

Table 4-2: *Conga object states and the object types that can exist in those states*

Code	Object State	Conga Object Types						Description
		0	1	2	3	4	5	
0	New	x	x	x	x	x	x	Transient state that the object exists in after it is created but before it has been initialised.
1	Incoming				x			A connection has been established but a `Connect` event has not yet been received.
2	RootInit	x						The root object exists and is connected to the Conga library (Microsoft Windows DLL or UNIX/Linux Shared Library).
3	Listen		x					The server is listening for incoming connection attempts.
4	Connected			x	x			The client/connection is connected to its connection/client peer.
5	APL					x	x	The thread that handles socket communications has a full buffer and no further processing can occur until the application calls the `DRC.Wait` function.

Table 4-2: *Conga object states and the object types that can exist in those states (continued)*

Code	Object State	Conga Object Types						Description
		0	1	2	3	4	5	
6	ReadyToSend					x	x	Data is ready to be sent.
7	Sending					x	x	Sending data.
8	Processing					x		The command has been passed to the server but no response has been issued.
9	ReadyToRecv					x	x	Waiting for data.
10	Receiving					x	x	Receiving data.
11	Finished	x	x	x	x	x	x	All data has been transmitted/received, connections closed and commands finished.
12	MarkedForDeletion	x	x	x	x	x	x	The Conga object is ready for deletion.
13	Error		x	x	x	x	x	An error has occurred.
14	*internal*	-	-	-	-	-	-	*internal*
15	*internal*	-	-	-	-	-	-	*internal*
16	SocketClosed			x	x			The socket has been closed.

Table 4-2: *Conga object states and the object types that can exist in those states (continued)*

Code	Object State	Conga Object Types						Description
		0	1	2	3	4	5	
17	APLLast			x	x			The connection has been closed but uncollected data still exists in the thread that handles socket communications.
18	SSL			x	x			The client/connection is negotiating an SSL connection with its connection/client peer.

4.1.3 Conga Object Modes

Conga clients and servers support six different modes for connection, that is, formats in which data can be transmitted:

- *Text*

  Allows transmission of character strings. Character strings can only comprise characters with Unicode code points less than 256. To transmit characters outside this range, Dyalog Ltd recommends that you either use UTF-8 character encoding (for information on this, see `⎕UCS` in the *Dyalog APL Language Reference Guide*) or switch to *Raw* mode and convert the character string to the appropriate format (for example, by applying `⎕UCS`).

- *BlkText*

  As *Text* mode, but each data transmission is considered as a block. If a block exceeds the maximum size (as defined by the `BufferSize` parameter of the `DRC.Clt`/`DRC.Srv` function) then the peer object receiving the transmission will return error 1135 (buffersize exceeded). Each block includes a header stating the:

  - block length: 32-bit integer giving the exact length of the block. Determined by the network/operating system.

- magic number (optional): 32-bit integer used to check that the data has not been corrupted. Set using the `DRC.Clt/DRC.Srv` function (the `Magic` parameter).

Each block is assigned the event name `Block`. If the connection closes before all blocks have been processed, then the final block tobe processed is assigned the event name `BlockLast`. The third element returned by the `DRC.Wait` function indicates the event name (see *Section A.32*).

Unless explicitly specified otherwise, information about *Text* mode can be assumed to apply to *BlkText* mode too. Only valid when both the server and client are using Conga.

- *Raw*

  Similar to *Text* mode, 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. Single-byte characters (type 80 or 82) can also be sent

- *BlkRaw*

  As *Raw* mode, but each data transmission is considered as a block. If a block exceeds the maximum size (as defined by the `BufferSize` parameter of the `DRC.Clt/DRC.Srv` function) then the peer object receiving the transmission can reject it. Each block includes a header stating the:

  - block length: 32-bit integer giving the exact length of the block. Determined by the network/operating system.
  - magic number (optional): 32-bit integer used to check that the data has not been corrupted. Set using the `DRC.Clt/DRC.Srv` function (the `Magic` parameter).

  Each block is assigned the event name `Block`. If the connection closes before all blocks have been processed, then the final block to be processed is assigned the event name `BlockLast`. The third element returned by the `DRC.Wait` function indicates the event name (see *Section A.32*).

  Unless explicitly specified otherwise, information about *Raw* mode can be assumed to apply to *BlkRaw* mode too. Only valid when both the server and client are using Conga.

- *Command*

  Each transmission is a complete APL object in a binary format. This is the default mode. Only valid when both the server and client are using Conga. For more information on *Command* mode, see *Section 4.5*.

- *HTTP*

  The standard `Receive` and `Block` events are replaced with events that signal the arrival of a complete piece of HTTP protocol, that is, `HTTPHeader`, `HTTPBody`, `HTTPChunk` and `HTTPTrailer`. This simplifies the task of receiving HTTP data and significantly improves performance. For more information, see *Chapter 6*.

A client and server can only exchange data if they are running in compatible modes. This means that:

- a *Command* mode client must be connected to a *Command* mode server. To use *Command* mode, both the client and server need to be using Conga.
- a *Text* mode or *Raw* mode client must be connected to a *Text* mode or *Raw* mode server; *BlkText* mode and *BlkRaw* mode are interchangeable with *Text* mode and *Raw* mode respectively, as long as the requisite header (containing the field length and magic number) is added to the message being sent. *Text* mode and *Raw* mode are typically used when only one end of the connection is using Conga.
- an *HTTP* mode client must be connected to an *HTTP* mode server. Either the client, the server, or both could be running Conga in *HTTP* mode. For example, any HTTP client (for example, as a web browser) can connect to an *HTTP*-mode Conga server, and an *HTTP*-mode Conga client can connect to any HTTP server (for example, a web server or web service server).

Command mode is the optimal way for APL clients and servers to communicate with each other, because:

- the internal representation is the binary format used by APL; this is more compact than a textual representation.
- numbers can be transmitted without having to be formatted and interpreted.
- no buffer size needs to be declared.

In *Command* mode, *BlkText* mode or *BlkRaw* mode, each transmission comprises an entire APL array or block of data; the `DRC.Wait` function does not report incoming data until the entire APL array or block of data has arrived. In *Text* mode and *Raw* mode, 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;

the `DRC.Wait` function reports incoming data each time a TCP packet arrives or when the receive buffer is full. The recipient may need to buffer incoming data in the workspace and analyse it to determine whether a complete message has arrived.

In *Text* and *Raw* modes, an EOM termination string can be set. In this situation, the `DRC.Wait` function terminates on receipt of the specified termination string. If an empty termination string is specified, then the `DRC.Wait` function terminates when the buffer contains `BufferSize` bytes (see *Section A.10* and *Section A.29*). If an EOM termination string is not specified, then the `DRC.Wait` function returns data each time a TCP packet is received. If a TCP packet is larger than `BufferSize` bytes then the data is returned in blocks of `BufferSize` bytes.

4.2 Root Objects

When an application and the tools used to maintain it both use TCP communications, they need to be independent so that each can be restarted/reset without interfering with the others. However, it is increasingly common that more than one component in an application uses Conga, leading to potential name conflicts between client/server objects and clashes between different state settings.

If a component initialises Conga using the `Conga.Init` function, then it has its own *Root object* under which Conga objects are created and operations are performed. This means that existing Conga connections can be disconnected and restarted without interfering with other components.



If Conga version 2.x is initialised using the `DRC.Init` function, then all components in the same process use the same `DRC` namespace. The first use calls the `DRC.Init` function and receives a clean return code, and subsequent calls to the `DRC.Init` function warn that Conga is already initialised. If a component resets or re-initialises Conga, then all other components are impacted. Although the `DRC` namespace is still provided for backwards compatibility, Dyalog Ltd recommends that you migrate existing applications to use the `Conga.Init` function (see *Section 3.2*).

Although you can use an empty right argument to the `Conga.Init` function to connect to the default root, for an application that might need to manage the state of Conga, Dyalog Ltd recommends initialising Conga using a right argument to identify your application. For example:

```
DRC←Conga.Init 'MyApp'
```

This statement can safely be called anywhere in an application – if a root with that name already exists, then a reference will be returned to the existing root instance.

To ensure that a new instance is created, use the `Conga.New` function. If this is called with an empty argument then a new unused root name is generated; if a non-empty argument is supplied then an error will be generated if the root name is already in use. The `Conga.RootNames` functions returns a list of all existing roots:

```
      DRC←Conga.Init ''  ⍝ Use the default root
      DRC1←Conga.New ''  ⍝ Create new root with generated name
      Conga.RootNames
DEFAULT  IC1
```

To re-initialise a root, erase the reference (all references) to the instance; it will be cleaned up, and you can create a new one.



The intention is not that you create a large number of roots (the process of creating and tearing down roots is expensive and complex.) Components might need a separate root, but you should not create new roots to, for example, create queries on the internet – in this situation, use the default root that you can get a reference to by passing an empty argument to `Conga.Init`.

4.3 A Simple Conga Client

A Conga *client* establishes contact with a service that is already running and listening on a pre-determined port at a known TCP address. The service could be an APL application that has created a Conga server or it could be any application or service that provides services through TCP sockets. For example, most UNIX systems (and many Microsoft Windows servers) provide a set of simple services like a Time of Day (TOD) service or a 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 function `DRC.Clt` can be used to create a Conga client. In the following example, this function is called with five elements in its right argument:

- the name to be used for 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 TOD service)
- the type of socket (`Text`)
- the size (in bytes) of the buffer that should be created to receive data (`1000`)



For this example to generate the error documented, run it on the Microsoft Windows operating system with the TOD service disabled (this can be done by going to **Control Panel > Programs and Features > Turn Windows features on or off** and unselecting the **Simple TCPIP services (i.e. echo, daytime etc)** checkbox, then rebooting).

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

The error message that is generated follows the syntax for all error codes generated by Conga functions (see *Section A.1*), that is, it is a vector in which:

- `[1]` is a return code (see *Section A.1*)
- `[2]` is the error name
- `[3]` is, optionally, additional information about the error.

This issue can be resolved in any of the following ways:

- Enable the service on localhost. This can be done by going to **Control Panel > Programs and Features > Turn Windows features on or off** and selecting the **Simple TCPIP services (i.e. echo, daytime etc)** checkbox. Reboot, then rerun the `DRC.Clt` function call:

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

  The result code of zero indicates that the client was successfully created (any other code indicates failure).

- Call a different server machine that does provide a TOD service, for example:

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

  The result code of zero indicates that the client was successfully created (any other code indicates failure).

After the client object has been successfully created, incoming data can be received. Receiving data from the server involves calling the `DRC.Wait` function with the name of the client. For example:

```
      ]DISP DRC.Wait 'C1'
┌→┬──┬─────┬──────────────────────┐
│0│C1│Block│15:01:44 07/10/2015   │
└─┴──┴─────┴──────────────────────┘
```

The returned message is a vector in which:

- `[1]` is the return code
- `[2]` is the object name
- `[3]` is the type of event (see *A.32*)
- `[4]` is data associated with the event

The client object can now be closed (good practice):

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

4.4 A Simple Conga Server

The TOD service referred to in *Section 4.3* is a very simple server and can be implemented by calling the `TODServer.RunText` function in the `TODServer` namespace (loaded from the **[DYALOG]/Samples/Conga/TODServer** directory) to create a server object. The `TODServer.RunText` function enters a loop where it waits for connections; this means that, to be able to experiment with using this service without starting a second APL session, it should be started using the *Spawn* operator (`&`) so that it runs in a separate thread:

```
      ]Load [DYALOG]Samples/Conga/TODServer/TODServer
      TODServer.RunText & 13
Text Mode TOD Server started on port 13
```

OS WIN For this to work on the Microsoft Windows operating system, the TOD service must be disabled (this can be done by going to **Control Panel > Programs and Features > Turn Windows features on or off** and unselecting the **Simple TCPIP services (i.e. echo, daytime etc)** checkbox, then rebooting).

The right argument in this function call is the port number; if a TOD service is already running on port 13, then an error message is returned and a different port must be used for the new service.

A client object can now be created, data received and the client object closed:

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

      DRC.Wait 'C1'
0  C1  Block  10:09:03 12-10-2015

      DRC.Close 'C1'
0
```

The TOD server created by calling the `TODServer.RunText` function is not restricted to only respond to Dyalog applications using Conga – it can be used by any program that is written to use a TOD service.

The server can be stopped as follows:

```
      TODServer.DONE←1
TOD Server terminated.
```

4.5 Command Mode

Section 4.3 and *Section 4.4* used connections in *Text* mode, which are appropriate for most web applications. Even when remote procedure calls are made over the internet, with arguments and results containing arguments that are not simply text strings, the parameters are usually encoded using SOAP/XML, which is a text-based encoding.

The TOD server created by calling the `TODServer.RunText` function in *Section 4.4* can be used by any program that is written to use a TOD service. It could be restricted to only respond to Dyalog applications using Conga by converting it to use *Command* mode – doing this means that it will return the time as a 7-element array in `⎕TS` format.

The `TODServer.RunCommand` function implements a TOD service running in *Command* mode. Unlike the *Text* mode TOD service, a server that is running in *Command* mode cannot initiate the transmission of data when the connection is made, but can only respond to a request from a client. This means that the `:Case 'Connect'` statement does not have any associated code. However, code could be added here so that the TOD server could (for example) record connections. The code handling the `:Case 'Receive'` event does no processing on any content received with the request, it merely returns the current timestamp irrespective of the content.

In *Text*/*Raw* mode, client and server can both initiate data transfer by calling the `DRC.Send` function (see *Section A.26*) – there is no concept of a request/respond protocol at the Conga level, although implementing an HTTP protocol over the connection can add such a protocol at the application level. However, in *Command* mode (and in *BlkRaw* mode and *BlkText* mode), Conga has an in-built protocol; communication on a connection is synchronous and consists of discrete commands. Each command comprises a request from the client followed by a response from the server; the server cannot initiate an unrequested transfer of data. The request message from the client and response message from the server are linked by an identifier (this is not the case in other modes). This means that, although the

`DRC.Send` function can be used to send data from a client in *Command* mode, a different function must be used to send data from a server in *Command* mode – the `DRC.Respond` function (see *Section A.24*). The server can also call the `DRC.Progress` function (see *Section A.19*); this sends progress messages to the client while the server is processing a command, allowing the client to show the user a progress bar or other status information.

Ideally, the *Command* mode TOD server should be started on a port other than port 13 so that it is not confused with a standard TOD server (if required, both the *Text* and *Command* TOD servers could be run at the same time, in different threads):

```
      TODServer.RunCommand & 913
Command Mode TOD Server started on port 913
```

A *Command* mode Dyalog client of this TOD server can now be created and retrieve a numeric timestamp from the server:

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

      DRC.Send 'C1' ''
0  C1.Auto00000000
```

The first element of the argument to the `DRC.Send` function can be either a client name or a connection name:

- if a client name is supplied (as in this example) then Conga generates a connection name and returns it as result element `[2]` in the format `clientname.connectionname` (in this example, `C1.Auto00000000`).
- if a connection name is supplied, then Conga returns it as result element `[2]`.

```
      DRC.Wait 'C1'
0  C1.Auto00000000  Receive  2015 10 19 9 36 48 845
```

Result element `[4]` is now a 7-element integer vector rather than a formatted timestamp; this is more performant on an APL client, but means that the TOD server is no longer usable by other TCP client programs that expect a *Text* mode TOD server.

Unlike the *Text* mode TOD server, the *Command* mode TOD server does not close the connection after sending a timestamp. This means that a second timestamp can be retrieved from the server (in this example the `DRC.Wait` function includes a maximum waiting time of 5 seconds):

```
      DRC.Send 'C1' ''
0  C1.Auto00000001

      DRC.Wait 'C1' 5000
0  C1.Auto00000001  Receive  2015 10 19 9 37 28 581
```

4.6 Parallel Commands

Although the *Command* mode protocol is synchronous, more than one command can be active at the same time – it is not necessary to wait for the response to one command before the next command is sent. In addition, multiple commands can be started and the results retrieved in any order.

EXAMPLE

(In this example command names are specified, whereas in *Section 4.5* the command name was auto-generated)

```
      DRC.Send 'C1.TS1' ''
0  C1.TS1

      DRC.Send 'C1.TS2' ''
0  C1.TS2

      DRC.Wait 'C1.TS2' 1000
0  C1.TS2  Receive  2015 10 19 9 38 7 957

      DRC.Wait 'C1.TS1' 1000
0  C1.TS1  Receive  2015 10 19 9 37 57 965
```

The timestamps show that the `TS1` command was executed before the `TS2` command even though the results were retrieved in the reverse order.

The *Command* mode protocol allows multiple APL threads to work independently. A request message from the client and the associated response message from the server are linked by a command name; this means that any APL thread can wait for the result of a command, as long as it knows the command name. Multiple APL threads can share the same server connection; one APL thread can send a command and then dispatch a new APL thread to wait for and process the result of that command.

Command names can be reused as soon as the result has been received (but not before).

EXAMPLE

Using client `C1` and the modified TOD server created in *Section 4.5*:

```
      DRC.Send 'C1.TS1' ''
0  C1.TS1

      DRC.Send 'C1.TS2' ''
0  C1.TS2

      {⎕TID,DRC.Wait ⍵ 1000}&¨ 'C1.TS1' 'C1.TS2'
2 0  C1.TS1  Receive  2015 11 16 11 25 28 850
3 0  C1.TS2  Receive  2015 11 16 11 25 32 474
```

This shows the asynchronous execution of a dynamic function; each of the two commands `TS1` and `TS2` calls the `DRC.Wait` function in a separate thread. Each function call returns the thread number and the result. Calls to the `DRC.Wait` function are thread switching points, which means that threads can be held while other threads continue execution.

4.6.1 Multi-threading

Conga supports multi-threaded applications. The ability to have a program work as both client and server simultaneously, without blocking other threads, has been an integral part of its design. All calls to Conga are implemented as asynchronous calls to an external library (Microsoft Windows DLL or UNIX/Linux Shared Library). Conga uses multiple operating system threads to handle TCP communications; this is independent of the interpreter. Each result is returned to the APL thread that is waiting for it.

When developing an application, it is important to ensure 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 APL thread waiting on the same object is not recommended – it can lead to unpredictable behaviour. For example:

Thread 1: `DRC.Wait 'S1' 1000`
Thread 2: `DRC.Wait 'S1' 1000`

could result in problems, whereas

Thread 1: `DRC.Wait 'S1' 1000`
Thread 2: `DRC.Wait 'S2' 1000`

is fine; this is determined by the application developer.



If a thread sustains an untrapped error then, by default, its execution is suspended and any other threads are paused; resuming execution of a suspended function only restarts the suspended thread. If the Session appears to lock while testing the multi-threading functionality, selecting the menu item **Threads** > **Resume all Threads** reactivates any paused threads.

5 Secure Connections

Conga supports secure connections using SSL/TLS protocols. Secure connections allow client and server applications to:

- verify the identity of the partner that they are connected to.
- encrypt messages so that the contents cannot be deciphered by a third party, even when using text or raw mode connections.
- 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 (Transport Layer Security) is the successor to SSL (Secure Socket Layer) and 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.

Recommended resources:

- http://technet.microsoft.com/en-us/library/cc784450(WS.10).aspx discusses the history, differences, benefits, etc. of SSL/TLS
- http://developer.mozilla.org/en/docs/Introduction_to_Public-Key_Cryptography provides an overview of the public key cryptography techniques used in SSL/TLS; the sections on the SSL protocol and CA (certificate authority) certificates are relevant for anyone who would like to make use of secure communications.
- http://en.wikipedia.org/wiki/X.509 includes an introduction to how X.509 certificates and certificate authorities are used to establish trust.

To use SSL/TLS, Conga needs to be passed the necessary certificate and public key files when client and server objects are created.

Once a secure connection has been established, the same functions/methods are used to send and receive data (and with the same arguments) as when using a non secure connection.

5.1 CA Certificates

A *certificate authority* (CA) is a third party who is trusted by the parties at each end of a secure communication. The CA certifies that the named issuer of a certificate is the owner of the public key included within that certificate; the party at one end of a secure communication can then verify the identity of the party at the other end (known as the *peer*) using their certificate.

Verifying a CA's signature on a certificate requires having access to the CA's public certificate (often called a *root certificate*). Conga can be used to secure many different types of system, and can require multiple (and sometimes private) root certificates from several CAs.

All public root certificates that are to be used with Conga should be located in a single root certificate directory; the `RootCertDir` property (set using the `DRC.SetProp` function) sets the full path to (and name of) this directory. The main root certificates for the most widely-used CAs are supplied with Conga and are located in the **[DYALOG]/PublicCACerts** directory. The following code can be used to set `RootCertDir` to use these certificates:

```
      DyalogDir←2 ⎕NQ '.' 'GetEnvironment' 'DYALOG'
      DRC.SetProp '.' 'RootCertDir' (DyalogDir,'/PublicCACerts')
0
```

Table 5-1 lists the download pages for root certificates for the CAs used to populate the **[DYALOG]/PublicCACerts** directory. However, most CAs have additional certificates available for download, some of which are application-specific; the latest certificates can be downloaded from the CA's websites.

Table 5-1: *Root Certificates for the most widely-used CAs*

Authority	Download Root Certificates From...
VeriSign, Geotrust & Thawte	http://www.verisign.com/support/roots.html
Comodo	http://www.comodo.com/repository/
GoDaddy & ValiCert	https://certs.godaddy.com/Repository.go

Table 5-1: *Root Certificates for the most widely-used CAs (continued)*

Authority	Download Root Certificates From...
Cybertrust	http://cybertrust.omniroot.com/support/sureserver/rootcert_ap.cfm
Entrust	http://www.entrust.net/developer/index.cfm
CAcert	http://www.cacert.org/index.php?id=3
GlobalSign	https://www.globalsign.com/support/root-certificate/osroot.htm
IPS Servidores	http://www.ips.es/Declaraciones/NuevasCAS/NuevasCAS.html The root certificate is not included in **[DYALOG]/PublicCACerts**.

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

5.2 Client and Server Certificates

Client and server certificates are used to verify the identity of the machines at each end of a secure connection (peers). Conga uses X.509 certificates to establish the identity of the peer in a TLS/SSL connection. An X.509 certificate contains information about the certificate subject and the certificate issuer (the CA that issued the certificate), including details of the public key algorithm and the issuer's digital signature.

Dyalog includes a set of test certificates that can be used to test SSL support. The test certificates are found in **[DYALOG]/TestCertificates**, which has three subfolders called **ca**, **client** and **server**. These test certificates can be used for testing your own code, but must not be used in production code. The provided certificates are:

- **TestCertificates/ca/ca-key.pem**
  The private key for the test CA. Used to sign the client/server and 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/client/John Doe-cert.pem** and **John Doe-key.pem**
  The certificate/key pair used for sample clients (alternatively, the **Jane Doe-cert** pair can be used).
- **TestCertificates/server/localhost-cert.pem** and **localhost-key.pem**
  The certificate/key pair used for sample servers.

5.2.1 Certificate Stores



This only applies when running on the Microsoft Windows operating system and is limited to client-side certificates.

Certificates can be stored in common repository known as a *certificate store*. Conga supports the ability to read certificates from the Microsoft certificate store.

5.2.2 Revocation Lists

Conga does not support the use of Certificate Revocation Lists. However, this functionality could be added in a future version if required.

5.3 Creating a Secure Client



Usually, a client should be defined as secure when it is created. However, a non-secure client can be upgraded to a secure client by setting its StartTLS property – see *Section 5.3.1*.

Conga creates secure clients by passing certificate and key information to the `DRC.Clt` function – this should be done through the `DRC.X509Cert` class.

EXAMPLE – ASSUMES SECURE SERVER ALREADY ESTABLISHED ON PORT 713

```
      DyalogDir←2 ⎕NQ '.' 'GetEnvironment' 'DYALOG'
      TestCerts←DyalogDir,'/TestCertificates/'
      Path←TestCerts,'Client/'
      cert←⎕NEW DRC.ReadCertFromFile Path,'John Doe-cert.pem'
      cert.KeyOrigin←'DER' (Path,'John Doe-key.pem')
      certs←('X509' cert)('SSLValidation' 16)
      DRC.Clt 'C1' 'localhost' 713 'Text', certs
0  C1
```

In this example, `ReadCertFromFile` reads the certificate called **client-cert.pem** and records all of its information in a new instance of the X509Cert class. The next line specifies the location of the **John Doe-key.pem** file, which contains the private key.

The sixth line creates a variable comprising the parameters required for the creation of the secure client and the last line includes this information when creating the secure client.

Alternatively, the locations of the certificate and key files can be explicitly specified:

```
      certs←⊂('PublicCertFile' ('DER' (Path,
                                       'John Doe-cert.pem')))
      certs,←⊂('PrivateKeyFile' ('DER' (Path,
                                       'John Doe-cert.pem')))
      certs,←⊂('SSLValidation' 16)
      DRC.Clt 'C1' 'localhost' 713 'Text', certs
0  C1
```

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

The `SSLValidation` parameter that is included when creating a secure client indicates the TLS flags that control the certificate checking process (see *Appendix C* for a complete list of TLS flag values). A typical flag value for a client connection is 16 (accept the server certificate even if its hostname does not match the one it was trying to connect to).

A certificate is not mandatory when creating a secure client; many secure servers accept connections from clients without certificates (known as *anonymous connections*). In this situation, although the server cannot verify the identity of the client, the connection is still encrypted and safe from tampering. Most commercial web sites use anonymous connections as they mean that sensitive data is protected when transmitted over the internet but customers are not required to have a digital signature. To enable an anonymous connection, an empty certificate can be created as follows:

```
      'X509' (⎕NEW DRC.X509Cert)
X509  #.[X509Cert]
```

OS WIN

EXAMPLE

```
      args←'C1' 'www.dyalog.com' 443 'Text' 100000
      args,←⊂('X509' (⎕NEW DRC.X509Cert))
      DRC.Clt args
0  C1
```

This is successful even though the `RootCertDir` property has not been explicitly set because, on many platforms, Conga will use the operating system's CA certificate collection.

OS UNIX

If a secure server's `RootCertDir` parameter has not been defined to point to a valid CA certificate, then a client will be unable to make a secure connection to that server.

EXAMPLE

```
      args←'C1' 'ssltest.dyalog.com' 713 'Text' 100000
      args,←⊂('X509' (⎕NEW DRC.X509Cert))
      DRC.Clt args
1202  ERR_INVALID_PEER_CERTIFICATE  /* The peers
                            certificate is not valid */  66
```

Without access to a valid CA certificate, validation fails. However, the connection is successful if validation is disabled; this means that, when trying to determine why a connection is failing, it can be useful to set the value of the `SSLValidation` parameter to 32 (accept the server certificate without validating it):

```
      args←'C1' 'ssltest.dyalog.com' 713 'Text' 100000
      args,←⊂('X509' (⎕NEW DRC.X509Cert))
      DRC.SetProp '.' 'RootCertDir' 'path/TestCertificates/ca/'
0
      DRC.Clt args,⊂'SSLValidation' 32
0  C1
```

Having connected without validation, the certificate information can be retrieved and a decision made whether to proceed with the conversation with this server.

EXAMPLE

```
      rc cert←DRC.GetProp 'C1' 'PeerCert'
      ,[1.5]1⊃cert.Formatted.(ValidFrom ValidTo Issuer Subject)
Wed Apr 01 15:28:02 2015
Fri May 04 17:32:04 2018
```

```
C=BE,O=GlobalSign nv-sa,CN=GlobalSign Organization Validation CA
                                                    - SHA256 - G2
C=GB,ST=Hampshire,L=Alton,OU=IT,O=Dyalog Limited,CN=*.dyalog.com
```

Once a secure server and client have been linked, operations are exactly the same as for a non-secure server and client.

5.3.1 Upgrading to a Secure Client

A non-secure client can be upgraded to be secure (as long as the host allows such upgrades) by setting the client's `StartTLS` property. To use `StartTLS` on a client you need to:

- supply the public certificate data from your certificate. This is most easily accomplished by creating an X509Cert instance and using its `AsArg` method.
- provide the address of the host in the `Address` parameter.
- optionally, provide any appropriate `SSLValidation` settings.

EXAMPLE

```
      cert←⍬NEW DRC.X509Cert

      ⍝ create an non-secure connection
      (rc clt)←DRC.Clt '' 'some-address' 5000 'text'

      ⍝ upgrade to a secure connection
      rc←DRC.SetProp clt 'StartTLS' (cert.AsArg,('SSLValidation'
                                 32)('Address' 'some-address'))
```

5.4 Creating a Secure Server



Usually, a server should be defined as secure when it is created. However, a non-secure client connection can be upgraded to be secure by setting its StartTLS property – see *Section 5.4.1*.

Secure servers are created in the same way as secure clients (see *Section 5.3*) with the additional rule that a secure server must have a certificate.

EXAMPLE

```
      DRC.SetProp '.' 'RootCertDir' 'path\TestCertificates\ca\'
0
      cert←⊃DRC.ReadCertFromFile 'path\TestCertificates\
                                          server\server-cert.pem'
```

```
      cert.KeyOrigin←'DER' 'path\TestCertificates\server\x
                                                  xserver-key.pem'
      certs←('X509' cert)('SSLValidation' 64)
      DRC.Srv 'S1' '' 713 'Text',certs
0  S1
```

When a client is connected to a secure server, the server can request the client's certificate information by calling the `DRC.GetProp` function on the connection object (see *Section A.15*). However, it is not mandatory for a client to have a certificate and a server can only request information about a client's certificate if the `SSLValidation` parameter that is included when creating a secure server indicates (see *Section A.29*) includes one of the following TLS flags (see *Appendix C* for a complete list of TLS flag values):

- *RequestClientCertificate* (64) – including this flag means that connections are permitted from clients even if they do not have a certificate; if a client does have a certificate then information on that certificate is passed to the server.
- *RequireClientCertificate* (128) – including this flag means that connections are only permitted from clients that have a certificate.

If no client certificate is requested, or no certificate exists, then certificate information will have zero rows when queried.

The validation of client certificates requires access to root certificates; before requesting any client certificate information the `DRC.SetProp` function (see *Section A.28*) must be called on the root object to identify the folder containing these certificates. For example:

```
DRC.SetProp '.' 'RootCertDir' 'path\TestCertificates\ca'
```

Connections that are rejected due to certificate validation failure do not generate events on the server, so no application code is required to handle this situation.

5.4.1 Upgrading to a Secure Connection

A non-secure connection can be upgraded to be secure by setting the connection's `StartTLS` property. To use `StartTLS` on a server connection you need to:

- supply the public certificate data from a valid certificate for the server. This is most easily accomplished by reading in a certificate as an X509Cert instance and using its `AsArg` method.
- optionally, provide any appropriate `SSLValidation` settings.

EXAMPLE

```
      ⍝ read the server test certificate
      cert←DRC.ReadCertFromFile 'path\TestCertificates\server\
                                              localhost-cert.pem'

      ⍝ create the server
      (rc srv)←DRC.Srv '' '' 5000 'text'

      ⍝ wait for connection (assume one arrives before time out)
      (err con evt dat)←DRC.Wait srv 1000

      ⍝ upgrade the connection
      DRC.SetProp con 'StartTLS' (cert.AsArg,('SSLValidation'
                                                        64))
```

5.5 Using the DRC.X509Cert Class

Conga includes a class to encapsulate certificate handling – `DRC.X509Cert`. For a complete description of the `DRC.X509Cert` class, see *Section A.33*.

Certificate information is returned as an `X509Cert` object. This can:

- be used to validate a peer certificate in combination with flags such as CertAcceptWithoutValidating (see *Table C-1*).
- enable a server to confirm the identity of a client without requiring a login.

The specific information can vary, but usually includes the certificate issuer, subject, public key algorithm, certificate format version, serial number and valid from/to dates. If no certificate exists or, in the case of a server object, no certificate information has been requested (see *Table C-1*), then the `X509Cert` object is an empty vector.

To read one or more certificates from a file:

```
      dyalog←2 ⎕NQ '.' 'GetEnvironment' 'DYALOG'
      path←dyalog,'/TestCertificates/'
      file←path,'client/Jane Doe-cert.pem'
      ⍪myCert←DRC.ReadCertFromFile file
1
```

As only a single certificate is present, the outermost layer of nesting can be removed:

```
      ⊢myCert←⊃myCert
#.[LIB].[X509Cert]
```

```
      myCert.⍞NL ¯2          ⍝ 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 (`Extended` may, in some instances, contain more information). For example:

```
      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)
 2018 5 17 3 33 24 0  2028 5 14 3 33 24 0

      myCert.Formatted.(ValidFrom ValidTo)
Thu May 17 03:33:24 2018
Sun May 14 03:33:24 2028

      myCert.Extended.(ValidFrom ValidTo)
Thu     May 17 03:33:24 2018
Sun    May 14 03:33:24 2028
```

5.5.1 Certificate Chains

A *certificate chain* (also known as a *certification path*) is a hierarchy of certificates used for authentication. The first entity in the chain is the certificate for a specified Conga object. Progressing through the chain, each certificate is signed by the owner of the next entity in the chain (that is, the `Issuer` of the lower certificate is the `Subject` for the certificate above it in the chain), as shown in *Figure 5-1*. The final entity in the chain is a root CA certificate.

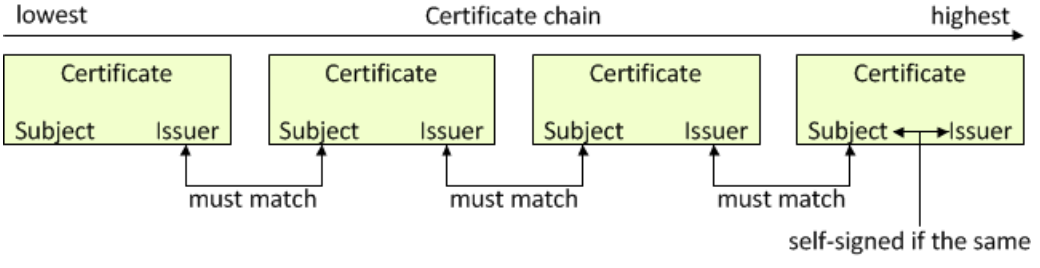


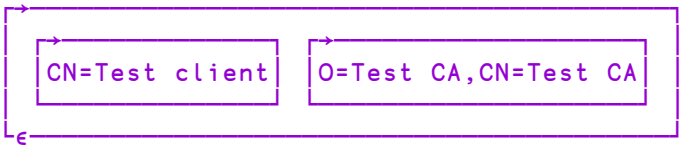
Figure 5-1: *Certificate chain*

Continuing the example in *Section 5.5*:

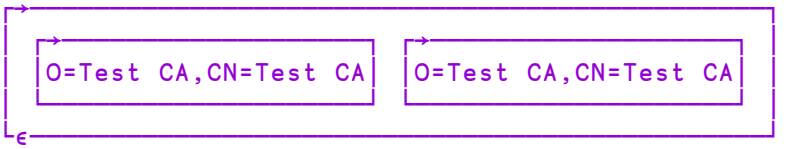
```
      myCert.Chain
#.[LIB].[X509Cert]  #.[LIB].[X509Cert]
```

This is the certificate chain for the test client certificate; it comprises two certificates, the client certificate and the test CA certificate. This can be verified by inspecting the `Issuer` for the lower certificate in the chain and the `Subject` for the higher certificate in the chain – they should be the same:

```
      ]DISPLAY myCert.Chain.Formatted.Subject
```




```
      ]DISPLAY myCert.Chain.Formatted.Issuer
```



The lowest certificate in the chain (in this case, the client certificate) is always shown on the left, and the highest certificate in the chain (in this case, the test CA certificate) is always shown on the far right, with any intermediate certificates in the appropriate location between them. In this example, the `Issuer` for the lower certificate in the chain and the `Subject` for the higher certificate in the chain can be seen to be the same, confirming the authenticity of the client certificate.

In addition, the test CA certificate has the same value for its `Issuer` as it does for its `Subject`; this certificate is, therefore, *self-signed*.

6 HTTP Mode

Hypertext Transfer Protocol (HTTP) is the foundation for communications on the World Wide Web. Prior to Conga v.3.0, applications that wanted to use HTTP would be required to parse and compose HTTP messages in the application. Conga v3.0 introduced *HTTP* mode which can do most of the processing of HTTP messages internally, passing the parsed elements of the HTTP message to your application. *HTTP* mode can be enabled for both clients and servers. Doing so means that Conga will parse messages that are received using the HTTP specification.

HTTP mode is enabled by specifying `'http'` as the mode parameter when the client or server is created. For example:

```
client ← DRC.Clt 'C1' 'www'dyalog.com' 80 'http'
server ← DRC.Srv 'S1' 'localhost' 8080 'http'
```

HTTP mode greatly simplifies the task of building web-enabled server applications, for example, web servers (which are typically accessed using web browsers) or web services that implement a standard medium for client and server applications to communicate. A Conga-based *HTTP*-mode client can be used to retrieve data from web services.

Several of Dyalog's utilities and frameworks use HTTP, including MiServer, JSONServer and HttpCommand.



This document describes how Conga has implemented *HTTP* mode. It is not a comprehensive treatment of HTTP.

6.1 Receiving HTTP Messages

Whether acting as a client or server, the process of receiving HTTP messages is the same and follows one of three patterns:

- Header only
  The entire message is contained in the message header. This is typical with the

HTTP GET method – all information is passed in the HTTP header and there is no body. This is indicated by a 0 (zero) value in the Content-Length HTTP header field.

- Header then Body
  The message is split between the HTTP header block and the HTTP body block. The body contains the data or payload for the message. The size of the body is indicated by the Content-Length header field.
- Header, then one or more Chunks, then a Trailer
  When the payload is large, or is being generated spontaneously, the payload can be split and transmitted in several chunks. A trailer, possibly empty, is sent after the last chunk. Chunked mode is indicated by the Transfer-Encoding header field having the value "chunked" and there being no Content-Length header field.

6.2 HTTP Mode Events

Four events are associated with Conga's HTTP mode:

- `HTTPHeader` (see *Section 6.2.1*)
- `HTTPBody` (see *Section 6.2.2*)
- `HTTPChunk` (see *Section 6.2.3*)
- `HTTPTrailer` (see *Section 6.2.4*)

When any of these events occur, the format of the event's data element depends on whether Conga has been instructed to decode the data. This is determined by the `Options` parameter.

EXAMPLE

```
DRC.Obj '' 'localhost' 8080 'http' ('Options' DRC.Options.DecodeHttp)
```

where `Obj` is either `Clt` or `Srv`.

For more information on the `Options` parameter, see *Section 9.8*.



For backwards compatibility, whether Conga has been instructed to decode the data can also be set using the deprecated server/client/connection object's `DecodeBuffers` property. This can take the values of 0 (Conga returns a simple character vector that needs to be parsed outside Conga) or 15 (Conga return an array of elements that are meaningful to the particular event and more convenient for processing). For example:

```
DRC.SetProp obj 'DecodeBuffers' 15
```

where `obj` is a server, connection or client object.

6.2.1 Event: HTTPHeader

The `HTTPHeader` event is always the first event when receiving an HTTP request or response. The data received with the `HTTPHeader` event depends on whether the object is a client or server. In both scenarios the header values can require additional processing, for example, the value of the header might be Base64-encoded.

When a server receives an HTTP request from a client, the `HTTPHeader` data contains:

`[1]` – the HTTP method

`[2]` – the requested resource (URL)

`[3]` – the HTTP version

`[4]` – a 2-column matrix of header name/value pairs

EXAMPLE

```
┌───┬─┬─────────┬──────────────┬──────────────┐
│GET│/│HTTP/1.1 │Host          │localhost     │
│   │ │         ├──────────────┼──────────────┤
│   │ │         │User-Agent    │Dyalog/Conga  │
│   │ │         ├──────────────┼──────────────┤
│   │ │         │Accept        │*/*           │
│   │ │         ├──────────────┼──────────────┤
│   │ │         │Accept-Encoding│gzip, deflate│
└───┴─┴─────────┴──────────────┴──────────────┘
```

When a client receives an HTTP response from a server, the `HTTPHeader` data contains:

`[1]` – the HTTP version

`[2]` – the HTTP status code

`[3]` – the HTTP status message

`[4]` – a 2-column matrix of header name/value pairs

EXAMPLE

HTTP/1.1	200	OK	Content-Type: text/html;charset=utf-8 Content-Encoding: gzip Server: MiServer/3.1 Content-Length: 876

6.2.2 Event: HTTPBody

The `HTTPBody` event occurs when there is a payload for the HTTP message that has not been broken into chunks by specifying a Transfer-Encoding header value of 'chunked'. The data received is a character vector containing the entire payload (body) of the message. Depending on the value of the Content-Type header, the data might require additional parsing and processing.

6.2.3 Event: HTTPChunk

The `HTTPChunk` event occurs when the message has a Transfer-Encoding header with a value of 'chunked'. Chunked transfer encoding splits the payload for the message into chunks. Each chunk can have *chunk extensions*, which are name/value pairs. Chunk extensions can be used to convey information like progress, a message digest or digital signature. The data for the `HTTPChunk` event contains:

`[1]` the data for the chunk

`[2]` a 2-column matrix of chunk extension name/value pairs

6.2.4 Event: HTTPTrailer

Like the `HTTPChunk` event, the `HTTPTrailer` event occurs when the message has a Transfer-Encoding header with a value of 'chunked'. It is effectively a 0-length chunk, and signals the end of the message. However, instead of chunk extensions, the `HTTPTrailer` event's data comprises 0 or more headers (name/value pairs) – these should be treated as additional headers similar to those received with the `HTTPHeader` event. Normally, such header fields would be sent in the message's header; however, it may be more efficient to determine them after processing the

entire message. In this situation, it is useful to send those headers in the trailer. The data for the `HTTPTrailer` event contains a 2-column matrix (possibly with 0 rows) of any header name/value pairs.

6.2.5 Event: HTTPFail

The `HTTPFail` event occurs if Conga cannot parse the data received for an `HTTPHeader`, `HTTPChunk` or `HTTPTrailer` event. In this situation, the unparsable data is returned as a character vector in the data element of the event.

6.2.6 Event: HTTPError

The `HTTPError` event occurs when the connection is closed during the receipt of an HTTP header, leaving the header portion of the HTTP message incomplete. In this situation, the received data is returned in the data element of the event.

6.3 Limiting HTTP Message Size

It is a good practice to set a limit for the maximum message size that an HTTP server or client can process. One technique used in denial-of-service (DOS) attacks is to send very large messages with the intent of crashing or crippling a server; limiting the message size can help mitigate these attacks.

By default, an HTTP server or client's `BufferSize` parameter is only used to check the length of data of the `HTTPHeader` event. If the length exceeds `BufferSize`, then error 1135 is generated and the connection is closed. No length checking is done on subsequent `HTTPBody`, `HTTPChunk`, or `HTTPTrailer` events.

There are two ways to impose length checking on all HTTP events. Both of these enforce length checking by:

- checking the value of the content-length header, if it exists, in the `HTTPHeader` event's headers.
- checking the length of the data in the `HTTPHeader`, `HTTPBody`, `HTTPChunk`, and `HTTPTrailer` events.

The first of these techniques can be achieved by setting the `DOSLimit` property on the root object. `DOSLimit` is independent of `BufferSize`. For example:

```
DRC.SetProp '.' 'DOSLimit' 100000
```

The second of these techniques can be achieved using `DRC.Options.EnableBufferSizeHttp` on the server or client (see *Section 9.8*). This causes a `BufferSize` check on all HTTP event data. For example:

```
DRC.Srv 'S1' 'localhost' 8080 'http' 10000 ('Options'
          DRC.Options.(DecodeHttp+EnableBufferSizeHttp))
```



If both `DOSLimit` and `BufferSize` are set, then the smaller value applies. Dyalog Ltd recommends using a modest `BufferSize` and not setting `EnableBufferSizeHttp` to ensure that abnormally large headers are not processed, then setting an appropriate `DOSLimit` to accommodate the expected size messages.

If the event data length exceeds `BufferSize`, then error 1135 is returned and the connection is closed. If the event data length exceeds `DOSLimit`, then error 1146 is returned and the connection is closed.

6.4 Sending HTTP Messages

To send an HTTP message, do one of the following:

- Compose a properly formatted HTTP message as a character vector and pass it to the `DRC.Send` function.
- Pass the `DRC.Send` function an array of up to 5 elements.

If passing an array of up to 5 elements to the `DRC.Send` function, the content of these elements depends on whether the message is a request or a response.

When sending a request the elements are:

`[1]` – HTTP method (for example, `'GET'` or `'POST'`)

`[2]` – URL of the requested resource

`[3]` – HTTP version (for example, `'HTTP/1.1'`)

`[4]` – 2-column matrix of head name/value pairs

`[5]` – Either a character vector representing the message body or a 2 or 3-element array of `('' 'filename' ['gzip|deflate'])` where `filename` is the name of a file to send (without having to first read it into the workspace) and `gzip|deflate` indicates the acceptable optional compression schemes that can be applied when sending the file.

When sending a response the elements are:

`[1]` – HTTP version (for example, `'HTTP/1.1'`)

`[2]` – HTTP status code (for example, `'200'`)

`[3]` – HTTP status message (for example, `'OK'`)

`[4]` – 2-column matrix of head name/value pairs

`[5]` – Either a character vector representing the message body or a 2 or 3-element array of `('' 'filename' ['gzip|deflate'])` where `filename` is the name of a file to send (without having to first read it into the workspace) and `gzip|deflate` is the compression scheme applied when sending the file – this must be one that the request specified as being acceptable in the accept-encoding header.

6.5 WebSocket Protocol

WebSocket protocol provides full-duplex communication channels over a single TCP connection. When an established HTTP connection is upgraded to a bi-directional WebSocket connection, both client and server can transmit data at any time rather than being restrained by the normal cycle of the client making a request followed by a server response. Conga client and server objects both support WebSockets.

6.5.1 Client-side WebSocket Upgrade

Requests to upgrade an existing HTTP connection to use the WebSocket protocol are always initiated by the client. Before initiating an upgrade, the client needs to know whether to automatically accept a positive response to an upgrade request from the server. This is achieved either by setting the value of the `WSFeatures` property (deprecated but retained for backwards compatibility) or by specifying the `WSAutoUpgrade` option in the client's `Options` parameter.



Dyalog Ltd recommends allowing automatic upgrades unless you have a good understanding of the WebSocket protocol.

EXAMPLE

There are three ways in which the client can be configured to automatically accept positive WebSocket upgrade responses from the server:

```
⍝ Recommended approach
      DRC.Clt 'C1' 'localhost' 8080 'http' ('Options'
                                      DRC.Options.WSAutoUpgrade)

⍝ Alternative approach (possibility of race conditions)
      DRC.Clt 'C1' 'localhost' 8080 'http'
      DRC.SetProp 'C1' 'Options' DRC.Options.WSAutoUpgrade

⍝ Deprecated approach but retained for backwards compatibility
      DRC.Clt 'C1' 'localhost' 8080 'http'
      DRC.SetProp 'C1' 'WSFeatures' 1
```

The subsequent upgrade process depends on the automatic upgrade setting; *Table 6-1* summarises the steps this process follows.

Table 6-1: *Effect of setting Automatic Upgrade on the Server/Client*

		Automatic Upgrade on Server	
		No	**Yes**
Automatic Upgrade on Client	**No**	1. Client sets the `WSUpgrade` property 2. Server receives a `WSUpgradeReq` event 3. Server inspects headers and, if acceptable, sets the `WSAccept` property 4. Client receives a `WSResponse` event 5. Client inspects headers (and optional additional data) and, if acceptable, sets the `WSAccept` property	1. Client sets the `WSUpgrade` property 2. Server receives (and automatically accepts) a `WSUpgrade` event 3. Client receives a `WSResponse` event 4. Client inspects headers (and optional additional data) and, if acceptable, sets the `WSAccept` property
	Yes	1. Client sets the `WSUpgrade` property 2. Server receives a `WSUpgradeReq` event 3. Server inspects headers and, if acceptable, sets the `WSAccept` property 4. Client receives a `WSUpgrade` event	1. Client sets the `WSUpgrade` property 2. Server receives (and automatically accepts) a `WSUpgrade` event 3. Client receives a `WSUpgrade` event

EXAMPLE

This example creates a client object and upgrades it to use the WebSocket protocol. The `WSAutoUpgrade` property on this example client object is not set (responses must be manually validated/confirmed)

Create an HTTP client:

```
      DRC.Clt 'C1' 'localhost' 8080 'http' ('Options'
                                          DRC.Options.DecodeHttp)
0  C1
```

Initiate the upgrade process by setting the `WSUpgrade` property. This property's value is a 3-element vector specifying:

`[1]` – the path that follows the hostname in the URL – setting it to `/` means use the default page/resource for the host.

`[2]` – the hostname (usually that of the server that the client is connected to).



The server needs to know what page that the WebSocket is going to be associated with. A complete URL specification includes a protocol (`ws://` or `wss://`), host and page in a single string, for example, `ws://demos.kaazing.com/echo` – when breaking it down, the protocol is implicit (based on whether the client connection was opened as a secure connection), the host is `demos.kaazing.com` and the page is `/echo`.

`[3]` – any additional header information that the server being connecting to might need to determine how it handles the connection.

```
      DRC.SetProp 'C1' 'WSUpgrade' ('/' 'localhost'
                                                  'prop:
value')
0
```

Call the `DRC.Wait` function. If the server accepts the request then the client will receive a `WSResponse` event including data containing status and header information from the server (this is a means of passing additional information between the client and server and is part of the WebSocket protocol – the specific information is determined by the developer, for example, a unique hash code for security):

```
      □←res←DRC.Wait 'C1'
0  C1  WSResponse  HTTP/1.1 101 Switching Protocols
                   Upgrade: websocket
                   Connection: Upgrade
                   Sec-Websocket-Accept:
                                      G/cEt4HtsYEnP0MnSVkKRk459
```

The `WSResponse` event was received because `WSFeatures` was set to `0`; if `WSFeatures` had been set to `1`, then the response would have been a `WSUpgrade` event:

```
0  C1  WSUpgrade  0
```

As auto-accept is not enabled, the headers must be validated manually. If they are acceptable then the `WSAccept` property should be set. You are required to confirm the headers that you want to accept as the first element. The second element is not used but is required for symmetry with the server call (see *Section 6.5.2*).



If the headers are not acceptable then an HTTP response with a 4XX status code can be sent (see https://developer.mozilla.org/en-US/docs/Web/HTTP/Status) or the connection can be closed.

```
      DRC.SetProp 'C1' 'WSAccept' ((4⊃res)'')
0
```

The next call to the `DRC.Wait` function should return a `WSResponse` event, after which the socket can be used as a WebSocket:

```
      DRC.Wait 'C1'
0  C1  WSResponse  HTTP/1.1 101 Switching Protocols
                   Upgrade: websocket
                   Connection: Upgrade
                   Sec-Websocket-Accept:
                                      G/cEt4HtsYEnP0MnSVkKRk459
```

6.5.2 Server-side WebSocket Upgrade

When a client requests an upgrade to use the WebSocket protocol, the server can automatically accept the request (Dyalog Ltd recommends allowing automatic upgrades unless you have a good understanding of the WebSocket protocol). This is achieved either by setting the value of the `WSFeatures` property (deprecated but retained for backwards compatibility) or by specifying the `WSAutoUpgrade` option in the server's `Options` parameter. If requests to upgrade to use the WebSocket protocol are automatically accepted, then the server will receive a `WSUpgrade` event when such an upgrade is requested by a client; otherwise the request will generate a `WSUpgradeReq` event from a call to the `DRC.Wait` function.

EXAMPLE

```
      □←res←DRC.Wait 'S1'
0  S1.CON00000000  WSUpgradeReq  GET / HTTP/1.1
                                 Host: localhost
                                 Upgrade: websocket
                                 Connection: Upgrade
                                 some-setting: value
                                 Sec-WebSocket-Version: 13
                                 Sec-WebSocket-Key:KSO+hOFs1q5
                                                SkEnx8bvp6w==
```

Any information that needs to be sent to the client comprises HTTP header properties; each key-value pair must be CRLF-terminated. The format of the messages is the same for both a `WSUpgrade` event and a `WSUpgradeReq` event. If a `WSUpgrade` event is received then element `[4]` is for information purposes only and the upgrade has been performed. If a `WSUpgradeReq` event is received then you need to follow a similar pattern as for the client-side upgrade (see *Section 6.5.1*), either closing the connection if the request is denied or responding with a confirmation of the received headers plus any information that need to be sent to the client:

```
      DRC.SetProp 'S1.CON00000000' 'WSAccept' ((4⊃res)
                                   ('server-says: hello',CRLF))
0
```

The socket is considered open and the server should be able to transmit data, even before the client has received a `WSResponse` event (if it is a Conga client or the equivalent in JavaScript or some other programming language). However, as confirmation that the WebSocket is operational, it might be prudent to wait for the client to initiate communications.

6.5.3 Transmitting WebSocket Data

Once an HTTP connection has been upgraded to a WebSocket, the `DRC.Send` function can be used to transmit data. This data can be sent in one or more fragments.

The argument to the `DRC.Send` function must be a 2- or 3-element vector, where:

`[1]` is a character or integer vector of the data to be transmitted

`[2]` is a Boolean that declares whether this is the final fragment in a sequence:

- 0 indicates that this is not the final fragment in a sequence
- 1 indicates that this is the final (or only) fragment in a sequence

`[3]` (optional) specifies the "operation code" (if an operation code is not specified then it is inferred from the data). Possible values are:

- 1 for Text (data must be character and will be converted to UTF-8)
- 2 for Binary (values between -128 and +255)
- 0 for a continuation (the data must have the same type as your earlier transmission)

When sending data:

- Binary data can be sent as single-byte (signed) integer (⎕DR 83) in the range -128 to 127 or in the range 0 to 255, which could be single-byte or double-byte (⎕DR 83 or ⎕DR 163). Data in the range -128 to 127 is mapped to hexadecimal values 80-FF (128 to 255).
- Single-byte character data (⎕DR 80 or ⎕DR 82) can be sent as binary by specifying an operation code of 2.

EXAMPLES

```
      DRC.Send 'C1' ('Hello there' 1 1)
0

⍝ Send 'Hello there' in 2 fragments
      DRC.Send 'C1' ('Hello ' 0 1)
0
      DRC.Send 'C1' ('there' 1 0)    ⍝ FIN flag set to 1
0
```

6.5.4 Receiving WebSocket Data

Incoming data is returned by the `DRC.Wait` function in the form of a `WSReceive` event; this comprises the same `data` elements as the argument to the `DRC.Send` function used to transmit the data (see *Section 6.5.3*).

A FIN flag of 0 indicates that the incoming data is split into multiple fragments; an application should buffer or concatenate the fragments until receiving a FIN flag of 1 (for an explanation of FIN flags, see *Section 6.5.3*).

6.5.5 Secure WebSockets

To use secure WebSockets, the client or server needs to be established using secure connections (see *Chapter 5*).

EXAMPLE

This example demonstrates using secure WebSockets with a publicly-visible WebSocket-echo server.

```
⍝ Create an empty certificate
      cert←⎕NEW DRC.X509Cert ⍬

⍝ Connect to the server (enabling automatic upgrade)
      DRC.Clt 'c1' 'echo.websocket.org' 80 'http' ('X509' cert)
                            ('Options' DRC.Options.WSAutoUpgrade)
```

```
┌─┬──┐
│0│c1│
└─┴──┘

A Request the upgrade
      DRC.SetProp 'c1' 'WSUpgrade' ('/' 'echo.websocket.org' '')
0

A Get the response
      DRC.Wait 'c1'
┌─┬──┬─────────┬──────────────────────────────────────────────┐
│0│c1│WSUpgrade│HTTP/1.1 101 Web Socket Protocol Handshake    │
│ │  │         │                                              │
│ │  │         │Connection: Upgrade                           │
│ │  │         │                                              │
│ │  │         │Date: Mon, 10 Jun 2019 04:12:15 GMT           │
│ │  │         │                                              │
│ │  │         │Sec-WebSocket-Accept: iY2q6p0pJDCyWJeWcN8wJCfM=│
│ │  │         │                                              │
│ │  │         │Server: Kaazing Gateway                       │
│ │  │         │                                              │
│ │  │         │Upgrade: websocket                            │
└─┴──┴─────────┴──────────────────────────────────────────────┘

A Send a message...
      DRC.Send 'c1' ('hello' 1)
┌─┬──────────────┐
│0│c1.Auto00000000│
└─┴──────────────┘

A ...which is simply echoed back
      DRC.Wait 'c1'
┌─┬──┬─────────┬─────┐
│0│c1│WSReceive│┌─┬─┐│
│ │  │         ││hello│1│1│
│ │  │         │└─┴─┘│
└─┴──┴─────────┴─────┘

A Close the connection
      DRC.Close 'c1'
0
```

7 The Conga Workspace

Table 7-1 summarises the contents of the `conga` workspace.

Table 7-1: *Contents of the `conga` workspace*

Name	Description
`DRC`	Deprecated namespace supplied for backwards compatibility with existing applications that copy this from the `conga` workspace. Contains the Conga interface functions and methods – these are detailed fully in *Appendix A*.
`Conga`	Namespace containing the Conga interface functions and methods – these are detailed fully in *Appendix A*.

8 Utilities and Samples

8.1 Utilities

Conga utilities are located in **[DYALOG]/Library/Conga**, which includes the following:

- **FTPClient** – a class that implements an FTP client and is used to send and retrieve files with FTP servers.
- **HttpCommand** – a general-purpose HTTP client that can be used to (for example) retrieve data from the web, communicate with web services and read web pages (`HttpCommand.Documentation` will display the documentation relating to `HttpCommand`).
- **InitConga** – a utility that helps a user to safely initialise Conga in an environment where other components might also be using Conga.

Each of these utilities can be loaded into your workspace using the `]Load` user command.

EXAMPLE

```
]Load HttpCommand
```

The most recently-updated versions of these Conga utilities are in Dyalog's GitHub repository at https://github.com/Dyalog/library-conga; documentation for each of these utilities is in Dyalog's GitHub repository at https://github.com/Dyalog/library-conga/tree/master/Documentation.

8.2 Samples

Conga samples are located in **[DYALOG]/Samples/Conga**, which includes the following sub-directories:

- **CertTool** – contains the `CertTool` namespace, which demonstrates how the GnuTLS certTool can be used to generate self-signed certificates.

- **HttpServers** – contains several sample HTTP servers, including a WebSocket-based chat server.

  These servers are based on a new, experimental, class-based architecture.

- **RPCServices** – contains several simple remote procedure call servers that use Conga's *Command* mod, which allows APL processes to exchange APL data directly.
- **TODServer** – contains the `TODServer` namespace containing two functions, `RunText` and `RunCommand` (for example usage, see *Section 4.4* and *Section 4.5*).

Each of these samples can be loaded into your workspace using the `]Load` user command.

EXAMPLE

```
]Load [DYALOG]/Samples/Conga/CertTool/CertTool
```

The most recently-updated versions of these Conga samples are in Dyalog's GitHub repository at https://github.com/Dyalog/samples-conga.

9 Advanced Usage

9.1 ConnectionOnly Property

Some server applications have a structure that makes it convenient to launch an APL thread for each client connection and leave that thread running for the duration of that client session. The `ConnectionOnly` property allows a user to specify that calling the `DRC.Wait` function on the server object should not report all events but only `Connect` events; individual application threads are expected to call the `DRC.Wait` function on the connection that they are managing.

The `ConnectionOnly` property is set using the `DRC.SetProp` function.

EXAMPLE

```
DRC.SetProp 's1' 'ConnectionOnly' 1
```

[DYALOG]/Samples/Conga/RPCServices/ThreadedRPC contains an example of a server that has the `ConnectionOnly` property set to `1` and runs a thread for each connection.

9.2 Sending Files

In addition to sending data directly from an APL workspace, the contents of a file can be sent without having to load it into the workspace first.



Prior to Conga version 3.0, the contents of a file had to be loaded into the workspace before they could be sent.

The syntax required for the `DRC.Send` functions data element depends on the mode.

In *Text*/*BlkText*/*Raw*/*BlkRaw* mode, `data` is a 2-element vector comprising:

- `[1]` any data to be transmitted before the contents of the file – this allows you to prepend information to the transmission if required
- `[2]` the full path and name of the file to send

EXAMPLE

```
DRC.Send 'c1' ('' 'c:\tmp\foo.txt')
```

In *HTTP* mode (when not upgraded to use the WebSocket protocol), `data` is one of the following:

- a 2-element vector in which:
  - [1] an empty vector (that is, `''`)
  - [2] the name of a file to send (containing a complete, properly-formatted, HTTP message)

  EXAMPLE

  ```
  DRC.Send obj ('' '/sample.txt')
  ```

- an array of up to 5 elements; the content of these elements depends on whether the message is a request or a response:

  When sending a request the elements are:

  - [1] HTTP method (for example, `'GET'` or `'POST'`)
  - [2] URL of the requested resource
  - [3] HTTP version (for example, `'HTTP/1.1'`)
  - [4] 2-column matrix of head name/value pairs
  - [5] Either a character vector representing the message body or a 2 or 3-element array of `('' 'filename' ['gzip|deflate'])` where `filename` is the name of a file to send (containing only the message body to be sent) and `gzip|deflate` indicates the acceptable optional compression schemes that can be applied when sending the file.

  When sending a response the elements are:

  - [1] HTTP version (for example, `'HTTP/1.1'`)
  - [2] HTTP status code (for example, `'200'`)
  - [3] HTTP status message (for example, `'OK'`)
  - [4] 2-column matrix of head name/value pairs
  - [5] Either a character vector representing the message body or a 2 or 3-element array of `('' 'filename' ['gzip|deflate'])` where

`filename` is the name of a file to send (containing only the message body to be sent) and `gzip|deflate` is the compression scheme applied when sending the file – this must be one that the request specified as being acceptable in the accept-encoding header.

EXAMPLE

```
      DRC.Send obj ('HTTP/1.1' 200 'OK' headers (''
'/foo.txt' 'gzip'))
```

In *HTTP* mode (when upgraded to use the WebSocket protocol), `data` is a 2- or 3-element vector, where:

- `[1]` is a 2-element vector comprising:
  - `[1]` any data to be transmitted before the contents of the file
  - `[2]` the full path and name of the file to send
- `[2]` is a Boolean that declares whether this is the final transmission in a sequence:
  - 0 indicates that this is not the final transmission in the sequence
  - 1 indicates that this is the final transmission in the sequence
- `[3]` (optional) specifies the "operation code". Possible values are:
  - 1 for Text (data must be character and will be converted to UTF-8)
  - 2 for Binary (values between -128 and +127)
  - 0 for a continuation (the data must have the same type as your earlier transmission)

EXAMPLE

```
      DRC.Send 'ws1' (('' 'c:\tmp\foo.txt') 1 1)
```

In *Command* mode, the file transfer mechanism is not possible.

9.3 Compression Level

The `CompLevel` property specifies the level of compression to use when sending data in *Command* mode or a file in *HTTP* mode. Valid values are `0-9`, where `0` indicates no compression and `9` offers maximum compression. Higher levels of compression might consume more CPU time, so you may need to balance the performance of your network versus your CPU. The default is `6`.

The compression level can be set either using the `DRC.SetProp` function (see *Section A.28*) or by specifying it when creating the client or server. Dyalog Ltd recommends the latter approach to avoid potential race conditions.

```
DRC.Srv '' '' 8080 'http' ('CompLevel' 9)
DRC.SetProp '.' 'CompLevel' 9
```

9.4 Temporarily Prevent New Connections

The `Pause` property sets the status of the server's listening socket. This is useful if a server needs a break from incoming connections because, for example, it is preparing to shut down for maintenance or is overloaded. Possible values are:

- 0 : Continue normal operations. This is the default.
- 1 : Keep the server's listening socket open but do not accept new incoming connections; connection attempts that have not timed out on the client side will be accepted when Pause is set to 0.
- 2: Close the server's listening socket but keep the server object alive; recreate the server's listening socket when Pause is set to 0.

EXAMPLE

```
DRC.SetProp 's1' 'Pause' 1    ⍝ do not accept connections
```

9.5 Allow/Deny Connections from Specific Address Ranges

Ranges of IPv4 and IPv6 addresses can automatically be granted/denied connections. This alleviates the need to perform validation of valid peer addresses in APL application code.

Address ranges are specified using the `AllowEndPoints` or `DenyEndPoints` properties when a server is started (using the `DRC.Srv` function). Each set of ranges is specified with a `/` character as the separator. Ranges can be specified using the IPv4 and/or IPv6 connection protocol, and any number of ranges can be specified with a `,` character as the separator. If both types of ranges are specified, then any overlap between the ranges will be disallowed.

EXAMPLE

To tell the server being created to only accept connections from IP addresses 192.168.1.1 through 192.168.1.127 and 10.17.221.67 through 10.17.221.75:

```
allow←,⊂'IPV4' '192.168.1.1/127,10.17.221.67/75'
DRC.Srv '' 'localhost' 8080 ('AllowEndPoints' allow)
```

9.6 Timeout and Closed Events

The `EventMode` property determines whether a socket closing/timing out results in an error message being generated or is reported as event. It is set on the root object and applies to all clients and servers created as children of that root. Possible values are:

- 0 : Closing a socket generates error number 1119; a socket timing out generates error number 100. This is the default.
- 1: Closing a socket generates a `Closed` event; a socket timing out generates a `Timeout` event.

The `EventMode` property is set using the `DRC.SetProp` function. For example, for a socket closing/timing out to generate an event rather than report an error:

```
DRC.SetProp '.' 'EventMode' 1
```

The effect that setting the `EventMode` property has on what is returned by the `DRC.Wait` function is shown in *Table 9-1*.

Table 9-1: *Returned by the `DRC.Wait` function*

	'Eventmode' 0	'Eventmode' 1
Socket closes	`1119 'ERR_CLOSED' 'Socket closed while receiving'`	`0 'S1' 'Closed' 1119`
Socket times out	`100 'TIMEOUT' ''`	`0 'S1' 'Timeout' 100`



Returning events instead of errors is potentially a breaking change for legacy code; it is, therefore, not enabled by default. However, Dyalog Ltd recommends that you set the `EventMode` property to `1`, and it is likely that this will become the default in a future version of Conga.

9.7 Sent Event

Conga can make repeated calls to the `DRC.Send` function without waiting for the previous call to the `DRC.Send` function to complete. This can cause large amounts of data to accumulate in buffers (either in Conga or the network layer), which can be undesirable – it also makes it difficult to cancel an operation, as a large number of operations could be in the queue.

To mitigate these issues, a receipt can be requested on completion of a transmission; this receipt takes the form of a `Sent` event. Receiving a `Sent` event indicates that data has been transmitted and is no longer in the buffer; this means that there is capacity for more `DRC.Send` function calls to be made.

A `Sent` event is requested by setting the `DRC.Send` function's `close` parameter to `3`.

EXAMPLE

```
      DRC.Send 'C1' data 3
0

      DRC.Wait 'C1'
0  C1  Sent  0
```



When running in *Command* mode, the `Sent` event is suppressed by the response to the command if that is received before the next call to the `DRC.Wait` function.

9.8 Options Parameter

Some client/server parameters can be set using the `Options` parameter (see *Table 9-2*).

Table 9-2: *Settings/Values of the Options parameter*

DRC.Options.*	Value	Applicable Modes	Action When Specified
WSAutoUpgrade	1	*Http*	Requests to upgrade to use the WebSocket protocol are automatically accepted
RawAsByte	2	*Raw*, *BlkRaw*	Returns data as type 83 (single-byte ¯128-127); otherwise type 163 (0-255)
DecodeHttp	4	*Http*	HTTP messages are parsed into an array of elements that are meaningful to the particular event and more convenient for processing; otherwise character streams are left for the application to parse
EnableBufferSizeHttp	16	*Http*	The BufferSize parameter specified at the creation of an HTTP client or server is used to check the size of HTTP header, body, chunk and trailer messages; messages that exceed BufferSize cause error 1135 to be generated.
EnableFifo	32	*all*	Conga processes requests in their order of arrival (First-In-First-Out). This overrides the object's ReadyStrategy property.

The values are additive, and should all be set in a single call. The cumulative value can be used instead of the names (names are used throughout this document for transparency). Not all settings apply to all Conga modes.

The `Options` parameter can be set either when creating a server/client or subsequently with the `DRC.SetProp` function. Dyalog Ltd recommends the former (at object creation) to eliminate the small window of exposure between the creation of the object and the setting of the parameter where a request could be received but not processed.

The `DRC.Options` namespace contains named enumerations for each of the settings (see *Section A.18*).

EXAMPLE

To create a Conga client that automatically accepts positive WebSocket upgrade responses from the server and decodes HTTP messages:

```
⍝ Recommended
      DRC.Clt 'C1' 'localhost' 8080 'http' ('Options'
                          DRC.Options.(DecodeHttp+WSAutoUpgrade))

⍝ Using the enumerated (less transparent) equivalent
      DRC.Clt 'C1' 'localhost' 8080 'http' ('Options' 5)

⍝ Alternative
      DRC.Clt 'C1' 'localhost' 8080 'http'
      DRC.SetProp 'C1' 'Options'
                           DRC.Options.(DecodeHttp+WSAutoUpgrade)
```

9.9 Magic Parameter

The `Magic` parameter is only relevant for servers/clients/connections in *BlkText* or *BlkRaw* mode. In these modes, Dyalog Ltd recommends that the `Magic` parameter is used, as its value is a 32-bit integer that is unique to the blocks in a single data transmission and can, therefore, be used to check the integrity of the received data.



A non-Conga client or server can communicate with a Conga peer if the non-Conga component adheres to the message format by pre-pending the result of the following expression to the data to be sent:

```
      ,⍉(4⍴256)⊤8 0+(⍴data) MagicNumber
```

EXAMPLE

Generate a 32-bit integer from a specified string using the `Conga.Magic` function:

```
      ⎕←magic←Conga.Magic 'my secret'
1836654707
```

This integer is used as the `Magic` parameter value for servers/clients/connections in *BlkText* or *BlkRaw* mode. For information on the `Conga.Magic` function, see *Section A.4*.

Start a server using the `Magic` parameter and generated value:

```
      DRC.Srv 'server' '' 8000 'blktext' ('Magic' magic)
0  server
```

There are then two different usage patterns, depending on whether the `Magic` parameter is set when the client is created ("pre-negotiated") or not.

Usage pattern 1: Pre-negotiated (the `Magic` parameter is set when the client is created)

```
      DRC.Clt 'client' '' 8000 'blktext' ('Magic' magic)
0  client
```

Usage pattern 2: Not pre-negotiated (the `Magic` parameter is not set when the client is created, so the client needs to find out the magic number from the server)

```
⍝ The client is started without specifying the Magic parameter:
      DRC.Clt 'client' '' 8000 'blktext'
0  client

⍝ The server sets the Magic property on the incoming connection:
      conx←2⊃ ⍞←DRC.Wait 'server'
0  server.CON00000000  Connect  0
      DRC.SetProp conx 'Magic' magic
0

⍝ The server sends the value to the client over the connection:
      DRC.Send conx ''
0

⍝ The client receives this and sets its own Magic property:
      msg←4⊃DRC.Wait 'client'
      DRC.SetProp 'client' 'Magic' (Conga.Magic msg)
```

9.10 HTTP/1.1 Tunnelling

HTTP tunnelling can be used to make a connection to a proxy server (and, subsequently, to the destination server) using the CONNECT HTTP method. Optionally, a non-secure (HTTP) connection to the proxy server can be upgraded to a

secure (HTTPS) connection to the destination server by setting the `StartTLS` property on the client.

First, connect to the proxy server:

```
      (rc clt)←DRC.Clt '' proxyAddr proxyPort 'http'
```

Next, use the CONNECT HTTP method to connect to the destination server. `proxyHeaders` are any headers required by the destination server, for example, a `Proxy-Authorization` header might be necessary to supply credentials for the proxy connection:

```
      (rc msg)←DRC.Send clt ('CONNECT' (destAddr:destPort)
'HTTP/1.1' proxyHeaders '')
```

If the destination server requires a secure (HTTPS) connection, create (or read) a certificate and set the client's `StartTLS` property using the destination server address and any appropriate `SSLValidation` settings:

```
      cert←⎕NEW DRC.X509Cert
      rc←DRC.SetProp clt 'StartTLS' (cert.AsArg,('SSLValidation'
                                          0)('Address' destAddr))
```

A Technical Reference

The functions, methods and objects in the `Conga` and `DRC` namespace are intended for use by applications; these are documented in this appendix. Any additional functions in these namespaces are for internal use and should not be called by application code.

The notation used when describing the syntax for a function/method in this document is as follows:

- square brackets `[]` indicate an optional argument
- curly braces `{}` indicate a mandatory argument
- a vertical line `|` separates mutually exclusive arguments
- italic text indicates an element that must be populated by the user

The order in which parameters are specified must be as shown in the syntax; however, individual parameters can be specified using parenthesised name-value pairs to eliminate the need to specify all parameters, for example, `('X509' myCert)` or `('EOM' (⎕UCS 13 10))`.



Functions, methods and objects that start "`DRC.`" refer to both objects within the legacy `DRC` namespace and objects in an instance of the `Conga.LIB` class (they are syntactically identical).

A.1 Return Codes

Many of the functions generate a *return code* as the first element of their result. The value of this return code indicates whether the function was successful in its action:

- If the return code is *0*, then the function successfully performed its requisite actions; the rest of the result is as described in this appendix.

- If the return code is not *0*, then the function did not successfully perform its requisite actions; the rest of the result vector comprises *error name* (vector element `[2]`) and *error description*, if available (vector element `[3]`).

Dyalog Ltd recommends that you check the return code in the result before attempting to process other elements of the result. As the number of items returned can vary depending on whether the function successfully performed its requisite actions, this requires code resembling the following:

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

A.2 Conga Object Properties

Different Conga object types can have different properties. The values of these properties are usually specified using the `DRC.SetProp` function (see *Section A.28*) or when a server/client is created using the `DRC.Srv`/`DRC.Clt` function (see *Section A.29* and *Section A.10* respectively), and can be retrieved using the `DRC.GetProp` function (see *Section A.15*).

The possible Conga object properties are described in *Table A-1*. Deprecated properties are described in *Table A-2*.

Table A-1: *Conga object properties*

Property	Object Type	Description/Syntax
CompLevel Get: Y | Set: Y	all	Limitation: *Command* and *HTTP* modes only The compression level to apply/in use when sending data in *Command* mode or a file in *HTTP* mode. Valid values are `0`-`9`, where `0` indicates no compression and `9` indicates maximum compression. The default is `6`.

Table A-1: *Conga object properties (continued)*

Property	Object Type	Description/Syntax
ConnectionOnly Get: Y | Set: Y	Server	Whether all or only `Connect` events are reported when calling the `DRC.Wait` function on the server object. Possible values are: • 0 : Calling the `DRC.Wait` function on the server object reports all events. • 1 : Calling the `DRC.Wait` function on the server object only reports `Connect` events; individual application threads are expected to call the `DRC.Wait` function on the connection that they are managing. The default is 0.
DOSLimit Get: Y | Set: Y	Root	The size limit for `HTTPHeader`, `HTTPBody`, `HTTPChunk` and `HTTPTrailer` events. Used to mitigate denial-of-service (DOS) attacks. The default is 1 MB.
EventMode Get: Y | Set: Y	Root	Whether a socket closing/timing out results in an event or an error message being generated. Possible values are: • 0 : Closing a socket generates error number 1119; a socket timing out generates error number 100. • 1 : Closing a socket generates a `Closed` event; a socket timing out generates a `Timeout` event. The default is 0.
HostName Get: Y | Set: N	Server	The hostname of the server.
HttpDate Get: Y | Set: N	Root	Today's date in the format defined by the HTTP protocol (Internet Standard RFC 1123).

Table A-1: *Conga object properties (continued)*

Property	Object Type	Description/Syntax
KeepAlive Get: Y | Set: Y	Set: Client, Server, Connection Get: Server	The frequency with which periodic heartbeat messages are sent to verify that the connection is live. A 2-element vector in which: • `[1]` is the time (in ms) to wait before sending the first heartbeat • `[2]` is the time interval (in ms) between heartbeats
LocalAddr Get: Y | Set: N	Client, Server, Connection	A 4-element vector in which: • `[1]` is the communication protocol • `[2]` is the IP address (formatted according to the communication protocol) and port number • `[3]` is the address bytes • `[4]` is the port number being used
Magic Get: Y | Set: Y	Client, Connection	Limitation: *BlkRaw* and *BlkText* modes only A 32-bit integer unique to the blocks in a single data transmission and used to check their integrity.
NoDelay Get: Y | Set: Y	Connection	Whether Nagel's algorithm is enabled. Possible values are: • 0 : Nagel's algorithm is enabled; small packages are aggregated rather than being sent immediately. • 1 : Nagel's algorithm is disabled; small packages do not get aggregated but are instead sent immediately. The default is 0.

Table A-1: *Conga object properties (continued)*

Property	Object Type	Description/Syntax
Options Get: Y | Set: Y	Client, Server, Connection	The requisite options for the object. Possible values for individual options are detailed in *Section 9.8*. The default is 0. Dyalog Ltd recommends using the `Options` parameter on the `DRC.Clt`/`DRC.Srv` functions to set options rather than using this property on `DRC.SetProp`.
OwnCert Get: Y | Set: N	Client, Server, Connection	The X509Cert object containing information about the certificate of the specified Conga object.
Pause Get: Y | Set: Y	Server	The "pause" status of the server's listening socket. Possible values are: • 0 : Continue normal operations. • 1 : Keep the server's listening socket open but do not accept new incoming connections; connection attempts that have not timed out on the client side will be accepted when Pause is set to 0. • 2: Close the server's listening socket but keep the server object alive; recreate the server's listening socket when Pause is set to 0. The default is 0.
PeerAddr Get: Y | Set: N	Client, Connection	The address of the specified Conga object's peer. A 4-element vector in which: • `[1]` is the communication protocol • `[2]` is the IP address (formatted according to the communication protocol) and port number • `[3]` is address bytes • `[4]` is the port number being used

Table A-1: *Conga object properties (continued)*

Property	Object Type	Description/Syntax
PeerCert Get: Y | Set: N	Client, Connection	The X509Cert object containing information about the certificate of the specified Conga object's peer.
PropList Get: Y | Set: N	all	A list of the properties relevant for the object's type.
Protocol Get: Y | Set: Y	Root	The communication protocol to use. Possible values are: • IPv4 : use the IPv4 connection protocol; if this is not possible then generate an error • IPv6 : use the IPv6 connection protocol; if this is not possible then generate an error • IP : use the IPv6 connection protocol; if this is not possible then use the IPv4 connection protocol The default is IP. This property value is inherited when creating a connection unless a different value is specified.

Table A-1: *Conga object properties (continued)*

Property	Object Type	Description/Syntax
ReadyStrategy Get: Y | Set: Y	Root	The strategy by which the next connection to process is determined when more than one connection has received data. Possible values are: • 0 : "Use First" – process the first connection in the object tree (for information on the object tree, see *Section A.30*). This approach can result in connections not being serviced. • 1 : "Round Robin" – process the first connection in the object tree after the one that has just been processed. • 2: "Oldest First" – process the connection that has been waiting for the longest time. This is considered to be the least biased approach but consumes slightly more CPU than strategy 1. The default is 2. If `FifoMode` is set on a server object then `ReadyStrategy` for that server object is overridden.
RootCertDir Get: Y | Set: Y	Root	The full path to (and name of) the directory that contains Certificate Authority root certificates.
StartTLS Get: N | Set: Y	Client, Connection	Upgrades an unsecure client/connection to a secure client/connection.

Table A-1: *Conga object properties (continued)*

Property	Object Type	Description/Syntax
TCPLookup Get: Y | Set: N	Root	Requires an additional argument comprising a 2-element vector in which: • `[1]` is a URL or IP address as a character string • `[2]` is the port number (0 means all ports at the URL/IP address) or service name The address of the specified URL/IP address and port. A 4-element vector in which: • `[1]` is the communication protocol • `[2]` is the IP address (formatted according to the communication protocol) and port number • `[3]` is the address bytes • `[4]` is the port number Multiple 4-element vectors can be returned if both IPv4 and IPv6 information is available.
WSAccept Get: N | Set: Y	Connection	Limitation: *HTTP* mode only The positive response to send to a WSUpgradeReq request from a client. A 2-element vector in which: • `[1]` is the data element (element `[4]`) of the `WSUpgradeReq` event received from the client • `[2]` is a character vector of any additional headers that the server might want to send

Table A-1: *Conga object properties (continued)*

Property	Object Type	Description/Syntax
WSUpgrade Get: N | Set: Y	Client	Limitation: *HTTP* mode only A request to upgrade a client to a WebSocket. A 3-element vector of character vectors in which: • `[1]` is the path that follows the hostname in the URL • `[2]` is the hostname (usually that of the server that the client is connected to) • `[3]` is any additional header information that the server being connecting to might need to determine how it handles the connection.

Table A-2: *Deprecated Conga object properties*

Property	Object Type	Description/Syntax
Certificates Get: Y | Set: N	Root	Deprecated: use `DRC.Certs` (see *Section A.7*) Limitation: Microsoft Windows only Retrieves the contents of the Microsoft certificate store named in the argument.
DecodeBuffers Get: N | Set: Y	Client, Server, Connection	Deprecated: use `DRC.Options` (see *Section A.18*) Limitation: *HTTP* mode only Whether HTTP events should return data in a decoded format. Possible values are: • 0 : do not decode any HTTP-related events (a simple character vector is returned) • 15: decode the data for all HTTP-related events (HTTPHeader, HTTPTrailer, HTTPChuck and HTTPBody) The default is 0.
ErrorText Get: Y | Set: N	Root	Deprecated: use `DRC.Error` (see *Section A.13*) Requires an additional argument comprising the single error number of interest; returns the error text corresponding to the error number.

Table A-2: *Deprecated Conga object properties (continued)*

Property	Object Type	Description/Syntax
FifoMode Get: Y | Set: Y	Server	Deprecated: use `DRC.Options` (see *Section A.18*) Boolean indicating whether Conga handles requests in their order of arrival (First-In-First-Out). Possible values are: • 0 : Conga handles requests in the order defined in the `ReadyStrategy` property • 1 : Conga handles requests in their order of arrival (this overrides the `ReadyStrategy` property)
Stores Get: Y | Set: N	Root	Deprecated: use `DRC.Certs` (see *Section A.7*) Limitation: Microsoft Windows only Retrieves the names of the Microsoft certificate stores defined on the local machine.

Table A-2: *Deprecated Conga object properties (continued)*

Property	Object Type	Description/Syntax
WSFeatures Get: N | Set: Y	Client, Server	Deprecated: use `DRC.Options` (see *Section A.18*) Limitation: *HTTP* mode only The details of how an HTTP client should be upgraded to use the WebSocket protocol. Possible values are: • 0 : a `WSUpgrade` request received by the server results in a `WSResponse` event being received by the client; this positive response is not automatically accepted, giving the client the opportunity to validate and confirm the upgrade. • 1 : a `WSUpgrade` request from the client generates a `WSUpgrade` event from the server, sent to the client; this positive response is automatically accepted by the client. This is the recommended approach unless you have a good understanding of the WebSocket protocol. In both cases, the `WSUpgrade` request from the client also generates a `WSUpgradeReq` event on the server. The server can then validate the request and, if acceptable, send a positive response by setting the `WSAccept` property on the connection. For more information see *Table 6-1*. The default is `0`.

A.3 Function: Conga.Init

Purpose: Initialises Conga (if not already initialised) and returns a reference to the new instance of `Conga.LIB`. Use `Conga.Init` in preference to `Conga.New` if an existing root can be reused.

Syntax: `ref ← [libpath] Conga.Init {rootname}`

where:

- `libpath` is the path to the Conga shared library files that are to be used. Possible values are:
  - `''` or `⍬` : if the environment variable `CONGA_LIBPATH` has been defined, use this value, otherwise use the default path as described in the documentation for `⎕NA`. This is the default value.
  - `'⍵'` : use the path to the current workspace
  - `'↓'` : use the current working directory (as returned by `⎕CMD 'cd'` under Microsoft Windows or `⎕CMD 'pwd'` on other platforms)
  - `'⍺'` : use the path to the currently running interpreter (as returned by `2 ⎕NQ '.' 'GetCommandLineArgs'`)
- `ref` is a reference to the instance of `Conga.LIB`.
- `rootname` is the name of the Conga root. An empty rootname (`''`) causes Conga to use the default root name of 'DEFAULT'. If the specified root name does not already exist, Conga creates a new instance of `Conga.LIB` and returns a reference to the new instance. If the specified root name already exists, `Conga.Init` returns a reference to the existing root.

EXAMPLES

```
      i1←Conga.Init ''
      i2←Conga.Init ''
      i3←Conga.New ''

      Conga.RootNames
┌───────┬───┐
│DEFAULT│IC1│
└───────┴───┘

      (i1 i2 i3).RootName
┌───────┬───────┬───┐
│DEFAULT│DEFAULT│IC1│
└───────┴───────┴───┘
```

Related functions:

- `Conga.New` – see *Section A.5*
- `Conga.RootNames` – see *Section A.6*
- `DRC.RootName` – see *Section A.25*

A.4 Function: Conga.Magic

Purpose: Returns a 32-bit integer generated from a specified string. This integer is used as the `Magic` parameter value for servers/clients/connections in *BlkText* or *BlkRaw* mode. It is useful when setting the `Magic` parameter on a client when the client and server have not pre-negotiated the value of the `Magic` parameter. For more information on the `Magic` parameter, see *Section 9.9*.

Syntax: `int ← Conga.Magic '{string}'`

where:

- `int` is a 32-bit integer encoding generated from the specified string.
- `string` is an arbitrary string of Unicode text .

EXAMPLE

```
      Conga.Magic 'Secret'
1399153522
```

A.5 Function: Conga.New

Purpose: Initialises Conga (if not already initialised) and returns a reference to the new instance of `Conga.LIB`. Use `Conga.New` in preference to `Conga.Init` if a new, unique root must be created (that is, an existing root must not be reused).

Syntax: `ref ← Conga.New {rootname}`

where:

- `ref` is a reference to the instance of `Conga.LIB`.
- `rootname` is the name of the Conga root. An empty rootname (`''`) causes Conga to generate a new instance of `Conga.LIB` with a unique root name and return a reference to the instance. If the root name already exists, `Conga.New` generates an error.

EXAMPLES

```
      i1←Conga.New ''
      i2←Conga.New ''
      (i1 i2).RootName
```

IC1	IC2

```
      i3←Conga.New 'IC1'
Name in use: IC1
      i3←Conga.New'IC1'
         ^

      i3←Conga.Init ''
      (i1 i2 i3).RootName
```

IC1	IC2	DEFAULT

Related functions:

- `Conga.Init` – see *Section A.3*
- `Conga.RootNames` – see *Section A.6*
- `DRC.RootName` – see *Section A.25*

A.6 Function: Conga.RootNames

Purpose: Returns the root names of all `Conga.LIB` instances.

Syntax: `rootnames ← Conga.RootNames`

where:

- `rootnames` is a vector of the root names of the Conga root.

Example

```
      i1←Conga.Init ''
      Conga.RootNames
```

DEFAULT

```
      i2←Conga.New 'myRoot'
      Conga.RootNames
```

DEFAULT	myRoot

Now use the `DRC` namespace to initialise Conga. This is not something that you would normally do but is being shown here to illustrate the difference in root reporting between root objects generated from the legacy `DRC` namespace and those generated from an instance of `Conga.LIB`.

```
      DRC.Init ''
┌─┬──────────────────────────┐
│0│Conga loaded from: conga32_64│
└─┴──────────────────────────┘

      Conga.RootNames ⍝ The DRC root name is not in the
result...
┌───────┬──────┐
│DEFAULT│myRoot│
└───────┴──────┘

      DRC.RootName    ⍝ ...even though the DRC ns has a RootName
DRC

      DRC←Conga.New '' ⍝ Make DRC refer to a Conga.LIB instance
      DRC.RootName
IC1

      Conga.RootNames  ⍝ The root name is now shown
┌───────┬──────┬───┐
│DEFAULT│myRoot│IC1│
└───────┴──────┴───┘
```

Related functions:

- `Conga.Init` – see *Section A.3*
- `Conga.New` – see *Section A.5*
- `DRC.RootName` – see *Section A.25*

A.7 Function: DRC.Certs

OS WIN This only applies when running on the Microsoft Windows operating system and is limited to client-side certificates.

Purpose: Returns information from the current user certificate store defined on the local machine.

Syntax: `rc data ← DRC.Certs {'ListMSStore'|'MSStore' 'storename'}`

where:

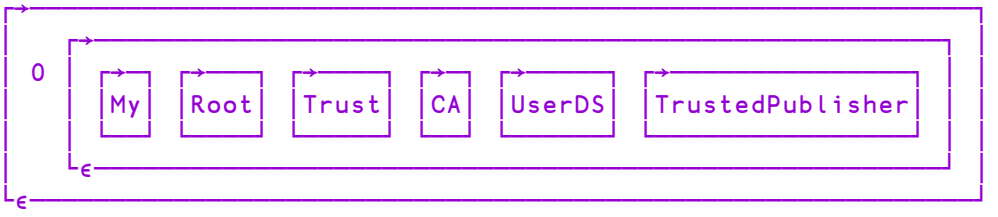
- `rc` is the return code (see *Section A.1*)
- `data` is one of:
  - a vector of character vectors each containing a store name, for example, **My** and **Root**

- a vector of integer vectors containing the certificates from the specified store

- `'ListMSStore'` is an instruction to include Microsoft certificate store names in the returned vector
- `''MSStore' 'storename'` is an instruction to return the list of certificates held in the certificate store named `storename`

EXAMPLES:

```
      ]Display DRC.Certs 'ListMSStore'
```




```
      ≠ca←2⊃DRC.Certs 'MSStore' 'CA'    ⍝ how many in CA?
111

⍝ Display some information about the second certificate
      (⎕NEW DRC.X509Cert (2⊃ca)).Formatted.Issuer
C=US,O=VeriSign\, Inc.,OU=VeriSign Trust Network,OU=(c) 2006
VeriSign\, Inc. - For authorized use only,CN=VeriSign Class 3
Public Primary Certification Authority - G5
```

A.8 Function: DRC.ClientAuth

OS WIN Integrated Windows Authentication is only available on a Microsoft Windows domain – both client and server must be running on Microsoft Windows.

Purpose: Performs client-side Integrated Windows Authentication (IWA). Only valid for a *Command*-mode client connected to a *Command*-mode server

Syntax: `rc ← DRC.ClientAuth {clientname} [{userid} {password}]`

where:

- `rc` is the return code (see *Section A.1*)
- `clientname` is the name of the *Command*-mode client.
- `userid` is the user's Microsoft Windows User ID.
- `password` is the user's Microsoft Windows password.



`DRC.ClientAuth` and `DRC.ServerAuth` (see *Section A.27*) must be run at the same time.

A.9 Function: DRC.Close

Purpose: Closes the specified Conga object.

Syntax: `rc ← DRC.Close {servername|clientname|connectionname|commandname}`

where:

- `rc` is the return code (see *Section A.1*)
- `servername|clientname|connectionname|commandname` is the name of the Conga server/client/connection/command to close (respectively).

EXAMPLE

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

A.10 Function: DRC.Clt

Purpose: Creates a Conga client object.

Syntax: `rc name ← DRC.Clt {clientname} {Address} [Port] [Mode] [BufferSize] [SSLValidation] [EOM] [IgnoreCase] [Protocol] [PublicCertData] [PrivateKeyFile] [PrivateKeyPass] [PublicCertFile] [PublicCertPass] [PrivateKeyData] [Priority] [Magic] [X509] [CompLevel] [Options]`

where:

- `rc` is the return code (see *Section A.1*)
- `name` is the name of the Conga client that has been created. If no `clientname` was specified (`''`) then this is auto-generated.
- `clientame` is the name of the Conga client to create. If `''` is specified rather than a specific name, then the name will be auto-generated.
- `Address` is the address of the server
- `Port` is the port number that the server will listen on

- `Mode` is the connection protocol (see *Section 4.1.3*). Possible values are *Command*, *Text*, *Raw*, *BlkText* or *BlkRaw*. The default is *Command*.
- `BufferSize` is the maximum size (in bytes) allocated to the buffer that receives data transmissions. The default is 16,384. Only valid for clients in *Raw*/*Text*/*BlkRaw*/*BlkText* mode, not those in *Command* mode.
- `SSLValidation` is the sum of the relevant TLS flags (see *Appendix C*). Only valid when creating a secure client (see *Section 5.3*).
- `EOM` is a simple character vector or a vector of vectors indicating the termination string(s) (see *Section 4.1.3*). Only valid for clients in *Raw*/*Text* mode, not those in *BlkRaw*/*BlkText*/*Command* mode.
- `IgnoreCase` is a Boolean indicating whether searches for the termination string defined in EOM are case sensitive. Possible values are:
  - 0 : do not ignore case when searching for termination strings
  - 1 : ignore case when searching for termination strings

  Only valid for clients in *Raw*/*Text* mode, not those in *BlkRaw*/*BlkText*/*Command* mode.
- `Protocol` is the communication protocol to use. Possible values are:
  - IPv4 : use the IPv4 connection protocol; if this is not possible then generate an error
  - IPv6 : use the IPv6 connection protocol; if this is not possible then generate an error
  - IP : use the IPv6 connection protocol; if this is not possible then use the IPv4 connection protocol.

  The default is to inherit the protocol defined for the root object.
- `PublicCertData` has been deprecated but is retained for backwards compatibility. Only valid when creating a secure client (see *Section 5.3*).
- `PrivateKeyFile` has been deprecated but is retained for backwards compatibility. Only valid when creating a secure client (see *Section 5.3*).
- `PrivateKeyPass` has been deprecated but is retained for backwards compatibility. Only valid when creating a secure client (see *Section 5.3*).
- `PublicCertFile` has been deprecated but is retained for backwards compatibility. Only valid when creating a secure client (see *Section 5.3*).
- `PublicCertPass` has been deprecated but is retained for backwards compatibility. Only valid when creating a secure client (see *Section 5.3*).

- `PrivateKeyData` has been deprecated but is retained for backwards compatibility. Only valid when creating a secure client (see *Section 5.3*).
- `Priority` is the GnuTLS priority string (for complete documentation of this, see http://www.gnutls.org/manual/gnutls.html#Priority-Strings).
- `Magic` is a 32-bit integer used to check that the data has not been corrupted. Only valid for clients in *BlkRaw*/*BlkText* mode, not those in *Raw*/*Text*/*Command* mode.
- `X509` is a reference to an instance of the X509Cert class. Only valid when creating a secure client (see *Section 5.3*).
- `CompLevel` is the level of compression to use when sending data in *Command* mode or a file in *HTTP* mode (see *Section 9.3*).
- `Options` is as integer representing the additive values of the client's parameters (see *Section 9.8*).

The `[PublicCertData]`, `[PrivateKeyFile]`, `[PrivateKeyPass]`, `[PublicCertFile]`, `[PublicCertPass]` and `[PrivateKeyData]` arguments are mutally exclusive with the `[X509]` argument.

EXAMPLES

To create C1, a *Command* mode client of a server at 192.168.1.1 listening on port 5050:

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

To create a secure *Command* mode client of the secure *Command* mode server at 192.168.1.1 listening on port 5050:

```
      mycert←⊃DRC.ReadCertFromFile 'path/client-cert.pem'
      mycert.KeyOrigin←'DER' 'path/client-key.pem'
      DRC.Clt 'C1' '192.168.1.1' 5050,⊂('X509' mycert)
                                      ('SSLValidation' 16)
0  C1
```

To create a *Text* mode client (with an auto-generated name) of a server on the same machine, listening on port 30, with a maximum buffer size of 1000 characters and a termination sequence of <CRLF>:

```
      DRC.Clt '' 'localhost' 30 'Text' 1000 ('EOM' (⎕UCS 13 10))
0  CLT00000000
```

Related function:

- `DRC.Srv` – see *Section A.29*

A.11 Function: DRC.DecodeOptions

Purpose: Returns a description of the settings represented by an `Options` value.

Syntax: `desc ← DRC.DecodeOptions {optval}`

where:

- `desc` is the name of the `DRC.Options.*` setting(s) represented by `optval`.
- `optval` is the value of the `Options` parameter.

EXAMPLES

```
      DRC.DecodeOptions 5
DecodeHttp+WSAutoUpgrade

      DRC.DecodeOptions¨1 2 4
```

WSAutoUpgrade	RawAsByte	DecodeHttp

A.12 Function: DRC.Describe

Purpose: Returns a description of the specified Conga object. Similar to the `DRC.Tree` function (see *Section A.30*) except that `DRC.Describe` only returns information for the specified object (not its children) and the descriptions are textual rather than numeric codes.

Syntax: `rc desc ← DRC.Describe {objectname}`

where:

- `rc` is the return code (see *Section A.1*)
- `desc` is a multi-element vector proving a textual description of the specified object
- `objectname` is the name of the Conga object to describe

For all objects except the root object, the description `desc` starts with the following elements:

- `[1]` is the name of the object
- `[2]` is the object's type (see *Section 4.1.1*)

- `[3]` is the object's state (see *Section 4.1.2*)

The description `desc` of objects of type 4 (commands) and 5 (messages) has additional elements:

- `[4]` is the size of the object that has been processed so far (in bytes)
- `[5]` is the size of the object that has not yet been processed (in bytes)

For the root object `'.'`, the description `desc` comprises the following elements:

- `[1]` is [DRC]
- `[2]` is the version of Conga
- `[3]` is the object's state (see *Section 4.1.2*)
- `[4]` is the thread count

EXAMPLES

```
      DRC.Describe '.'
0   [DRC]  Conga Dynamic Link Library 2.6.956.0 Copyright (C)
2004-2015 Dyalog Ltd. built Nov  2 2015 10:48:51. GnuTLS 3.2.15
Copyright (c) 2000-2015 Free Software Foundation, Inc.  Built
Jul  9 2015 at 09:48:37.  Revision: 106   State=RootInit
Threads=0

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

Similar functions:

- `DRC.Names` – see *Section A.17*
- `DRC.Tree` – see *Section A.30*

A.13 Function: DRC.Error

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

Syntax: `no name [desc] ← DRC.Error {no}`

where:

- `no` is the error number
- `name` is the name of the error
- `desc` is a description of the error, for example, `/* unable to complete a TLS handshake with the peer */`

EXAMPLE

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

A.14 Function: DRC.Exists

Purpose: Verifies whether a specified Conga object exists.

Syntax: `bool ← DRC.Exists {objectname}`

where:

- `bool` indicates whether the object exists. Possible values are:
  - 0 – the object does not exist
  - 1 – the object exists
- `objectname` is the name of the Conga object whose existence is being verified

EXAMPLE

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

A.15 Function: DRC.GetProp

Purpose: Retrieves property values for the specified Conga object.

Syntax: `rc res ← DRC.GetProp {objectname} {property} [arg]`

where:

- `rc` is the return code (see *Section A.1*)
- `res` is the retrieved property value; its format depends on the property requested (see *A.15*)
- `objectname` is the name of the Conga object for which to retrieve the value of the specified property
- `property` is the name of the property to retrieve the value for; not all properties are relevant for all object types (see *A.15*)
- `arg` is an additional argument that might be required depending on the property requested (see *A.15*)

The property values that can be retrieved depend on the type of the specified Conga object; they are described in *Appendix A.2*. These values are usually specified using the `DRC.SetProp` function (see *Section A.28*) or when a server/client is created using the `DRC.Srv`/`DRC.Clt` function (see *Section A.29* and *Section A.10* respectively).

EXAMPLES

```
      DRC.GetProp '.'  'PropList'
0   Certificates  DecodeCert  PropList  Protocol  ReadyStrategy
RootCertDir  Stores  TCPLookup

      DRC.GetProp  '.' 'TCPLookup' 'www.dyalog.com' 80
0    IPv6  [2a02:2658:1012::35]:80  42 2 38 88 16 18 0 0 0 0 0 0
0 0 0 53  80   IPv4  81.94.205.35:80  81 94 205 35  80

      DRC.GetProp 'C1' 'OwnCert'
0   #.DRC.[X509Cert]
```

Related functions:

- `DRC.SetProp` – see *Section A.28*

A.16 Function: DRC.Init

Purpose: Loads and initialises (or reinitialises) the Conga Dynamic Link Library (Microsoft Windows) or Shared Library (UNIX/Linux).

Syntax: `rc ← [reset] DRC.Init {libpath}`

where:

- `rc` is the return code (see *Section A.1*)
- `reset` is a code indicating the action to take if Conga has already been initialised. Possible values are:
  - `1` : close any existing Conga objects
  - `¯1` : reload the library
- `libpath` is the path to the Conga shared library files that are to be used. Possible values are:
  - `''` or `⍬` : if the environment variable `CONGA_LIBPATH` has been defined, use this value, otherwise use the default path as described in the documentation for `⎕NA`
  - `'⍵'` : use the path to the current workspace

- `'↓'` : use the current working directory (as returned by `⎕CMD 'cd'` under Microsoft Windows or `⎕CMD 'pwd'` on other platforms)
- `'⍺'` : use the path to the currently running interpreter (as returned by `2 ⎕NQ '.' 'GetCommandLineArgs'`)

For any other value, a message is returned stating that Conga has already been loaded.

EXAMPLE

```
      DRC.Init ''
0  Conga loaded from: ...\conga27x64Uni
```

A.17 Function: DRC.Names

Purpose: Returns the names of existing Conga objects that are first-level descendants of the specified Conga object.

Syntax: `names ← DRC.Names {objectname}`

where:

- `names` is a list of the names of all first-level descendants of the specified Conga object. If there are no first-level descendants of the specified Conga object, then `names` is an empty vector.
- `objectname` is a character vector of the name of the Conga object for which to return the names of its first-level descendants

EXAMPLES

```
      DRC.Names ''
C1  C2  C3

      DRC.(Close¨Names '')
0  0  0
```

Similar functions:

- `DRC.Describe` – see *Section A.12*
- `DRC.Tree` – see *Section A.30*

A.18 Namespace: DRC.Options

Purpose: Contains named enumerations for the `Options` parameter (see *Section 9.8*). While numeric values result in smaller source code, the enumerations are more descriptive. For example, the following statements are equivalent:

```
      DRC.Clt '' '' 8080 'http' ('Options'
                           DRC.Options.
(WSAutoUpgrade+DecodeHttp))

      DRC.Clt '' '' 8080 'http' ('Options' (1+4))

      DRC.Clt '' '' 8080 'http' ('Options' 5)
```

The enumerations in the `Options` namespace are:

```
      DRC.Options.(⍳((⊂∘⍋)⌷⊢){(±⍵)⍵}⍤⌷NL ¯3)
 1  WSAutoUpgrade
 2  RawAsByte
 4  DecodeHttp
16  EnableBufferSizeHttp
32  EnableFifo
```

A.19 Function: DRC.Progress

Purpose: Sends an APL array to a client in response to a command received from that client. Only valid for a *Command*-mode server.

A server can call the `DRC.Progress` function any number of times before calling the `DRC.Respond` function (see *Section A.24*).

Syntax: `rc ← DRC.Progress {commandname} {data}`

where:

- `rc` is the return code (see *Section A.1*)
- `commandname` is the name of the command received from the client. It must match the `objectname` returned by the `DRC.Wait` function (see *Section A.32*)
- `data` is any array

EXAMPLE

A *Command*-mode client sends data to a *Command*-mode server using the `DRC.Wait` function. The server stores the result in `waitresult`. The following expression can be used to send a progress report to the client:

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

Related functions:

- `DRC.Respond` – see *Section A.24*
- `DRC.Send` – see *Section A.26*
- `DRC.Wait` – see *Section A.32*

A.20 Function: DRC.ReadCertFromFile

Purpose: Reads one or more certificates from a **.pem** certificate file as X509Cert instances.

Syntax: `X509 ← DRC.ReadCertFromFile {filename}`

where:

- `X509` is an instance of the X509Cert class for each certificate found within the file.
- `filename` is the complete path and filename of the **.pem** certificate to be read. If a relative path is specified, it will relative to the current directory as reported by `⊃1 ⎕NPARTS`.

EXAMPLE

```
      Dyalog←2 ⎕NQ '.' 'GetEnvironment' 'DYALOG'
      certfile←Dyalog,'/TestCertificates/client/
                                        Jane Doe-cert.pem'
      DRC.ReadCertFromFile certfile
#.[LIB].[X509Cert]
```

Related functions:

- `DRC.ReadCertFromFolder` – see *Section A.21*
- `DRC.ReadCertFromStore` – see *Section A.22*
- `DRC.ReadCertUrls` – see *Section A.23*

A.21 Function: DRC.ReadCertFromFolder

Purpose: Reads one or more certificates from each certificate file in a directory and returns them as instances of the X509Cert class.

Syntax: `X509 ← DRC.ReadCertFromFolder {folderspec}`

where:

- `X509` is a vector of X509Cert instances created from the certificates that were read. If a certificate file contains more than one certificate, the corresponding element in `X509` will be a vector of instances of X509Cert representing the certificates in the file.
- `folderspec` is the name of the directory containing the certificate files to be read. Note that:
  - it can contain the wildcard characters '`?`' and '`*`'.
  - only files conforming to the **.pem** certificate file format will be processed.
  - any files matching `folderspec` that do not confirm to the `.pem` certificate file format will be ignored.

EXAMPLE

```
      Dyalog←2 ⎕NQ '.' 'GetEnvironment' 'DYALOG'
      folder←Dyalog,'/TestCertificates/server/'
      DRC.ReadCertFromFolder folder
#.[LIB].[X509Cert]  #.[LIB].[X509Cert]
```

Related functions:

- `DRC.ReadCertFromFile` – see *Section A.20*
- `DRC.ReadCertFromStore` – see *Section A.22*
- `DRC.ReadCertUrls` – see *Section A.23*

A.22 Function: DRC.ReadCertFromStore

Purpose: Reads all certificates in the current user's Microsoft Windows Certificate Store and returns them as instances of X509Cert.

Syntax: `X509 ← DRC.ReadCertFromStore {storename}`

where:

- `X509` is a vector of X509Cert instances created from the certificates that were read from the certificate store.
- `storename` is the name of a Microsoft Certificate Store; it must match one of the entries returned by `DRC.Certs 'ListMSStore'`.

EXAMPLE

Return the number of certificates in the 'Root' Certificate Store. This corresponds to the "Trusted Root Certification Authorities" store found in the Microsoft Certificate Manager:

```
      ≠DRC.ReadCertFromStore 'Root'
98
```

Related functions:

- `DRC.Certs` – see *Section A.7*
- `DRC.ReadCertFromFile` – see *Section A.20*
- `DRC.ReadCertFromFolder` – see *Section A.21*
- `DRC.ReadCertUrls` – see *Section A.23*

A.23 Function: DRC.ReadCertUrls

Purpose: Reads all certificates in a current user's "Personal" Microsoft Windows Certificate Store and returns them as instances of X509Cert.

Equivalent to running `DRC.ReadCertFromStore 'My'`.

Syntax: `X509 ← DRC.ReadCertUrls`

where:

- `X509` is a vector of X509Cert instances created from the certificates that were read.

EXAMPLE

Return the number of certificates in the current user's "Personal" Microsoft Windows Certificate Store:

```
      ≠DRC.ReadCertUrls
13
```

Related functions:

- `DRC.Certs` – see *Section A.7*
- `DRC.ReadCertFromStore` – see *Section A.22*

A.24 Function: DRC.Respond

Purpose: Sends an APL array to a client in response to a command received from that client. Only valid for a *Command*-mode server.

Syntax: `rc ← DRC.Respond {commandname} {data}`

where:

- `rc` is the return code (see *Section A.1*)
- `commandname` is the name of the command received from the client. It must match the `objectname` returned by the `DRC.Wait` function (see *Section A.32*)
- `data` is any array

EXAMPLE

A *Command*-mode client sends data to a *Command*-mode server using the `DRC.Wait` function. The server stores the result in `waitresult`. The following expression can be used to call the `Process` function on the data that accompanied the most recent command and send the result to the client:

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

Related functions:

- `DRC.Progress` – see *Section A.19*
- `DRC.Send` – see *Section A.26*
- `DRC.Wait` – see *Section A.32*

A.25 Function: DRC.RootName

Purpose: Returns the root name for the `DRC` namespace or `Conga.LIB` instance.

Syntax: `rootname ← DRC.RootName`

where:

- `rootname` is the name of the Conga root:
  - If `DRC` refers to the `DRC` namespace, then `rootname` is the hard-coded character vector `'DRC'`.
  - If `DRC` refers to an instance of `Conga.LIB`, then the `rootname` is the root name generated when the instance was created.

EXAMPLES

```
      DRC.Init ''
┌─┬──────────────────────────────┐
│0│Conga loaded from: conga32_64 │
└─┴──────────────────────────────┘

      DRC.RootName    ⍝ DRC namespace has a RootName of 'DRC'
DRC

      DRC←Conga.New 'myDRC' ⍝ DRC refers to a Conga.LIB instance
      DRC.RootName          ⍝ RootName is as specified
myDRC
```

Related functions:

- `Conga.Init` – see *Section A.3*
- `Conga.New` – see *Section A.5*
- `Conga.RootNames` – see *Section A.6*

A.26 Function: DRC.Send

Purpose: Send data to the peer client/server. Not valid for a *Command*-mode server (see *Section A.24* for sending a response from a *Command*-mode server).

Syntax: `rc clientname|connectionname.commandname|messagename ← DRC.Send {clientname[.commandname|.messagename]|connectionname} {data} [close]`

where:

- `rc` is the return code (see *Section A.1*)
- `clientname[.commandname]|connectionname` is dependent on the mode and whether a full name is supplied or auto-generation is required:

- for client objects and server objects in *Text* mode or *Raw* mode:
  - if `clientname` is supplied, Conga auto-generates a `messagename` and returns it as the second element of the result in the format `clientname.messagename`.
  - if `connectionname` is supplied, Conga auto-generates a `messagename` and returns it as the second element of the result in the format `connectionname.messagename`.
  - if `clientname.messagename` or `connectionname.messagename` is supplied, Conga returns it unaltered as the second element of the result.

  `messagename` is not the name of a message object.
- for client objects in *Command* mode:
  - if `clientname` is supplied, Conga auto-generates a `commandname` and returns it as the second element of the result in the format `clientname.commandname`.
  - if `clientname.commandname` is supplied, Conga returns it unaltered as the second element of the result.

- `data` is the array to send to the peer server/client object; its format is dependent on the mode:
  - *Command* mode (sending data) – any APL data array (including namespaces but not classes or instances of classes)
  - *Command* mode (sending a file) – not possible
  - *Text/BlkText* mode (sending data) – character strings comprising characters with Unicode code points less than 256. To transmit characters outside this range, Dyalog Ltd recommends that you either use UTF-8 character encoding (for information on this, see ⎕UCS in the *Dyalog APL Language Reference Guide*) or switch to *Raw* mode and convert the character string to the appropriate format (for example, by applying ⎕UCS).
  - *Raw/BlkRaw* mode (sending data) – a vector of integers in the range 0 to 255 (negative integers -128 to -1 are also accepted and mapped to 128-255).
  - *Text/BlkText/Raw/BlkRaw* mode (sending a file) – a 2-element vector comprising:

    `[1]` any data to be transmitted before the contents of the file

`[2]` the full path and name of the file to send

- *HTTP* mode (not upgraded to use the WebSocket protocol) – one of the following:
  - a character vector comprising the entire HTTP message
  - a 2-element vector where:
    - [1] an empty vector (that is, `''`)
    - [2] the name of a file to send (containing a complete, properly-formatted, HTTP message)
  - an array of up to 5 elements; the content of these elements depends on whether the message is a request or a response:

    When sending a request the elements are:

    - [1] HTTP method (for example, `'GET'` or `'POST'`)
    - [2] URL of the requested resource
    - [3] HTTP version (for example, `'HTTP/1.1'`)
    - [4] 2-column matrix of head name/value pairs
    - [5] Either a character vector representing the message body or a 2 or 3-element array of `('' 'filename' ['gzip|deflate'])` where `filename` is the name of a file to send (containing only the message body to be sent) and `gzip|deflate` indicates the acceptable optional compression schemes that can be applied when sending the file.

    When sending a response the elements are:

    - [1] HTTP version (for example, `'HTTP/1.1'`)
    - [2] HTTP status code (for example, `'200'`)
    - [3] HTTP status message (for example, `'OK'`)
    - [4] 2-column matrix of head name/value pairs
    - [5] Either a character vector representing the message body or a 2 or 3-element array of `('' 'filename' ['gzip|deflate'])` where `filename` is the name of a file to send (containing only the message body to be sent) and `gzip|deflate` is the compression scheme applied when sending the file – this must be one that the request specified as being acceptable in the accept-encoding header.

- *HTTP* mode (upgraded to use the WebSocket protocol) – a 2- or 3-element vector, where:

  `[1]` is one of the following:

  - when sending data – a character or integer vector of the data to be transmitted
  - when sending a file – a 2-element vector comprising:

    `[1]` any data to be transmitted before the contents of the file

    `[2]` the full path and name of the file to send

  `[2]` is the FIN flag, a Boolean that declares whether this is the final fragment in a sequence. Possible values are:

  - 0 indicates that this is not the final fragment in a sequence
  - 1 indicates that this is the final (or only) fragment in a sequence

  `[3]` (optional) specifies the "operation code" (if an operation code is not specified then it is inferred from the data). Possible values are:

  - 1 for Text (data must be character and will be converted to UTF-8)
  - 2 for Binary (values between -128 and +255)
  - 0 for a continuation (the data must have the same type as your earlier transmission)

- `close` indicates the action to take after sending the data to the peer object. Possible values are:
  - 0 : no action taken. This is the default value.
  - 1 : the connection will be closed.
  - 2 : the connection remains open, the command object will be closed – only relevant for client objects in *Command* mode

    3 : generate a receipt (a `Send` event) on completion of the transmission of the `data`

EXAMPLES

A client object in Command mode:

To create a command with an auto-generated name below client `C1` and send an APL array to the server:

```
DRC.Send 'C1' ('PlusReduce' (⍳10))
```

A server object in Command mode:

Not applicable – server objects in *Command* mode use the `DRC.Respond` function rather than the `DRC.Send` function (see *Section A.24*).

A server or client object in Raw mode or Text mode:

To send the text 'Bye' on client `C1` and subsequently close the connection:

```
DRC.Send 'C1' ('Bye', ⎕UCS 13) 1
```

A server or client object in HTTP mode :

To send the file **foo.txt** (containing only the message body to be sent) using the gzip compression scheme:

```
DRC.Send obj ('HTTP/1.1' 200 'OK' headers ('' '/foo.txt'
                                           'gzip'))
```

To send the file **sample.txt** (containing a complete, properly-formatted, HTTP message) :

```
DRC.Send obj ('' '/sample.txt')
```

A server or client object sending a file after being upgraded to use the WebSocket protocol:

```
DRC.Send 'ws1' (('' 'c:\tmp\foo.txt') 1 1)
```

Related functions:

- `DRC.Progress` – see *Section A.19*
- `DRC.Respond` – see *Section A.24*
- `DRC.Wait` – see *Section A.32*

A.27 Function: DRC.ServerAuth



Integrated Windows Authentication is only available on a Microsoft Windows domain – both client and server must be running on Microsoft Windows.

Purpose: Performs server-side Integrated Windows Authentication (IWA).

Syntax: `rc ← DRC.ServerAuth {connectionname}`

where:

- `rc` is the return code (see *Section A.1*)
- `connectionname` is the name of the connection through which a *Command*-mode server responds to a *Command*-mode client.



`DRC.ClientAuth` (see *Section A.8*) and `DRC.ServerAuth` must be run at the same time.

A.28 Function: DRC.SetProp

Purpose: Updates the qualified properties of the specified Conga object.

Syntax: `DRC.SetProp {objectname} {property} {value}`

where:

- `objectname` is the Conga object for which to set a new property value
- `property` is the name of the property to set
- `value` is the value to set the specified property to

The property values that can be set depend on the type of the specified Conga object; they are described in *Appendix A.2*. Some of these values can also be specified when a server/client is created using the `DRC.Srv`/`DRC.Clt` function (see *Section A.29* and *Section A.10* respectively). The current values can be retrieved using the `DRC.GetProp` function (see *Section A.15*).

EXAMPLES

```
      DRC.SetProp '.' 'RootCertDir' 'C:\..\TestCertificates\ca'
0

      DRC.SetProp 'C1' 'KeepAlive' (1000 2000)
0
```

Related functions:

- `DRC.GetProp` – see *Section A.15*

A.29 Function: DRC.Srv

Purpose: Creates a Conga server to listen on a specified port. If certificate information is provided, then a secure server is created.

Syntax: `rc name ← DRC.Srv {Name} [Address] [Port] [Mode] [BufferSize] [SSLValidation] [EOM] [IgnoreCase] [Protocol] [PublicCertData] [PrivateKeyFile] [PrivateKeyPass] [PublicCertFile] [PublicCertPass] [PrivateKeyData] [Priority] [Magic] [X509] [AllowEndPoints] [DenyEndPoints] [CompLevel] [Options]`

where:

- `rc` is the return code (see *Section A.1*)
- `name` is the name of the Conga server that has been created. If no `Name` was specified (`''`) then this is auto-generated.
- `Name` is the name of the Conga server to create. If `''` is specified rather than a specific name, then the name will be auto-generated.
- `Address` is the address of the server
- `Port` is the port number that the server will listen on
- `Mode` is the connection protocol (see *Section 4.1.3*). Possible values are *Command*, *Text*, *Raw*, *BlkText* or *BlkRaw*. The default is *Command*.
- `BufferSize` is the maximum size (in bytes) allocated to the buffer that receives data transmissions. The default is 16,384. Only valid for clients in *Raw*/*Text*/*BlkRaw*/*BlkText* mode, not those in *Command* mode.
- `SSLValidation` is the sum of the relevant TLS flags (see *Appendix C*). Only valid when creating a secure client (see *Section 5.3*).
- `EOM` is a simple character vector or a vector of vectors indicating the termination string(s) (see *Section 4.1.3*). Only valid for clients in *Raw*/*Text* mode, not those in *BlkRaw*/*BlkText*/*Command* mode.

- `IgnoreCase` is a Boolean indicating whether searches for the termination string defined in EOM are case sensitive. Possible values are:
    - 0 : do not ignore case when searching for termination strings
    - 1 : ignore case when searching for termination strings

  Only valid for clients in *Raw*/*Text* mode, not those in *BlkRaw*/*BlkText*/*Command* mode
- `Protocol` is the communication protocol to use. Possible values are:
    - IPv4 : use the IPv4 connection protocol; if this is not possible then generate an error
    - IPv6 : use the IPv6 connection protocol; if this is not possible then generate an error
    - IP : use the IPv6 connection protocol; if this is not possible then use the IPv4 connection protocol.

  The default is to inherit the protocol defined for the root object.
- `PublicCertData` has been deprecated but is retained for backwards compatibility. Only valid when creating a secure server (see *Section 5.4*).
- `PrivateKeyFile` has been deprecated but is retained for backwards compatibility. Only valid when creating a secure server (see *Section 5.4*).
- `PrivateKeyPass` has been deprecated but is retained for backwards compatibility. Only valid when creating a secure server (see *Section 5.4*).
- `PublicCertFile` has been deprecated but is retained for backwards compatibility. Only valid when creating a secure server (see *Section 5.4*).
- `PublicCertPass` has been deprecated but is retained for backwards compatibility. Only valid when creating a secure server (see *Section 5.4*).
- `PrivateKeyData` has been deprecated but is retained for backwards compatibility. Only valid when creating a secure server (see *Section 5.4*).
- `Priority` is the GnuTLS priority string (for complete documentation of this, see http://www.gnutls.org/manual/gnutls.html#Priority-Strings).
- `Magic` is a 32-bit integer used to check that the data has not been corrupted. Only valid for servers in *BlkRaw*/*BlkText* mode, not those in *Raw*/*Text*/*Command* mode.
- `X509` is a reference to an instance of the X509Cert class. Only valid when creating a secure server (see *Section 5.4*).

- `AllowEndPoints` is a range or set of ranges of addresses that are automatically granted connections without validation. Each set of ranges is specified with a `/` character as the separator. Ranges can be specified using the IPv4 and/or IPv6 connection protocol, and any number of ranges can be specified with a `,` character as the separator. Any addresses that are also included in the range specified for `DenyEndPoints` will be denied connection (see *Section 9.5*).
- `DenyEndPoints` is a range or set of ranges of addresses that are automatically denied connections without validation. Each set of ranges is specified with a `/` character as the separator. Ranges can be specified using the IPv4 and/or IPv6 connection protocol, and any number of ranges can be specified with a `,` character as the separator (see *Section 9.5*).
- `CompLevel` is the level of compression to use when sending data in *Command* mode or a file in *HTTP* mode (see *Section 9.3*).
- `Options` is as integer representing the additive values of the server's parameters (see *Section 9.8*).

The `[PublicCertData]`, `[PrivateKeyFile]`, `[PrivateKeyPass]`, `[PublicCertFile]`, `[PublicCertPass]` and `[PrivateKeyData]` arguments are mutally exclusive with the `[X509]` argument.

EXAMPLES

To create APLRPC, a *Command* mode server listening on port 5050:

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

To create a secure *Command* mode server on port 5050 of the local machine using the named certificate/key files and a TLS flag value of 64 (RequestClientCertificate):

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

To create a *Text* mode server (with an auto-generated name) listening on port 23, with a maximum buffer size of 1000 characters and a termination sequence of <CR>:

```
      DRC.Srv '' '' 23 'Text' 1000 ('EOM' (⎕UCS 13))
0  SRV00000000
```

Related function:

- `DRC.Clt` – see *Section A.10*

A.30 Function: DRC.Tree

Purpose: Returns information about the specified Conga object and all of its first generation children (or all existing Conga objects if the specified object is root).

Syntax: `rc tree ← DRC.Tree {objectname}`

where:

- `rc` is the return code (see *Section A.1*)
- `tree` is a 2-element vector in which the first element describes the specified object and the second element is a vector of trees describing each of its first-generation children (the second element is empty if the specified object has no children and it contains all existing Conga objects if the specified object is root).
- `objectname` is the name of the Conga object to describe.

For all objects, the description starts with the following elements:

- `[1]` is the name of the object
- `[2]` is the object's type (see *Section 4.1.1*)
- `[3]` is the object's state (see *Section 4.1.2*)

Some types of Conga object also include additional elements:

- Conga objects of type 0 (root):
  - `[4]` is the version of Conga
  - `[5]` is the thread count
- Conga objects of type 4 (commands) or 5 (messages):
  - `[4]` is the size of the object that has been processed so far (in bytes)
  - `[5]` is the size of the object that has not yet been processed (in bytes)

EXAMPLE

```
      DRC.Srv 'S1' '' 5000
0  S1

      DRC.Clt 'C1' 'localhost' 5000
0  C1

      DRC.Wait  'S1' 100
0  S1.CON00000000  Connect  0

      (rc (root subtree)) ← DRC.Tree '.'
```

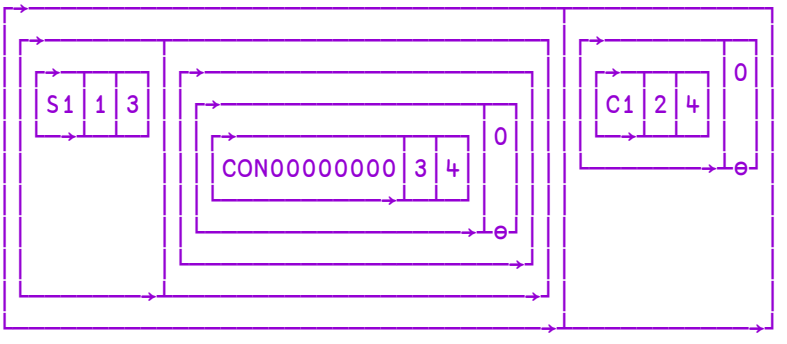
```
]Disp root
```



This elements in this result indicate that:

- `[1]` : The root object has no name (element is empty)
- `[2]` : The object type code is 0 (Root)
- `[3]` : The object state code is 2 (RootInit)
- `[4]` : The version number (and additional information) is available
- `[5]` : The number of semaphores currently in use for thread synchronisation is 1

```
]Disp subtree
```



The elements in this result indicate that there are two first-level children of the root:

- Conga object S1:
  - its object type code is 1 (server)
  - it is in state 3 (Listen)
  - it has a child object, CON00000000:
    - its object type is 3 (Connection)
    - it is in state 4 (Connected)
- Conga object C1:
  - its object type code is 2 (client)
  - it is in state 4 (Connected)

Similar functions:

- `DRC.Describe` – see *Section A.12*
- `DRC.Names` – see *Section A.17*

A.31 Function: DRC.Version

Purpose: Returns the Conga version number.

Syntax: `ver ← DRC.Version`

where:

- `ver` is a 3-element vector in which:
  - `[1]` is the major version number
  - `[2]` is the minor version number
  - `[3]` is the build number

EXAMPLE

```
      DRC.Version
3 5 1642
```

A.32 Function: DRC.Wait

Purpose: Waits for an event to occur.

Syntax: `rc objectname event data ← DRC.Wait {clientname|servername|connectionname|commandname} [timeout]`

where:

- `rc` is the return code (a return code of 100 indicates a timeout, for other return codes see *Section A.1*)
- `objectname` is the name of the object on which the event occurred.
- `event` is the name of the event that occurred– see *Table A-3*
- `data` is the received data.
- `clientname|servername|connectionname|commandname` is the name of the object waiting for an event:
  - if a `servername` or `connectionname` name is specified, the `DRC.Wait` function will report events on the named object or any of its children.
  - If a *Command*-mode client is waiting on a specific command, the full `commandname` can be specified.
- `timeout` is the number of ms to wait before timing out. The default is 1,000.

Table A-3: *Types of event that can occur*

Event Name	Description
`Block`	*Text* or *Raw* mode only: A block of data was received and the connection is still open.
`BlockLast`	*Text* or *Raw* mode only: A block of data was received and the connection was closed; no more data is expected. If the connection is closed while it is inactive, a `BlockLast` event will be reported with empty data.
`Closed`	The socket has been closed. Only valid when `EventMode` is set to 1 (see *Section A.28*).
`Connect`	The Conga object has been created but has not yet participated in any data transmissions.
`Error`	An error occurred.
`HTTPBody`	*HTTP* mode only: The body of an HTTP message (representing the end of an HTTP message) was received. Only valid when receiving an HTTP message with the Transfer-Encoding header not set to 'chunked' and a Content-Length header that is not 0.
`HTTPChunk`	*HTTP* mode only: A chunk was received. Only valid when receiving an HTTP message with the Transfer-Encoding header set to 'chunked'.
`HTTPHeader`	*HTTP* mode only: an HTTP header (the start of an HTTP message) was received.
`HTTPTrailer`	*HTTP* mode only: an HTTP trailer (representing the end of an HTTP message) was received. Only valid when receiving an HTTP message with the Transfer-Encoding header set to 'chunked'.
`Progress`	*Command*-mode client only: The server transmitted data using the `DRC.Progress` function.
`Receive`	*Command* mode only: Data has been received.

Table A-3: *Types of event that can occur (continued)*

Event Name	Description
`Sent`	The transmission of data has been completed. Only valid when the `DRC.Send` function's `close` parameter is set to 3 (see *Section A.26*).
`Timeout`	The `DRC.Wait` function has timed out. Only valid when `EventMode` is set to `1` (see *Section A.28*).
`WSReceive`	*HTTP* mode only: Data has been received using the WebSocket protocol.
`WSResponse`	*HTTP*-mode client only: The server has accepted a request from the client to upgrade the connection to use the WebSocket protocol. Only valid when `WSFeatures` on the client is set to `0` (see *Section A.28*).
`WSUpgrade`	*HTTP* mode only: The client has requested that the connection is upgraded to use the WebSocket protocol or the server has accepted a request from the client to upgrade the connection to use the WebSocket protocol. Only valid when `WSFeatures` is set to `1` (see *Section A.28*).
`WSUpgradeReq`	*HTTP*-mode server only: The client has requested that the connection is upgraded to use the WebSocket protocol. Only valid when `WSFeatures` on the server is set to `0` (see *Section A.28*).

EXAMPLES

```
      DRC.Srv 'S1' '' 5000
0  S1

      DRC.Clt 'C1' 'localhost' 5000
0  C1

      DRC.Wait 'S1' 5000
0  S1.CON00000000  Connect  0

      DRC.Send 'C1.fakename' 'Testing'
0  C1.fakename

      DRC.Wait 'S1' 5000
0  S1.CON00000000.fakename  Receive  Testing
```

In *Command* mode, command names are carried over to the server.

Related functions:

- `DRC.Progress` – see *Section A.19*
- `DRC.Respond` – see *Section A.24*
- `DRC.Send` – see *Section A.26*

A.33 Class: DRC.X509Cert

Purpose: Provides a container to encapsulate X.509-style certificates. Dyalog Ltd recommends that this class is used when providing secure communications.

`#.DRC.X509Cert.[X509Cert]` is an instance of the X509Cert class.

Syntax – Shared Methods: These read certificates from various sources; they are not instance-specific.

These shared methods have been superseded by functions of the same name but are included in this document for backwards-compatibility purposes. Dyalog Ltd recommends using the following:

- `DRC.ReadCertFromFile` (see *Section A.20*) instead of `DRC.X509Cert.ReadCertFromFile`
- `DRC.ReadCertFromFolder` (see *Section A.21*) instead of `DRC.X509Cert.ReadCertFromFolder`
- `DRC.ReadCertFromStore` (see *Section A.22*) instead of `DRC.X509Cert.ReadCertFromStore`
- `DRC.ReadCertURLs` (see *Section A.23*) instead of `DRC.X509Cert.ReadCertURLs`

To return certificate instances of all the certificates in a specified file:

```
certs ← DRC.X509Cert.ReadCertFromFile {filename}
```

To return certificate instances of all the certificates in a specified directory that match the specified pattern:

```
certs ← DRC.X509Cert.ReadCertFromFolder {pathname}
```

To return certificate instances of all the certificates in "My" certificate store:

```
certs ← DRC.X509Cert.ReadCertUrls
```

where:

- `certs` is a vector of certificate instances where each element is of type DRC.X509Cert.
- `filename` is a certificate file name as a character vector, for example, `'server-cert.pem'`. Although it is a single file name, the file can contain multiple certificates.
- `pathname` is a character vector specifying the path to the directory that contains certificate files. It can be fully-qualified or relative to the current working directory. Wildcard characters can be used, for example, **'c:\mycerts*.pem'**, although if files match the pattern but are not valid certificate files then `certs` will be an empty vector.

Syntax – Instance Methods: These act on specific certificate instances.

To return the certicate's `PublicCertFile` and `PrivateKeyFile` parameters in a format suitable for use with `DRC.Clt` and `DRC.Srv`:

```
parms ← #.DRC.X509Cert.[X509Cert].AsArg
```

To identify the keys that the certificate has access to:

```
keys ← #.DRC.X509Cert.[X509Cert].IsCert
```

To return the certificate chain for the certificate:

```
certs ← #.DRC.X509Cert.[X509Cert].Chain
```

To save the certificate instance to a file:

```
result ← [name] #.DRC.X509Cert.[X509Cert].Save path
```

To follow the certificate chain from the current certificate until a root certificate is found and, if possible, updates the `DRC.GetProp` function's `PeerCert` property so that the certificate chain is complete:

```
chain ← #.DRC.X509Cert.[X509Cert].CopyCertificateChainFromStore
```

where:

- `keys` indicates the keys that the certificate has access to (determines how it can be used):
  - 0 : the certificate does not have access to either a public key or a private key; connections will be anonymous
  - 1 : the certificate has access to a public key but not a private key; can be used to authenticate a certificate chain

    - 2: the certificate has access to both a public key and a private key; can be used for full Conga functionality
- `certs` is a vector of certificate instances where each element is of type DRC.X509Cert
- `result` indicates whether the file saved successfully:
    - 0 : the file saved successfully
    - `EN` : the file did not save successfully
- `name` is the name under which to save the file. If not specified, a name is built from the instance's Subject.
- `path` is a character vector specifying the path to the directory in which to save the certificate file. It can be fully-qualified or relative to the current working directory.
- `chain` is the number of certificates in the chain (including the calling instance and the root certificate).

EXAMPLES

Identify the keys that `john` (an instance of the X509Cert class in the `Samples` namespace) has access to:

```
      Samples.john.IsCert
1  A access to public key but not private key
```

Return the issuer information for `john` (an instance of the X509Cert class in the `Samples` namespace):

```
      (Samples.john.Chain).Formatted.Issuer
O=Test CA,CN=Test CA
```

A.33.1 Instances of the DRC.X509Cert Class

Each instance of the `DRC.X509Cert` class has the properties detailed in *Table A-4*.

Table A-4: *Properties of each instance of the `DRC.X509Cert` class*

Property	Description
`Cert`	Integer vector of raw certificate data.

Table A-4: *Properties of each instance of the DRC.X509Cert class (continued)*

Property	Description

`CertOrigin`	For certificates read from a certificate store this is:
	`'MSStore' storename`
	For certificates read from a file this is:
	`'DER'` and a fully qualified filename
	For example:
	`'DER' C:\apps\dyalog141U64\TestCertificates\client\John Doe-cert.pem`

`Elements`
`Extended`
`Formatted`
	`Elements`, `Extended`, and `Formatted` are namespaces that 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 (`Extended` may, in some instances, contain additional information).

`KeyOrigin`	For keys read from a certificate store this is:
	`'MSStore' storename`
	For keys read from a file this is:
	`'DER'` and a fully qualified filename
	For example:
	`'DER' C:\apps\dyalog141U64\TestCertificates\client\John Doe-key.pem`

`LDRC`	*Internal reference to the local DRC namespace*.
`ParentCert`	An instance of the certificate directly above this one in the certificate chain. Only relevant if this certificate is part of a certificate chain but not at the top of the chain.
`UseMSStoreAPI`	Boolean indicating which API to use to decode certificate information. Possible values are: • 0 : Use the GnuTLS API • 1 Use the Microsoft certificate store API For applications that could be deployed on an operating system other than Microsoft Windows, the GnuTLS API should be used.

Not all certificates have values for all of the elements that are contained in the `Elements`, `Extended`, and `Formatted` properties, and some elements are more useful than others.*Table A-5* lists some of the more useful of the possible elements (it is not a comprehensive reference of X.509 certificate structure).

Table A-5: *Some of the elements that can comprise the `Elements`, `Extended`, and `Formatted` properties of each instance of the `DRC.X509Cert` class*

Property	Description
AlgorithmID	The cryptographic algorithm used to generate the signature, for example, RSA-SHA1 and DSA-SIGN.
Description	A text description of the certificate.
Issuer	The issuer of the certificate. Useful when validating certificate chains (the Issuer of a certificate should match the Subject of its parent certificate). Self-signed certificates have identical Issuer and Subject elements.
Key	The certificate's key in Boolean format.
KeyHex	The certificate's key in hexadecimal format.
KeyID	The certificate's key's cryptosystem, for example, RSA or DHE.
KeyLength	The length of the certificate's key (in bits). Maximum value is 16,384.
SerialNo	A number that uniquely identifies the certificate and is issued by the certification authority.
Subject	The subject of the certificate. Useful when validating certificate chains (the Subject of a certificate should match the Issuer of its child certificate). Self-signed certificates have identical Issuer and Subject elements.
ValidFrom ValidTo	Together, these two elements define the period of validity for the certificate.
Version	The version of the X.509 standard applied when creating the certificate (currently this is 3).

B Certificates

Many different file formats can be used for storing X.509 certificates, including PEM, DER, PFX, P7C and P12; the popularity of these formats varies between platforms. Conga supports the PEM and DER format files on all platforms; these have file extensions **.pem** and **.der** respectively. Certificates in other formats can be used after they have been converted to PEM or DER format files; this conversion can be performed using open source tools such as GnuTLS and OpenSSL (see the guide at http://gagravarr.org/writing/openssl-certs/general.shtml for information on converting between formats using OpenSSL).

B.1 PEM File Format

Files that have the PEM file format start with:

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

and end with:

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

They contain a base64-encoded version of the certificates and do not include any control characters.

The secure communications library GnuTLS (http://www.gnu.org/software/gnutls/) comes with a command line tool called **certtool** that can be used for creating certificates, keys and certificate requests as **.pem** files. It is documented at http://www.gnu.org/software/gnutls/manual/html_node/Invoking-certtool.html.

B.2 Generating Certificates and Keys



This is not supported on the AIX operating system.

The following example creates a set of certificates similar to the test certificates provided in the **[DYALOG]/TestCertificates** directory. This example is operating-system-independent and uses the GnuTLS open source secure communications library (see *Section B.1*).

The `CertTool` namespace includes code that checks whether **certtool.exe** exists in the location specified by `EXEC`. It then checks the version number of **certtool.exe** and signals a DOMAIN ERROR if it is less than 3.4.0 (version 3.4.0 introduced support for PKCS#7 and PKCS#12 encoded certificates).

To create a set of example certificates

1. Download and unzip the latest version of the GnuTLS secure communications library from ftp://ftp.gnutls.org/gcrypt/gnutls/.

   Download and unzip the latest version of the GnuTLS secure communications library from your distribution's repository rather than using the above link.

2. Start a Dyalog Session and load the `CertTool` namespace into the workspace. For example:
   ```
   ]LOAD [DYALOG]/Samples/Conga/CertTool/CertTool.dyalog
   ```
3. Edit the values of the following names in the `CertTool` namespace:

EXEC	(in the `Init` function, under the appropriate operating system) Fully-qualified path to the **certtool.exe** file in the **bin** directory of the unzipped GnuTLS secure communications library.  The **certtool.exe** file is assumed to be on the path (this is true if it has been installed from the distribution's installation media or repositories). For example: `'c:\apps\gnutls-3.4.7\bin\certtool.exe'`

TARGET	(in the `Init` function, under the appropriate operating system) Fully-qualified path to the directory in which to store the generated certificates and files. For example: `'c:\temp\TestCertificates\'`
SERIAL	(in the `Init` function) Serial number of the first certificate generated. The other four certificates that are generated are assigned numbers related to this using the formula `SERIAL+7×¯1+⍳5` (the `7` can be changed in the `CommonAttr` function, line `[3]`). For example: `100`
C	(in the `Init` function) Country in which the certificate is to be produced. This will be included in the subject information of the generated certificates. For example: `UK`
O	(in the `Init` function) Organisation that is producing the certificate. This will be included in the subject information of the generated certificates. For example: `DyalogLtd`
OU	(in the `Init` function) Department within the Organisation that is producing the certificate. This will be included in the subject information of the generated certificates. For example: `Test`
ST	(in the `Init` function) State/county/district in which the Organisation is located. This will be included in the subject information of the generated certificates. For example: `Hampshire`

CN	(in the `Examples` function) The common name used on the certificate. For client certificates this is usually the name of a client or person. For example: `Ken Iverson` For server certificates this is usually the DNS name of the server. For example: `www.dyalog.com`

4. Run the `CertTool.Examples` function.
   The directory specified by TARGET is created and populated with further directories and files.

The generated output is as follows:

- **CA** directory:
  - **ca-cert.pem**
    The public certificate for the example CA. Used to authenticate client/server certificates.
  - **caconf.cfg**
    Information about the CA certificate's properties.
  - **ca-key.pem**
    The private key for the example CA. Used to sign client/server and CA certificates.
- **client** directory – four files for each `ClientCert` name defined in the `Examples` function. In this example, these are `John Doe` and `Jane Doe` (see lines `[17]` and `[18]` in the `Examples` function):
  - **<name>.cer**
    A password-encrypted ASCII file containing the client's key and certificate in PKCS #7 file format.
  - **<name>.p12**
    A binary file containing the client's key and certificate in PKCS #12 file format.
  - **<name>-cert.pem**
    With **<name>-key.pem**, forms a client certificate's certificate/key pair.
  - **<name>-key.pem**
    With **<name>-cert.pem**, forms a client certificate's certificate/key pair.
- **server** directory – four files for each `ServerCert` name defined in the `Examples` function. In this example, these are `localhost` and `myserver` (see lines `[15]` and `[16]` in the `Examples` function):
  - **<name>.cer**
    A password-encrypted ASCII file containing the server's key and

certificate in PKCS #7 file format.

- **<name>.p12**
  A binary file containing the server's key and certificate in PKCS #12 file format.
- **<name>-cert.pem**
  With **localhost-key.pem**, forms a server certificate's certificate/key pair.
- **<name>-key.pem**
  With **localhost-cert.pem**, forms a server certificate's certificate/key pair.

C TLS Flags

TLS flags are employed as part of the certificate checking process; they determine whether a secure client or server can connect with a peer that does not have a valid certificate.

The code numbers of the TLS flags described in *Table C-1* can be added together and passed to the `DRC.Clt` / `DRC.Srv` functions to control the certificate checking process. If you do not require any of these flags, then the `SSLValidation` parameter of these functions should be set to 0.

Table C-1: *TLS Flags*

Code	Name	Description
1	CertAcceptIfIssuerUnknown	Accept the peer certificate even if the issuer (root certificate) cannot be found.
2	CertAcceptIfSignerNotCA	Accept the peer certificate even if it has been signed by a certificate not in the trusted root certificates' directory.
4	CertAcceptIfNotActivated	Accept the peer certificate even if it is not yet valid (according to its *valid from* information).
8	CertAcceptIfExpired	Accept the peer certificate even if it has expired (according to its *valid to* information).
16	CertAcceptIfIncorrectHostName	Accept the peer certificate even if its hostname does not match the one it was trying to connect to.

Table C-1: *TLS Flags (continued)*

Code	Name	Description
32	CertAcceptWithoutValidating	Accept the peer certificate without checking it (useful if the certificate is to be checked manually – see *Section A.15*).
64	RequestClientCertificate	Only valid for a server; asks the client for a certificate but allows connections even if the client does not provide one.
128	RequireClientCertificate	Only valid for a server; 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.

TLS flags have the same meanings for a server as for a client. However, for a server they are applied each time a new connection is established whereas for a client they are only applied when the client object is created.

D Conga Libraries

If an application that includes the `conga` workspace is shipped, then the relevant libraries will also need to be shipped. The libraries depend on the interpreter that is shipped with the application. *Table D-1* shows the necessary libraries for each platform ("<bits>" indicates the width – possible values are 64 or 32).

Table D-1: *Conga libraries for supported platforms*

Platform	Conga Shared Library	Conga SSL Shared Library
AIX	conga37_<bits>.so	libconga37ssl<bits>.so
Linux	conga37_<bits>.so	libconga37ssl<bits>.so
macOS	conga37_<bits>.dylib	libconga37ssl<bits>.dylib
Microsoft Windows	conga37_<bits>.dll	libconga37ssl<bits>.dll
Raspberry Pi OS	conga37_<bits>.so	libconga37ssl<bits>.so

E Error Codes

Errors can be signalled at several levels within the Conga framework, including from the operating system, the Conga shared library or the GnuTLS library and within the APL coded portion of Conga.

If an error is generated when running Conga, then more information on that error can be obtained by entering:

```
DRC.Error {errorcode}
```

in the Dyalog Session.

Table E-1 details some of the errors that can be encountered and provides possible resolutions.

Table E-1: *Possible error codes returned by Conga*

Code	Source	Reason for Error	Possible Resolution
13	UNIX	Attempted to allocate a port with number less than 1025 without having root permission.	Either allocate a port number above 1024 or sign on as root (see `4001⌶` in the *Dyalog APL Language Reference Guide*).
98	UNIX	Specified port number is already in use.	Allocate a different port number (it can take several minutes to de-allocate a port before it can be reused).
100	Conga	Timeout – nothing was received within the specified timeout period.	This is a normal occurrence and should be accommodated for in the client/server code.

Table E-1: *Possible error codes returned by Conga (continued)*

Code	Source	Reason for Error	Possible Resolution

1006	Conga	The root object could not be found.	If using the DRC namespace, try:
			`⍎1 DRC.Init ''`
			If using the Conga namespace, try:
			`root←Conga.Init ''`

1105	Conga	Could not receive data.	Re-establish the connection.
1119	Conga	Socket closed while receiving data (occurs when the connection is broken mid-block transfer).	Reconnect and resend the data.
1135	Conga	Maximum block size (as defined by the `BufferSize` parameter of the `DRC.Clt/DRC.Srv` function) exceeded when attempting to send/receive data.	Increase the value of the `BufferSize` parameter or chunk the data into smaller blocks.
1146	Conga	An HTTP message with a size that exceeded the maximum defined by the `DOSLimit` property was received .	This could be an indication of a denial-of-service (DOS) attack. Investigate and if the cause is not a DOS attack then increase the value of the `DOSLimit` property.
1201	TLS	The handshake process that sets up a secure connection between the client and server before the certificates are exchanged is failing.	Ensure that the client and server are using the same encryption protocol and that both are using SSL/TLS.

Table E-1: *Possible error codes returned by Conga (continued)*

Code	Source	Reason for Error	Possible Resolution
1202	TLS	The certificate supplied by the peer is not valid.	Supply the TLS flag CertAcceptWithoutValidating to the `DRC.Srv`/`DRC.Clt` function (see *Appendix C*) to allow this connection and examine the certificate manually.
1203	TLS	One or more of the specified certificate files could not be loaded (either the file does not exist, it cannot be read or it is not a valid certificate file).	Ensure that the filenames being passed to the `DRC.Clt` and `DRC.Srv` functions are correct, that the files exist and that they are valid certificate files.
1204	TLS	There was an error setting up the TLS libraries.	Ensure that all GnuTLS files are present and valid.

F Change History

This appendix details the changes made at each version of Conga since the release of Conga version 2.0.

F.1 Version 3.7

Released with Dyalog version 21.0.

This version:

- adds a new Conga object property, `NoDelay`, on the connection object; this specifies whether Nagel's algorithm is enabled.

F.2 Version 3.6

Released with Dyalog version 20.0.

This version:

- adds a new argument – `libpath` – to the functions used when initialising Conga. This is optional when calling the `Conga.Init` function, but mandatory when calling the `DRC.Init`. function.

F.3 Version 3.5

Released with Dyalog version 19.0.

This version:

- adds a new property to the `DRC.SetProp` function – `StartTLS` – to support upgrading an non-secure proxy connection to a secure end host (client and connection object types only).

F.4 Version 3.4

Released with Dyalog version 18.2.

This version:

- adds features to improve protection against denial-of-service (DOS) attacks when running in HTTP mode. Specifically:
  - a new property to the `DRC.SetProp` and `DRC.GetProp` functions – `DOSLimit` (root object types only).
  - a new named enumeration for the `Options` parameter – `EnableBufferSizeHttp`.
  - a new error code – 1146.
- reintroduces support for First-In-First-Out mode. Specifically:
  - a new property to the `DRC.SetProp` and `DRC.GetProp` functions – `FifoMode` (server object types only).
  - a new named enumeration for the `Options` parameter – `EnableFifo`
  - two new events – `HTTPFail` and `HTTPError`.

F.5 Version 3.3

Released with Dyalog version 18.0.

This version:

- adds two new properties to the `DRC.GetProp` function – `CompLevel` (root, client, server and connection object types only) and `Options` (client, server and connection object types only).
- adds one new property to the `DRC.SetProp` function – `CompLevel` (root, client, server and connection object types only) and `Options` (client, server and connection object types only).
- adds two new parameters to the `DRC.Srv` and `DRC.Clt` functions – `CompLevel` and `Options`.
- adds a new namespace, `DRC.Options`, which contains named enumerations for the `Options` parameter.
- adds a new function, `DRC.DecodeOptions`, which translates a numeric options setting into a character description using the named enumerations from the `Options` namespace.
- extends the datatypes that can be sent by in *Raw* and *BlkRaw* mode to include single-byte characters (type 80 or 82).

- modifies the allowed range of numeric values that can be sent using a WebSocket to include both signed integer (-128 to 127) and unsigned integer (0 to 255).
- deprecates the `DecodeBuffers` property of HTTP events and limits its possible values to 0 or 15 (any other value will generate an error).

F.6 Version 3.2

Released with Dyalog version 17.1.

This version:

- removes samples and utilities from the `conga` workspace.

  This removed functionality is available elsewhere, for example, the functionality provided by `Samples.HTTPGet`, `Samples.HTTPReq`, `Samples.HTTPPost`, `Samples.HTTPCmd` and `HTTPUtils` is now accessible through **[DYALOG]/Library/Conga/HttpCommand**, which can be loaded using `]Load HttpCommand`.
- removes support for the `FIFOMode` server-only property.
- adds the ability to set the compression level when directly transmitting a file.
- adds a new `ErrorText` property to the `DRC.GetProp` function (root object types only).

F.7 Version 3.1

Released with Dyalog version 17.0.

This version:

- changes the internal format of wide floating-point numbers from DPD to BID.

F.8 Version 3.0

Released with Dyalog version 16.0.

This version:

- adds:
  - four new functions to read certificates and return them as instances of the X509Cert class – `DRC.ReadCertFromFile`, `DRC.ReadCertFromFolder`, `DRC.ReadCertFromStore`, and `DRC.ReadCertURLs`. These supercede the four methods of the same

names in the `DRC.X509Cert` class (these are still included for backwards compatibility, but Dyalog Ltd recommends using the new functions).
    - two new server-only properties – `FIFOMode` and `ConnectionOnly`
    - integrated support for HTTP, including four new events (`HTTPBody`, `HTTPChunk`, `HTTPHeader` and `HTTPTrailer`), a new connection mode (*HTTP*), messages and utility libraries
    - a `Sent` event to enable the management of data buffers when sending large amounts of data
    - `Timeout` and `Closed` events (as an alternative to error codes 100 and 1119)
    - a new `Conga.Magic` function that generates the `Magic` property value
    - two new parameters to the `DRC.Srv` function – `AllowEndPoints` and `DenyEndPoints`
    - two new properties to the `DRC.GetProp` function – `HttpDate` (root object types only) and `HostName` (server object types only).
    - a new option, *3*, to the `DRC.Send` function's `close` parameter.
    - dynamic loading of secure socket support
    - numerous new samples and utilities in GitHub repositories (https://github.com/Dyalog/samples-conga and https://github.com/Dyalog/library-conga respectively)
- adds support for:
    - multiple isolated Conga root objects
    - WebSockets, enabling bi-directional asynchronous data transmission, including four new events (`WSReceive`, `WSResponse`, `WSUpgrade` and `WSUpgradeReq`)
    - UDP (User Datagram Protocol) – *experimental*
    - GnuTLS 3.5.16
- adds the ability to:
    - temporarily prevent new connections by setting the new Pause property
    - transmit files directly without having to first read the data into the workspace
    - permit or deny connections from specific address ranges
- simplifies configuration by establishing a default location for the Conga shared libraries
- replaces support for the deflate compression scheme with compression levels 0-9 (new property `CompLevel`)
- removes or relocates seldom-used or atrophied samples

F.9 Version 2.7

Released with Dyalog version 15.0.

This version:

- adds a new namespace, `CertTools`; this can be used to generate certificates.
- removes the obsolete `TelnetServer` and `TelnetClient` classes from the `conga` workspace (the associated `TestTelnetServer` and `TestSecureTelnetServer` functions and `Parser` utility in the `Samples` namespace are also removed).
- merges the `WebServer.HttpsRun` method into the `WebServer.Run` method.
- allows an empty left argument to be supplied to the `Samples.HTTPGet` function.

F.10 Version 2.6

Released with Dyalog version 14.1.

This version:

- adds support for "blocked" raw and ASCII communications modes.
- adds a new `DRC.X509Cert.Save` method that saves the current certificate to file.
- adds a new strategy option (*3*) to the `DRC.GetProp` function's `ReadyStrategy` property; this selects the oldest connection but has improved performance over strategy option *2*.
- adds new `HTTPCmd` operator and `HTTPPost` function to the `Samples` namespace.
- removes the need to specify protocol IPv4 on machines that do not support IPv6; in this situation, IPv4 will be selected by default.

OS WIN In addition, this version:

- when using SSL/TLS, uses the Microsoft Windows "Trusted Root Certification Authorities" certificate store to verify system trust if the folder specified by the `DRC.GetProp` function's `RootCertDir` parameter contains no certificates
- adds a new `DRC.X509Cert.CopyCertificationChainFromStore` method that follows the certificate chain from the current certificate until a root certificate is found and, if possible, updates the `DRC.GetProp` function's `PeerCert` property so that the certificate chain is complete.

F.11 Version 2.5

Released with Dyalog version 14.0.

This version:

- incorporates a new version of the GnuTLS library to provide secure communications using SSL/TLS – this addresses a bug (CVE-2014-0092) whereby attackers could bypass the SSL/TLS protections.

F.12 Version 2.4

An internal update incorporating features in support of Ride.

F.13 Version 2.3

Released with Dyalog version 13.2.

This version:

- adds a new `KeepAlive` property to the `DRC.GetProp` function; this causes a server to send periodic (heartbeat) messages to a client to determine whether the a connection is still live.

OS WIN In addition, this version:

- now supports Integrated Windows Authentication (IWA), using the domain credentials of a Windows user for authentication. Two new functions, `DRC.ClientAuth` and `DRC.ServerAuth` provide client and server side IWA capabilities respectively.

F.14 Version 2.2

Released with Dyalog version 13.1.

This version:

- adds a new `DRC.Version` function that returns the current version of Conga.
- adds a new `DRC.flate` class that implements the deflate compression scheme (one of several content encoding schemes used by all major web servers and browsers to optimise the flow of data across networks) using the zlib open source compression library (for more information on zlib, see http://zlib.net).
- adds a new option, *2*, to the `DRC.Send` function's `close` parameter; this sends 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 subsequently calling the `DRC.Respond` function.
- adds support for deflate HTTP compression in the `Samples.HTTPGet` function.
- enhances the `DRC.Describe` function to report the GnuTLS version.

F.15 Version 2.1

Released with Dyalog version 13.0.



Version 2.1 modifies how certificates are used to facilitate secure communications. Changes to the `DRC.Srv` and `DRC.Clt` functions when using certificates mean that Conga 2.0 applications that use certificates will require minor modification to use Conga 2.1.

This version:

- adds a new `DRC.X509Cert` class that encapsulates the structure and function necessary to use X.509 certificates with Conga. This is the recommended method for providing certificate information to the `DRC.Clt` and `DRC.Srv` functions.
- adds a new `DRC.Certs` function that provides the underlying functionality used by the `DRC.X509Cert` class to read and decode certificates.
- adds a new `PeerCert` property to the `DRC.GetProp` function; this returns an X509Cert object (certificate information).
- modifies the syntax used to pass certificate information to the `DRC.Srv` and `DRC.Clt` functions.

- adds a new strategy option (*-1*) to the `DRC.Init` function's `reset` parameter; this causes Conga to reload its underlying drivers.
- enhances the `Samples.HTTPGet` function to accept an X509Cert object as its (optional) left argument.
- enhances the `WebServer.HttpsRun` method to accept an X509Cert object argument.



In addition, this version:

- now reads/uses certificates located in Certificate Stores.

Index

B

BlkRaw mode 14
BlkText mode 13
Block events 13
BlockLast events 13

C

CAs See *Certificate authorities*
Certificate authorities 26
Certificate chains 34
Certificate revocation lists 28
Certificate stores 28
Classes
  DRC.X509Cert 106
  Notation when calling 63
Client-side WebSocket Upgrade . 42
Closed events 57
Command mode 15
Compatibility
  Between object modes 15
  Compression levels 55
  With Dyalog 5
Compression level 55
Conga object
  Modes 13
  Names 8
  Properties 64
  States 10
  Types 8
Conga object modes 13
  BlkRaw mode 14
  BlkText mode 13
  Command mode 15
  Compatibility 15
  HTTP mode 15, 36
  Raw mode 14
  Text mode 13
Conga object names 8
Conga object properties 64, 85
Conga object states 10
Conga object types 8
  Client 9
  Client-Server relationship 9
  Command 9
  Connection 9
  Message 9
  Root 9
  Server 9
Conga objects 8
conga workspace 50
Conga.Init (function) 73
Conga.Magic (function) 75
Conga.New (function) 75
Conga.RootNames (function) 76
Connect events 103

D

DRC.Certs (function) 77
DRC.ClientAuth (function) 78
DRC.Close (function) 79
DRC.Clt (function) 79
DRC.DecodeOptions (function) .. 82
DRC.Describe (function) 82
DRC.Error (function) 83
DRC.Exists (function) 84
DRC.GetProp (function) 84

DRC.Init (function) 85
DRC.Names (function) 86
DRC.Options (namespace) 87
DRC.Progress (function) 87
DRC.ReadCertFromFile (function) 88
DRC.ReadCertFromFolder (function) 89
DRC.ReadCertFromStore (function) 89
DRC.ReadCertUrls (function) 90
DRC.Respond (function) 91
DRC.RootName (function) 91
DRC.Send (function) 92
DRC.ServerAuth (function) 97
DRC.SetProp (function) 97
DRC.Srv (function) 98
DRC.Tree (function) 101
DRC.Version (function) 103
DRC.Wait (function) 103
DRC.X509Cert (class) 106

E

Error codes 119
Error events 103
Event types 104
Events
  Block 13
  BlockLast 13
  Closed 57
  Connect 103
  Error 103
  HTTPBody 39
  HTTPChuck 39
  HTTPError 40
  HTTPFail 40
  HTTPHeader 38
  HTTPTrailer 39
  Progress 103
  Receive 103
  Sent 58
  Timeout 57
  WSReceive 48
  WSResponse 44
  WSUpgrade 44
Events in HTTP mode 37

F

Functions
  Conga.Init 73
  Conga.Magic 75
  Conga.New 75
  Conga.RootNames 76
  DRC.Certs 77
  DRC.ClientAuth 78
  DRC.Close 79
  DRC.Clt 79
  DRC.DecodeOptions 82
  DRC.Describe 82
  DRC.Error 83
  DRC.Exists 84
  DRC.GetProp 84
  DRC.Init 85
  DRC.Names 86
  DRC.Progress 87
  DRC.ReadCertFromFile 88
  DRC.ReadCertFromFolder 89
  DRC.ReadCertFromStore 89
  DRC.ReadCertUrls 90
  DRC.Respond 91
  DRC.RootName 91
  DRC.Send 92
  DRC.ServerAuth 97
  DRC.SetProp 97
  DRC.Srv 98
  DRC.Tree 101
  DRC.Version 103
  DRC.Wait 103
  Notation when calling 63

H

HTTP mode 15, 36
  Events 37
  Limiting message size 40
  Receiving messages 36
  Sending messages 41
  WebSocket Protocol 42
HTTP Tunnelling 61
HTTPBody event 39
HTTPChunk event 39

HTTPError event 40
HTTPFail event 40
HTTPHeader event 38
HTTPTrailer event 39

I

Initialisation 6
Installation 5

L

Libraries 118
Limiting HTTP message size 40

M

Methods
Notation when calling 63
Multi-threading 23

N

Namespaces
Options 87

O

Operators
Notation when calling 63

P

Parallel commands 22
Progress events 103

R

Raw mode 14
Receive events 103
Receiving HTTP messages 36
Return codes 63
Root objects 16

S

Secure connections 25
Sending files 53
Sending HTTP messages 41
Sent events 58
Server-side WebSocket Upgrade 46

T

Text mode 13
Timeout events 57
TLS flags 116
Troubleshooting 119
Tunnelling 61

W

WebSocket protocol 42
Client-side upgrade 42
Server-side upgrade 46
WSReceive events 48
WSResponse events 44
WSUpgrade events 44
WSUpgradeReq events 44