# Contents

1 About This Document ........................................................................................................ 1
   1.1 Audience ................................................................................................................. 1
   1.2 Conventions .............................................................................................................. 1

2 Introduction ....................................................................................................................... 3

3 Installation ......................................................................................................................... 4
   3.1 Compatibility ............................................................................................................. 4
      3.1.1 Compatibility with Applications Using Conga Version 2.x ......................... 4
   3.2 Initialisation .............................................................................................................. 5

4 Getting Started ................................................................................................................. 7
   4.1 Conga Objects ......................................................................................................... 7
      4.1.1 Conga Object Types .......................................................................................... 7
      4.1.2 Conga Object States ......................................................................................... 9
      4.1.3 Conga Object Modes ....................................................................................... 11
   4.2 Root Objects ............................................................................................................. 14
   4.3 A Simple Conga Client ............................................................................................ 15
   4.4 A Simple Conga Server ........................................................................................... 17
   4.5 Command Mode ....................................................................................................... 18
   4.6 Parallel Commands .................................................................................................. 20
      4.6.1 Multi-threading ............................................................................................... 21

5 Secure Connections ............................................................................................................ 22
   5.1 CA Certificates ........................................................................................................ 23
   5.2 Client and Server Certificates .................................................................................. 24
      5.2.1 Certificate Stores ............................................................................................. 25
      5.2.2 Revocation Lists ............................................................................................. 25
   5.3 Creating a Secure Client ............................................................................................ 25
   5.4 Creating a Secure Server ........................................................................................... 27
   5.5 Using the DRC.X509Cert Class ............................................................................... 28
      5.5.1 Certificate Chains ............................................................................................. 30

6 HTTP Mode ....................................................................................................................... 32
   6.1 Receiving HTTP Messages ....................................................................................... 32
   6.2 HTTP Mode Events .................................................................................................. 33
      6.2.1 Event: HTTPHeader ......................................................................................... 34
      6.2.2 Event: HTTPBody ............................................................................................ 35
      6.2.3 Event: HTTPChunk ......................................................................................... 35
      6.2.4 Event: HTTPTrailer ......................................................................................... 35
   6.3 Sending HTTP Messages ........................................................................................... 36

revision 20191003_320
1 About This Document

This document is a complete guide to Conga, Dyalog's framework for TCP/IP communications. It describes the tools with whichConga 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.

ℹ️ Information note highlighting 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.

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. A full list of the platforms on which Dyalog version 17.1 is supported is available at [https://www.dyalog.com/dyalog/current-platforms.htm](https://www.dyalog.com/dyalog/current-platforms.htm). Within this document, differences in behaviour between operating systems are identified with the following icons (representing macOS, Linux, UNIX and Microsoft Windows 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), SAWS (the Stand-Alone Web Service framework), 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
     ...
     ```
   - 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.

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>
<thead>
<tr>
<th>Code</th>
<th>Object Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Root</td>
<td>The highest level object. The root object contains information about the Conga installation; it is created with the DRC.Init function.</td>
</tr>
</tbody>
</table>
Table 4-1: Conga object types (continued)

<table>
<thead>
<tr>
<th>Code</th>
<th>Object Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Server*</td>
<td>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.</td>
</tr>
<tr>
<td>2</td>
<td>Client*</td>
<td>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.</td>
</tr>
<tr>
<td>3</td>
<td>Connection</td>
<td>A server object can respond to multiple client connections; a connection object maintains information pertaining to each client connection.</td>
</tr>
<tr>
<td>4</td>
<td>Command</td>
<td>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).</td>
</tr>
<tr>
<td>5</td>
<td>Message</td>
<td>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).</td>
</tr>
</tbody>
</table>

* 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.7); 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.19). 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.
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.

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**

<table>
<thead>
<tr>
<th>Code</th>
<th>Object State</th>
<th>Conga Object Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>New</td>
<td>x</td>
<td>Transient state that the object exists in after it is created but before it has been initialised.</td>
</tr>
<tr>
<td>1</td>
<td>Incoming</td>
<td>x</td>
<td>A connection has been established but a Connect event has not yet been received.</td>
</tr>
<tr>
<td>2</td>
<td>RootInit</td>
<td>x</td>
<td>The root object exists and is connected to the Conga library (Microsoft Windows DLL or UNIX/Linux Shared Library).</td>
</tr>
<tr>
<td>3</td>
<td>Listen</td>
<td>x</td>
<td>The server is listening for incoming connection attempts.</td>
</tr>
<tr>
<td>4</td>
<td>Connected</td>
<td>x</td>
<td>The client/connection is connected to its connection/client peer.</td>
</tr>
<tr>
<td>5</td>
<td>APL</td>
<td>x</td>
<td>The thread that handles socket communications has a full buffer and no further processing can occur until the application calls the DRC.Wait function.</td>
</tr>
<tr>
<td>6</td>
<td>ReadyToSend</td>
<td>x</td>
<td>Data is ready to be sent.</td>
</tr>
<tr>
<td>7</td>
<td>Sending</td>
<td>x</td>
<td>Sending data.</td>
</tr>
<tr>
<td>8</td>
<td>Processing</td>
<td>x</td>
<td>The command has been passed to the server but no response has been issued.</td>
</tr>
<tr>
<td>9</td>
<td>ReadyToRecv</td>
<td>x</td>
<td>Waiting for data.</td>
</tr>
<tr>
<td>10</td>
<td>Receiving</td>
<td>x</td>
<td>Receiving data.</td>
</tr>
</tbody>
</table>
Table 4-2: Conga object states and the object types that can exist in those states (continued)

<table>
<thead>
<tr>
<th>Code</th>
<th>Object State</th>
<th>Conga Object Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Finished</td>
<td>x x x x x x</td>
<td>All data has been transmitted/received, connections closed and commands finished.</td>
</tr>
<tr>
<td>12</td>
<td>MarkedForDeletion</td>
<td>x x x x x x</td>
<td>The Conga object is ready for deletion.</td>
</tr>
<tr>
<td>13</td>
<td>Error</td>
<td>x x x x x x</td>
<td>An error has occurred.</td>
</tr>
<tr>
<td>14</td>
<td>internal</td>
<td>- - - - - -</td>
<td>internal</td>
</tr>
<tr>
<td>15</td>
<td>internal</td>
<td>- - - - - -</td>
<td>internal</td>
</tr>
<tr>
<td>16</td>
<td>SocketClosed</td>
<td>x x</td>
<td>The socket has been closed.</td>
</tr>
<tr>
<td>17</td>
<td>APPLast</td>
<td>x x</td>
<td>The connection has been closed but uncollected data still exists in the thread that handles socket communications.</td>
</tr>
<tr>
<td>18</td>
<td>SSL</td>
<td>x x</td>
<td>The client/connection is negotiating an SSL connection with its connection/client peer.</td>
</tr>
</tbody>
</table>

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 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.22).

  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).

- **BlkRaw**

  As *Raw* mode, but each data transmission that exceeds the maximum size (as defined by the `BufferSize` parameter of the `DRC.Clt/DRC.Srv` function) is chunked into blocks. 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 and cannot exceed the `BufferSize`.
  - magic number: 32-bit integer unique to the blocks in a single data transmission; used to identify blocks in the same transmission. Set using the `DRC.Clt/DRC.Srv` function (the *Magic* parameter).

  If data is too large to fit into a single block, then multiple blocks are created; each block is assigned the event name `Block`. If the connection closes before all blocks have been processed, then the final block in the transmission is assigned the event name `BlockLast`. The third element returned by the `DRC.Wait` function indicates the event name (see Section A.22).
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. Specifically, a Command mode client must be connected to a Command mode server and 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 as long as the requisite header (containing the field length and magic number) is added. An HTTP mode client must be connected to an HTTP mode server when both are running under Conga (for example, to connect the HttpCommand utility to MiServer) but a Conga server running in HTTP mode can communicate with a web browser client that does not use Conga. Text mode and Raw mode are typically used when only one end of the connection is an APL application.

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 that 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 Buffer Size bytes (see Section A.7 and Section A.19). 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 Buffer Size bytes then the data is returned in blocks of Buffer Size 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 function returns a list of all existing roots:

```
DRC←Conga.Init '' A Use the default root
DRC1←Conga.New '' A Create a new root with a generated name
Conga.RootNames
```

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'
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.22)
- [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
```

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.16)—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.15). The server can
also call the DRC.Progess function (see Section A.14); 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):

\[ \text{TODServer.RunCommand & 913} \]

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:

\[ \text{DRC.Clt 'C1' 'localhost' 913} \]
0 C1

\[ \text{DRC.Send 'C1' ''} \]
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].

\[ \text{DRC.Wait 'C1'} \]
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):

\[ \text{DRC.Send 'C1' ''} \]
0 C1.Auto00000001

\[ \text{DRC.Wait 'C1' 5000} \]
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
```
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://developer.mozilla.org/en/docs/Introduction_to_Public-Key_Cryptography](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](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 located and downloaded for use with Conga should be placed in a single root certificate directory. The DRC.SetProp function can be used to inform Conga of the location of this directory. For example, the sample root certificates supplied with Conga can be used by entering:

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

(The Samples.CertPath function returns the location of the TestCertificates folder.)

*Table 5-1* lists the download pages for root certificates for the most widely used CAs. The main root certificates for all these are supplied with Conga and can be found in [DYALOG]/PublicCACerts. 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*

<table>
<thead>
<tr>
<th>Authority</th>
<th>Download Root Certificates From...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comodo</td>
<td><a href="http://www.comodo.com/repository/">http://www.comodo.com/repository/</a></td>
</tr>
<tr>
<td>GoDaddy &amp; ValiCert</td>
<td><a href="https://certs.godaddy.com/Repository.go">https://certs.godaddy.com/Repository.go</a></td>
</tr>
<tr>
<td>Entrust</td>
<td><a href="http://www.entrust.net/developer/index.cfm">http://www.entrust.net/developer/index.cfm</a></td>
</tr>
</tbody>
</table>
Table 5-1: Root Certificates for the most widely-used CAs (continued)

<table>
<thead>
<tr>
<th>Authority</th>
<th>Download Root Certificates From...</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAcert</td>
<td><a href="http://www.cacert.org/index.php?id=3">http://www.cacert.org/index.php?id=3</a></td>
</tr>
<tr>
<td>GlobalSign</td>
<td><a href="https://www.globalsign.com/support/root-certificate/osroot.htm">https://www.globalsign.com/support/root-certificate/osroot.htm</a></td>
</tr>
<tr>
<td>IPS Servidores</td>
<td><a href="http://www.ips.es/Declaraciones/NuevasCAS/NuevasCAS.html">http://www.ips.es/Declaraciones/NuevasCAS/NuevasCAS.html</a></td>
</tr>
<tr>
<td></td>
<td>The root certificate is not included in [DYALOG]/PublicCACerts.</td>
</tr>
</tbody>
</table>

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 – these are used by the TestSecure* functions in the Samples namespace (see Section 1.0.1). 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/ca/DyalogCaPublic.pem**
  The public certificate for the test CA for https://ssltest.dyalog.com/, a Dyalog supplied test site used by the Samples.TestSecureWebClient function. It uses a different CA key to the one used by TestCertificates/ca/ca-cert.pem.

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

- **TestCertificates/server/server-cert.pem** and **server-key.pem**
  The certificate/key pair used for sample servers.
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 (both client and server must be running on Microsoft Windows).

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

A client must be defined as secure when it is created – it is not possible to convert an existing non-secure client into a secure client.

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**

```plaintext
  cert←⊃DRC.X509Cert.ReadCertFromFile 'path/client/client-cert.pem'
  cert.KeyOrigin←'DER' ('path/client/client-key.pem')
  certs←('X509' cert)('SSLValidation' 16)
  DRC.Clt 'C1' 'localhost' 713 'Text', certs 0 C1
```

In this example, the first line of code uses the `ReadCertFromFile` method in the `DRC.X509Cert` class to read the certificate called `client-cert.pem` and record all of its information in a new instance of the `X509Cert` class. The second line specifies the location of the `client-key.pem` file, which contains the private key. The third line creates a variable comprising the parameters required for the creation of the secure client and the fourth line includes this information when creating the secure client.

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

```plaintext
  certs←⊂('PublicCertFile' ('DER' ('path/client/client-cert.pem')))
    certs←⊂('PrivateKeyFile' ('DER' ('path/client/client-key.pem')))
    certs←⊂('SSLValidation' 16)
```
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 #. [X509Cert]
```

**EXAMPLE**

```
args←'C1' 'ssltest.dyalog.com' 713 '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; in this situation Conga uses Dyalog's certificate in the Microsoft Certificate Store's trusted root certificates directory.

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.

### 5.4 Creating a Secure Server

A server must be defined as secure when it is created – it is not possible to convert an existing non-secure server into a secure server.

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.X509Cert.ReadFile 'path/TestCertificates/server/server-cert.pem'
cert.KeyOrigin←'DER' 'path/TestCertificates/server/server-key.pem'
certs←('X509' cert)('SSLValidation' 64)
```
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.11). 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.19) 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.18) 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.

### 5.5 Using the DRC.X509Cert Class

Conga includes a class to encapsulate certificate handling – DRC.X509Cert. This class has methods to read certificates from files, folders and Microsoft certificate stores; for a complete description of the DRC.X509Cert class, see Section A.23.

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:

```lisp
path←Samples.CertPath
file←path,'client/client-cert.pem'
pmyCert←DRC.X509Cert.ReadCertFromFile file
1
```

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

```lisp
⊢myCert←⊃myCert
#.DRC.X509Cert.[X509Cert]
```

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:

```lisp
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
```

```lisp
myCert.Elements.(ValidFrom ValidTo)
2008 2 15 16 19 50 0 2018 2 12 16 20 4 0
```

```lisp
myCert.Formatted.(ValidFrom ValidTo)
Fri Feb 15 11:19:50 2008
Mon Feb 12 11:20:04 2018
```

```lisp
myCert.Extended.(ValidFrom ValidTo)
Fri Feb 15 16:19:50 2008
Mon Feb 12 16:20:04 2018
```

```lisp
myCert.⎕NL °3 A examine methods
AsArg Chain CopyCertificationChainFromStore IsCert
ReadCertFromFile ReadCertFromFolder ReadCertFromStore Save
```

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

![Certificate chain diagram]

Figure 5-1: Certificate chain

Continuing the example in Section 5.5:

```plaintext
myCert.Chain
#.DRC.X509Cert.[X509Cert]  #.DRC.X509Cert.[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:

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

CN=Test client  O=Test CA,CN=Test CA

]DISPLAY myCert.Chain.Formatted.Issuer

O=Test CA,CN=Test CA  O=Test CA,CN=Test CA
```

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:

```drc
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 of the 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 too 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 occurs, the format of the event's data element depends on the setting of the server/client/connection object's `DecodeBuffers` property. Each HTTP event has a related, additive, `DecodeBuffers` value. When `DecodeBuffers` is not set for an HTTP event, Conga returns a simple character vector that needs to be parsed outside Conga. When `DecodeBuffers` is set, Conga will do some parsing and return an array of elements that are meaningful to the particular event and more convenient for processing. The `DecodeBuffers` values for each event are:

- HTTPHeader = 1
- HTTPTrailer = 2
- HTTPChunk = 4
- HTTPBody = 8

These values are additive. Typically, `DecodeBuffers` will be set to 15 to decode the data for all HTTP-related events. For example:

```
DRC.SetProp obj 'DecodeBuffers' 15 \ a 1+2+4+8 = 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| 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
```

---

*Conga User Guide*

*revision 20191003_320*
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.
6.3 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.
6.4 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.

6.4.1 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, set the value of the WSFeatures property as required (0 to manually validate/confirm the response, 1 to automatically accept a positive response from the server). Dyalog Ltd recommends setting WSFeatures to 1 unless you have a good understanding of the WebSocket protocol.

The subsequent upgrade process depends on the value of the WSFeatures property; Table 6-1 summarises the steps this process follows.
### Table 6-1: Effect of setting WSFeatures on the Server/Client

<table>
<thead>
<tr>
<th>WSFeatures on Client</th>
<th>WSFeatures on Server</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1. Client sets the WSUpgrade property</td>
<td>1. Client sets the WSUpgrade property</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. Server receives a WSUpgradeReq event</td>
<td>2. Server receives (and automatically accepts) a WSUpgrade event</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. Server inspects headers and, if acceptable, sets the WSAccept property</td>
<td>3. Client receives a WSResponse event</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. Client receives a WSResponse event</td>
<td>4. Client inspects headers (and optional additional data) and, if acceptable, sets the WSAccept property</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5. Client inspects headers (and optional additional data) and, if acceptable, sets the WSAccept property</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1. Client sets the WSUpgrade property</td>
<td>1. Client sets the WSUpgrade property</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. Server receives a WSUpgradeReq event</td>
<td>2. Server receives (and automatically accepts) a WSUpgrade event</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. Server inspects headers and, if acceptable, sets the WSAccept property</td>
<td>3. Client receives a WSUpgrade event</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. Client receives a WSUpgrade event</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example**

This example creates a client object and upgrades it to use the WebSocket protocol. The WSFeatures property on this example client object is set to 0 (responses must be manually validated/confirmed).

Create an HTTP client:

```
DRC.Clt 'C1' 'localhost' 8080 'http' A create HTTP client
0  C1
```

Do not automatically accept a positive server response:

```
DRC.SetProp 'C1' 'WSFeatures' 0
0
```
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.

```ruby
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 6.4.2).

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.

---

**Revision:** 20191003_320

---

**Conga User Guide**
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

### 6.4.2 Server-side WebSocket Upgrade

When a client requests an upgrade to use the WebSocket protocol, the server will either receive a WSUpgrade event (if the WSFeatures property is set to 1) or a WSUpgradeReq event from a call to the DRC.Wait function (if the WSFeatures property is set to 0).

**Example**

```
0 ← 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+h0Fs1q5
```

SkEnx8bvp6w=

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 6.4.1), 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')
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.

6.4.3 Transmitting WebSocket Data

Once an HTTP connection has been upgraded to a WebSocket, the DRC.Send function can be used to transmit data. The data must be sent as 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 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**

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

6.4.4 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 6.4.3).

6.4.5 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.

```autoexec
Create an empty certificate
cert←⎕NEW DRC.X509Cert ¢
```
A Connect to the server
   DRC.Clt 'c1' 'echo.websocket.org' 80 'http' ('X509' cert)

A Enable WebSockets
   DRC.SetProp 'c1' 'WSFeatures' 1

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

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: iY2q6p0pJDCyWJeWcU8wJcFm=
   Server: Kaazing Gateway
   Upgrade: websocket

A Send a message...
   DRC.Send 'c1' ('hello' 1)

0 0 c1 Auto00000000

A ...which is simply echoed back
   DRC.Wait 'c1'

0 c1 WSRceive hello 1 1
A Close the connection
DRC.Close 'c1'
0
# The Conga Workspace

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

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRC</td>
<td>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.</td>
</tr>
<tr>
<td>Conga</td>
<td>Namespace containing the Conga interface functions and methods – these are detailed fully in Appendix A.</td>
</tr>
</tbody>
</table>
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.
- **HttpGetCommand** – 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.
- **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](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](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.
- **HttpGetServers** – 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](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**

```r
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**

```r
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.
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**

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

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 indicates 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.

**Example**

```plaintext
DRC.SetProp '.' 'CompLevel' 9  // use maximum compression
```
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:

\[
\text{DRC.SetProp} \ '.\ ' \text{'EventMode'} 1
\]

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

<table>
<thead>
<tr>
<th>EventMode</th>
<th>Socket closes</th>
<th>Socket times out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1119 'ERR_CLOSED' 'Socket closed while receiving'</td>
<td>100 'TIMEOUT' ''</td>
</tr>
<tr>
<td>1</td>
<td>0 'S1' 'Closed' 1119</td>
<td>0 'S1' 'Timeout' 100</td>
</tr>
</tbody>
</table>

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**

```plaintext
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.
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:

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

## A.2 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 ← Conga.Init {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 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 ''
```

<table>
<thead>
<tr>
<th>Conga.RootNames</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT</td>
</tr>
<tr>
<td>---------</td>
</tr>
</tbody>
</table>

```
(i1 i2 i3).RootName
```

<table>
<thead>
<tr>
<th>(i1 i2 i3).RootName</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT</td>
</tr>
</tbody>
</table>
Related functions:
- Conga.New – see Section A.3
- Conga.RootNames – see Section 1.1
- DRC.RootName – see Section 1.1

A.3 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: \texttt{ref ← Conga.New \{rootname\}}

where:
- \texttt{ref} is a reference to the instance of Conga.LIB.
- \texttt{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
\begin{verbatim}
  i1←Conga.New ''
i2←Conga.New ''
  (i1 i2).RootName
\end{verbatim}

\begin{verbatim}
  i3←Conga.New 'IC1'
Name in use: IC1
  i3←Conga.New 'IC1'
  ^
  i3←Conga.Init ''
  (i1 i2 i3).RootName
\end{verbatim}

Related functions:
- Conga.Init – see Section A.2
- Conga.RootNames – see Section 1.1
A.4 Function: DRC.Certs

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

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

**Syntax:**

```dfn
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:**

```dfn
]Display DRC.Cert 'ListMSStore'
```

```
0  My  Root  Trust  CA  UserDS  TrustedPublisher
```

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

A Display some information about the second certificate

```dfn
(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.5 Function: DRC.ClientAuth

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


Syntax: \texttt{rc \leftarrow DRC.ClientAuth \{clientname\} [{\{userid\} \{password\}}]}

where:
- \texttt{rc} is the return code (see Section A.1)
- \texttt{clientname} is the name of the Command-mode client.
- \texttt{userid} is the user's Microsoft Windows User ID.
- \texttt{password} is the user's Microsoft Windows password.

DRC.ClientAuth and DRC.ServerAuth (see Section A.17) must be run at the same time.

A.6 Function: DRC.Close

Purpose: Closes the specified Conga object.

Syntax: \texttt{rc \leftarrow DRC.Close \{servername|clientname|connectionname|commandname\}}

where:
- \texttt{rc} is the return code (see Section A.1)
- \texttt{servername|clientname|connectionname|commandname} is the name of the Conga server/client/connection/command to close (respectively).

Example
\begin{verbatim}
DRC.Close 'C1'
\end{verbatim}
0

A.7 Function: DRC.Clt

Purpose: Creates a Conga client object.
Syntax: \texttt{rc name + DRC.Clt \{clientname\} \{Address\} \{Port\} \{Mode\} [BufferSize] [SSLValidation] [EOM] [IgnoreCase] [Protocol] [PublicCertData] [PrivateKeyFile] [PrivateKeyPass] [PublicCertFile] [PublicCertPass] [PrivateKeyData] [Priority] [Magic] [X509]}

where:

- \texttt{rc} is the return code (see Section A.1)
- \texttt{name} is the name of the Conga client that has been created. If no clientname was specified (' ') then this is auto-generated.
- \texttt{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.
- \texttt{Address} is the address of the server
- \texttt{Port} is the port number that the server will listen on
- \texttt{Mode} is the connection protocol (see Section 4.1.3). Possible values are \texttt{Command}, \texttt{Text}, \texttt{Raw}, \texttt{BlkText} or \texttt{BlkRaw}. The default is \texttt{Command}.
- \texttt{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 \texttt{Raw/Text/BlkRaw/BlkText} mode, not those in \texttt{Command} mode.
- \texttt{SSLValidation} is the sum of the relevant TLS flags (see Appendix C). Only valid when creating a secure client (see Section 5.3).
- \texttt{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 \texttt{Raw/Text} mode, not those in \texttt{BlkRaw/BlkText/Command} mode.
- \texttt{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 \texttt{Raw/Text} mode, not those in \texttt{BlkRaw/BlkText/Command} mode.
- \texttt{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.

**EXAMPLES**

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

```lisp
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:

```lisp
mycert←⊃DRC.X509Cert.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>:

\[
\text{DRC.Clt ' ' 'localhost' 30 'Text' 1000 ('EOM' (\texttt{UCS 13 10}))}
\]

0  CLT00000000

Related function:
- DRC.Srv – see Section A.19

### A.8 Function: DRC.Describe

**Purpose:** Returns a description of the specified Conga object. Similar to the DRC.Tree function (see Section A.20) except that DRC.Describe only returns information for the specified object (not its children) and the descriptions are textual rather than numeric codes.

**Syntax:** \( \text{rc desc ← DRC.Describe \{objectname\}} \)

where:
- \( \text{rc} \) is the return code (see Section A.1)
- \( \text{desc} \) is a multi-element vector proving a textual description of the specified object
- \( \text{objectname} \) is the name of the Conga object to describe

For all objects except the root object, the description \( \text{desc} \) 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 \( \text{desc} \) 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 \( \text{desc} \) 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)
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.13
- DRC.Tree – see Section A.20

A.9 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.10 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.11 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 Table A-1)
  • 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 Table A-1)
  • arg is an additional argument that might be required depending on the property requested (see Table A-1)

The property values that can be retrieved depend on the type of the specified Conga object; they are described in Table A-1. These values are specified using the DRC.SetProp function (see Section A.18) or when a server/client is created using the DRC.Srv/DRC.Clt function (see Section A.19 and Section A.7 respectively).

Table A-1: Properties that can be retrieved by the DRC.GetProp function

<table>
<thead>
<tr>
<th>Property</th>
<th>Object Type</th>
<th>Description/Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeepAlive</td>
<td>Client, Server,</td>
<td>The frequency with which periodic heartbeat messages are sent to verify that the</td>
</tr>
<tr>
<td></td>
<td>Connection</td>
<td>connection is live. A 2-element vector in which:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• [1] is the time (in ms) to wait before sending the first heartbeat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• [2] is the time interval (in ms) between heartbeats</td>
</tr>
<tr>
<td>Property</td>
<td>Object Type</td>
<td>Description/Syntax</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LocalAddr</td>
<td>Client, Server, Connection</td>
<td>A 4-element vector in which:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- [1] is the communication protocol</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- [2] is the IP address (formatted according to the communication protocol) and port number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- [3] is the address bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- [4] is the port number being used</td>
</tr>
<tr>
<td>Magic</td>
<td>Connection</td>
<td>A 32-bit number unique to the blocks in a single data transmission. Only valid in BlkRaw/BlkText mode, not Raw/Text/Command mode.</td>
</tr>
<tr>
<td>OwnCert</td>
<td>Client, Server, Connection</td>
<td>The X509Cert object containing information about the certificate of the specified Conga object.</td>
</tr>
<tr>
<td>PeerAddr</td>
<td>Client, Connection</td>
<td>The address of the specified Conga object's peer. A 4-element vector in which:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- [1] is the communication protocol</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- [2] is the IP address (formatted according to the communication protocol) and port number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- [3] is address bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- [4] is the port number being used</td>
</tr>
<tr>
<td>PeerCert</td>
<td>Client, Connection</td>
<td>The X509Cert object containing information about the certificate of the specified Conga object's peer.</td>
</tr>
<tr>
<td>PropList</td>
<td>all</td>
<td>A list of the properties relevant for the object's type.</td>
</tr>
<tr>
<td>Protocol</td>
<td>Root</td>
<td>The communication protocol in use. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- IPv4 : use the IPv4 connection protocol; if this is not possible then generate an error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- IPv6 : use the IPv6 connection protocol; if this is not possible then generate an error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- IP : use the IPv6 connection protocol; if this is not possible then use the IPv4 connection protocol</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This property value is inherited when creating a connection unless a different value is specified.</td>
</tr>
</tbody>
</table>
**Table A-1:** Properties that can be retrieved by the DRC.GetProp function (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Object Type</th>
<th>Description/Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadyStrategy</td>
<td>Root, Server</td>
<td>The strategy by which the next connection to process is determined when more than one connection has received data. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: &quot;Use First&quot; – process the first connection in the object tree (for information on the object tree, see Section A.20). This approach can result in connections not being serviced.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: &quot;Round Robin&quot; – process the first connection in the object tree after the one that has just been processed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: &quot;Oldest First&quot; – 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.</td>
</tr>
<tr>
<td>RootCertDir</td>
<td>Root</td>
<td>Full path to (and name of) the directory that contains Certificate Authority root certificates.</td>
</tr>
<tr>
<td>TCPLookup</td>
<td>Root</td>
<td>Requires an additional argument comprising a 2-element vector in which:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[1] is a URL or IP address as a character string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[2] is the port number (0 means all ports at the URL/IP address) or service name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The address of the specified URL/IP address and port. A 4-element vector in which:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[1] is the communication protocol</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[2] is the IP address (formatted according to the communication protocol) and port number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[3] is the address bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[4] is the port number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Multiple 4-element vectors can be returned if both IPv4 and IPv6 information is available.</td>
</tr>
</tbody>
</table>

**EXAMPLES**

```
DRC.GetProp "." "PropList"
0 Certificates DecodeCert PropList Protocol ReadyStrategy RootCertDir Stores TCPLookup
```
A.12  Function: DRC.Init

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

Syntax: \texttt{rc ← \{reset\} DRC.Init \{'\'}\}

where:
- \texttt{rc} is the return code (see Section A.1)
- \texttt{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

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

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

\textbf{EXAMPLE}

\begin{verbatim}
DRC.Init ''
0 Conga loaded from: ...\conga27x64Uni
\end{verbatim}

A.13  Function: DRC.Names

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

Syntax: \texttt{names ← DRC.Names \{objectname\}}

where:
- \texttt{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 \texttt{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.8
• DRC.Tree – see Section A.20

A.14  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.15).

Syntax: \texttt{rc ← DRC.Progress \{commandname\} \{data\}}

where:
• \texttt{rc} is the return code (see Section A.1)
• \texttt{commandname} is the name of the command received from the client. It must match the \texttt{objectname} returned by the DRC.Wait function (see Section A.22)
• \texttt{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:

\texttt{DRC.Progress (2⊃waitresult) 'Task 50\% completed'}

Related functions:
• DRC.Respond – see Section A.15
• DRC.Send – see Section A.16
• DRC.Wait – see Section A.22
A.15  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.22*)
- `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.14*
- `DRC.Send` – see *Section A.16*
- `DRC.Wait` – see *Section A.22*

A.16  Function: DRC.Send

**Purpose:** Send data to the peer client/server. Not valid for a *Command*-mode server (see *Section A.15* 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.

- 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 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)

• 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:

```apl```
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.15).
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.14
- `DRC.Respond` – see Section A.15
- `DRC.Wait` – see Section A.22

### A.17 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.5) and **DRC.ServerAuth** must be run at the same time.
A.18 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 Table 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.19 and Section A.7 respectively). The current values can be retrieved using the DRC.GetProp function (see Section A.11).

Table A-2: Properties that can be set by the DRC.SetProp function

<table>
<thead>
<tr>
<th>Property</th>
<th>Object Type</th>
<th>Description/Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompLevel</td>
<td>Root, Client, Server, Connection</td>
<td><code>Command and HTTP modes only.</code> The compression level to apply when sending data in <code>Command</code> mode or a file in <code>HTTP</code> mode. Valid values are 0–9, where 0 indicates no compression and 9 indicates maximum compression. The default is 6.</td>
</tr>
</tbody>
</table>
| ConnectionOnly| 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. |
Table A-2: Properties that can be set by the DRC.SetProp function (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Object Type</th>
<th>Description/Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>DecodeBuffers</td>
<td>Client, Server,</td>
<td>HTTP mode only.</td>
</tr>
<tr>
<td></td>
<td>Connection</td>
<td>The HTTP events that should return data in a decoded</td>
</tr>
<tr>
<td></td>
<td></td>
<td>format. Possible values for individual events are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• HTTPHeader = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• HTTPTrailer = 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• HTTPChunk = 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• HTTPBody = 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>These values are additive. Typically, DecodeBuffers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>will be set to 15 to decode the data for all HTTP-related</td>
</tr>
<tr>
<td></td>
<td></td>
<td>events. If DecodeBuffers is not set or is set to 0, a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>simple character vector is returned.</td>
</tr>
<tr>
<td>EventMode</td>
<td>Root</td>
<td>Whether a socket closing/timing out results in an</td>
</tr>
<tr>
<td></td>
<td></td>
<td>event or an error message being generated. Possible</td>
</tr>
<tr>
<td></td>
<td></td>
<td>values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0: Closing a socket generates error number 1119; a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>socket timing out generates error number 100.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1: Closing a socket generates a Closed event; a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>socket timing out generates a Timeout event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The default is 0.</td>
</tr>
<tr>
<td>KeepAlive</td>
<td>Client, Server,</td>
<td>The frequency with which periodic heartbeat messages</td>
</tr>
<tr>
<td></td>
<td>Connection</td>
<td>are sent to verify that the connection is live. A 2-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>element vector in which:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• [1] is the time (in ms) to wait before sending the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>first heartbeat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• [2] is the time interval (in ms) between heartbeats</td>
</tr>
</tbody>
</table>
### Table A-2: Properties that can be set by the DRC.SetProp function (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Object Type</th>
<th>Description/Syntax</th>
</tr>
</thead>
</table>
| **Pause** | 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. |
| **Protocol** | 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-2: Properties that can be set by the DRC.SetProp function (continued)**

<table>
<thead>
<tr>
<th>Property</th>
<th>Object Type</th>
<th>Description/Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadyStrategy</td>
<td>Root, Server</td>
<td>The strategy by which the next connection to process is determined when more than one connection has received data. Possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0: &quot;Use First&quot; – process the first connection in the object tree (for information on the object tree, see Section A.20). This approach can result in connections not being serviced.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1: &quot;Round Robin&quot; – process the first connection in the object tree after the one that has just been processed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 2: &quot;Oldest First&quot; – 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The default is 2.</td>
</tr>
<tr>
<td>RootCertDir</td>
<td>Root</td>
<td>Full path to (and name of) the directory that contains Certificate Authority root certificates.</td>
</tr>
<tr>
<td>WSAccept</td>
<td>Connection</td>
<td>HTTP mode only. The positive response to send to a WSUpgradeReq request from a client. A 2-element vector in which:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• [1] is the data element (element [4]) of the WSUpgradeReq event received from the client</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• [2] is a character vector of any additional headers that the server might want to send</td>
</tr>
</tbody>
</table>
### Table A-2: Properties that can be set by the DRC.SetProp function (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Object Type</th>
<th>Description/Syntax</th>
</tr>
</thead>
</table>
| WSFeatures  | Client, Server | *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 WSError 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. |
| WSUpgrade   | Client       | *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. |

**Examples**

```plaintext
DRC.SetProp '"RootCertDir' 'C:\..\TestCertificates\ca'
0
```
A.19  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]

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).
- **PublicKeyFile** has been deprecated but is retained for backwards compatibility. Only valid when creating a secure server (see Section 5.4).
- **PublicKeyData** 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).
The [PublicCertData], [PrivateKeyFile], [PrivateKeyPass], [PublicCertFile], [PublicCertPass] and [PrivateKeyData] arguments are mutually exclusive with the [X509] argument.

**EXAMPLES**

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

```plaintext
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):

```plaintext
cert←⊃DRC.X509Cert.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>`:

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

Related function:
- `DRC.Clt` – see *Section A.7*

## A.20 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
```

---

**Revision 20191003_320**
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.8
- DRC.Names – see Section A.13

A.21 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
2 6 936
```

A.22 Function: DRC.Wait

**Purpose:** Waits for an event to occur.
Syntax: \( \text{rc objectname event data} \rightarrow \text{DRC.Wait\{clientname|servername|connectionname|commandname\}} \ [\text{timeout}] \)

where:
- \text{rc} is the return code (a return code of 100 indicates a timeout, for other return codes see Section A.1)
- \text{objectname} is the name of the object on which the event occurred.
- \text{event} is the name of the event that occurred—see Table A-3
- \text{data} is the received data.
- \text{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.
- \text{timeout} is the number of ms to wait before timing out. The default is 1,000.

\textbf{Table A-3: Types of event that can occur}

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>\textit{Text or Raw mode only}: A block of data was received and the connection is still open.</td>
</tr>
<tr>
<td>BlockLast</td>
<td>\textit{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.</td>
</tr>
<tr>
<td>Closed</td>
<td>The socket has been closed. Only valid when \text{EventMode} is set to 1 (see Section A.18).</td>
</tr>
<tr>
<td>Connect</td>
<td>The Conga object has been created but has not yet participated in any data transmissions.</td>
</tr>
<tr>
<td>Error</td>
<td>An error occurred.</td>
</tr>
</tbody>
</table>
**Table A-3: Types of event that can occur (continued)**

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTPBody</td>
<td><em>HTTP mode only:</em> 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.</td>
</tr>
<tr>
<td>HTTPChunk</td>
<td><em>HTTP mode only:</em> A chunk was received. Only valid when receiving an HTTP message with the Transfer-Encoding header set to 'chunked'.</td>
</tr>
<tr>
<td>HTTPHeader</td>
<td><em>HTTP mode only:</em> an HTTP header (the start of an HTTP message) was received.</td>
</tr>
<tr>
<td>HTTPTrailer</td>
<td><em>HTTP mode only:</em> 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'.</td>
</tr>
<tr>
<td>Progress</td>
<td><em>Command</em>-mode client only: The server transmitted data using the DRC.Progress function.</td>
</tr>
<tr>
<td>Receive</td>
<td><em>Command</em> mode only: Data has been received.</td>
</tr>
<tr>
<td>Sent</td>
<td>The transmission of data has been completed. Only valid when the DRC.Send function's close parameter is set to 3 (see Section A.16).</td>
</tr>
<tr>
<td>Timeout</td>
<td>The DRC.Wait function has timed out. Only valid when EventMode is set to 1 (see Section A.18).</td>
</tr>
<tr>
<td>WSReceive</td>
<td><em>HTTP mode only:</em> Data has been received using the WebSocket protocol.</td>
</tr>
</tbody>
</table>
**Table A-3: Types of event that can occur (continued)**

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSResponse</td>
<td>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.18).</td>
</tr>
<tr>
<td>WSUpgrade</td>
<td>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.18).</td>
</tr>
<tr>
<td>WSUpgradeReq</td>
<td>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.18).</td>
</tr>
</tbody>
</table>

**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.14
- DRC.Respond – see Section A.15
- DRC.Send – see Section A.16

A.23 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.

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 the specified certificate store:

certs ← DRC.X509Cert.ReadCertFromStore {storename}

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.
- storename is a single certificate store name as a character vector (a list of all certificate store names is returned by DRC.Certs 'ListMSStore' – see Section A.4).

Syntax – Instance Methods: These act on specific certificate instances.
To verify whether the certificate is structurally valid:

```csharp
bool ← #.DRC.X509Cert.[X509Cert].IsCert
```

To return the certificate chain for the certificate:

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

To save the certificate instance to a file:

```csharp
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:

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

where:

- `bool` is a Boolean indicating whether the certificate has a valid structure:
  - 0: the certificate does not have a valid structure
  - 1: the certificate has a valid structure
- `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**

Verify whether `john` (an instance of the X509Cert class in the Samples namespace) is a structurally valid certificate:

```csharp
Samples.john.IsCert
```

1
Return the issuer information for *john* (an instance of the X509Cert class in the *Samples* namespace):

\[(\text{Samples.john.Chain}).\text{Formatted.Issuer}\]

\[O=\text{Test CA}, \text{CN=Test CA}\]

### A.23.1 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*

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cert</strong></td>
<td>Integer vector of raw certificate data.</td>
</tr>
</tbody>
</table>
| **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-cert.pem  
| **Elements** | 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). |
| **Extended** |                                                                                                                                          |
| **Formatted** |                                                                                                                                          |
| **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-key.pem  
| **LDRC**     | *Internal reference to the local DRC namespace.*                                                                                         |
Table A-4: Properties of each instance of the DRC.X509Cert class (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParentCert</td>
<td>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.</td>
</tr>
</tbody>
</table>
| 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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlgorithmID</td>
<td>The cryptographic algorithm used to generate the signature, for example, RSA-SHA1 and DSA-SIGN.</td>
</tr>
<tr>
<td>Description</td>
<td>A text description of the certificate.</td>
</tr>
<tr>
<td>Issuer</td>
<td>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.</td>
</tr>
<tr>
<td>Key</td>
<td>The certificate's key in Boolean format.</td>
</tr>
<tr>
<td>KeyHex</td>
<td>The certificate's key in hexadecimal format.</td>
</tr>
<tr>
<td>KeyID</td>
<td>The certificate's key's cryptosystem, for example, RSA or DHE.</td>
</tr>
<tr>
<td>KeyLength</td>
<td>The length of the certificate's key (in bits). Maximum value is 16,384.</td>
</tr>
</tbody>
</table>
### Table A-5: Some of the elements that can comprise the Elements, Extended, and Formatted properties of each instance of the DRC.X509Cert class (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SerialNo</td>
<td>A number that uniquely identifies the certificate and is issued by the certification authority.</td>
</tr>
<tr>
<td>Subject</td>
<td>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.</td>
</tr>
<tr>
<td>ValidFrom</td>
<td>Together, these two elements define the period of validity for the certificate.</td>
</tr>
<tr>
<td>ValidTo</td>
<td></td>
</tr>
<tr>
<td>Version</td>
<td>The version of the X.509 standard applied when creating the certificate (currently this is 3).</td>
</tr>
</tbody>
</table>
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

   - 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/Certificates/CertTool.dyalog

3. Edit the values of the following names in the CertTool namespace:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXEC</td>
<td>Fully-qualified path to the certool.exe file in the bin directory of the unzipped GnuTLS secure communications library. The certool.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\certool.exe'</td>
</tr>
<tr>
<td>TARGET</td>
<td>Fully-qualified path to the directory in which to store the generated certificates and files. For example: 'c:\temp\TestCertificates'</td>
</tr>
</tbody>
</table>
### 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 \( \text{SERIAL} + 7 \times 1 + 15 \) (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: Dyalog Ltd

### 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

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CertAcceptIfIssuerUnknown</td>
<td>Accept the peer certificate even if the issuer (root certificate) cannot be found.</td>
</tr>
<tr>
<td>2</td>
<td>CertAcceptIfSignerNotCA</td>
<td>Accept the peer certificate even if it has been signed by a certificate not in the trusted root certificates' directory.</td>
</tr>
<tr>
<td>4</td>
<td>CertAcceptIfNotActivated</td>
<td>Accept the peer certificate even if it is not yet valid (according to its valid from information).</td>
</tr>
<tr>
<td>8</td>
<td>CertAcceptIfExpired</td>
<td>Accept the peer certificate even if it has expired (according to its valid to information).</td>
</tr>
<tr>
<td>16</td>
<td>CertAcceptIfIncorrectHostName</td>
<td>Accept the peer certificate even if its hostname does not match the one it was trying to connect to.</td>
</tr>
<tr>
<td>32</td>
<td>CertAcceptWithoutValidating</td>
<td>Accept the peer certificate without checking it (useful if the certificate is to be checked manually – see Section A.11).</td>
</tr>
</tbody>
</table>
**Table C-1: TLS Flags (continued)**

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>RequestClientCertificate</td>
<td>Only valid for a server; asks the client for a certificate but allows connections even if the client does not provide one.</td>
</tr>
<tr>
<td>128</td>
<td>RequireClientCertificate</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 – the following five tables show the necessary libraries for each of the supported combinations of operating system, edition and width.

**Table D-1: AIX operating system**

<table>
<thead>
<tr>
<th></th>
<th>Unicode Edition</th>
<th>Classic Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit</td>
<td>conga32x64Uni.so</td>
<td>conga32x64.so</td>
</tr>
<tr>
<td></td>
<td>libconga32ssl64.a</td>
<td>libconga32ssl64.a</td>
</tr>
<tr>
<td>32-bit</td>
<td>conga32Uni.so</td>
<td>conga32.so</td>
</tr>
<tr>
<td></td>
<td>libconga32ssl32.a</td>
<td>libconga32ssl32.a</td>
</tr>
</tbody>
</table>

**Table D-2: Linux operating system**

<table>
<thead>
<tr>
<th></th>
<th>Unicode Edition</th>
<th>Classic Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit</td>
<td>conga32x64Uni.so</td>
<td>conga32x64.so</td>
</tr>
<tr>
<td></td>
<td>libconga32ssl64.so</td>
<td>libconga32ssl64.so</td>
</tr>
</tbody>
</table>

**Table D-3: macOS operating system**

<table>
<thead>
<tr>
<th></th>
<th>Unicode Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit</td>
<td>conga32x64Uni.dylib</td>
</tr>
<tr>
<td></td>
<td>libconga32ssl64.dylib</td>
</tr>
<tr>
<td>32-bit</td>
<td>conga32Uni.dylib</td>
</tr>
<tr>
<td></td>
<td>libconga32ssl32.dylib</td>
</tr>
</tbody>
</table>
### Table D-4: Microsoft Windows operating system

<table>
<thead>
<tr>
<th></th>
<th>Unicode Edition</th>
<th>Classic Edition</th>
</tr>
</thead>
</table>
| **64-bit**| Conga32x64Uni.dll
          | Conga32ssl64.dll           | Conga32x64.dll
          |                             | Conga32ssl64.dll          |
| **32-bit**| Conga32Uni.dll
          | Conga32ssl32.dll           | Conga32.dll
          |                             | Conga32ssl32.dll          |

### Table D-5: Raspbian operating system

<table>
<thead>
<tr>
<th></th>
<th>Unicode Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>32-bit</strong></td>
<td>conga32Uni.so</td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Code</th>
<th>Source</th>
<th>Reason for Error</th>
<th>Possible Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>UNIX</td>
<td>Attempted to allocate a port with number less than 1025 without having root permission.</td>
<td>Either allocate a port number above 1024 or sign on as root (see 4001 in the <em>Dyalog APL Language Reference Guide</em>).</td>
</tr>
<tr>
<td>98</td>
<td>UNIX</td>
<td>Specified port number is already in use.</td>
<td>Allocate a different port number (it can take several minutes to de-allocate a port before it can be reused).</td>
</tr>
<tr>
<td>100</td>
<td>Conga</td>
<td>Timeout – nothing was received within the specified timeout period.</td>
<td>This is a normal occurrence and should be accommodated for in the client/server code.</td>
</tr>
<tr>
<td>1105</td>
<td>Conga</td>
<td>Could not receive data.</td>
<td>Re-establish the connection.</td>
</tr>
</tbody>
</table>
### Table E-1: Possible error codes returned by Conga (continued)

<table>
<thead>
<tr>
<th>Code</th>
<th>Source</th>
<th>Reason for Error</th>
<th>Possible Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1119</td>
<td>Conga</td>
<td>Socket closed while receiving data (occurs when the connection is broken mid-block transfer).</td>
<td>Reconnect and resend the data.</td>
</tr>
<tr>
<td>1135</td>
<td>Conga</td>
<td>Maximum block size (as defined by the BufferSize parameter of the DRC.Clt/DRC.Srv function) exceeded when attempting to send/receive data.</td>
<td>Increase the value of the BufferSize parameter or chunk the data into smaller blocks.</td>
</tr>
<tr>
<td>1201</td>
<td>TLS</td>
<td>The handshake process that sets up a secure connection between the client and server before the certificates are exchanged is failing.</td>
<td>Ensure that the client and server are using the same encryption protocol and that both are using SSL/TLS.</td>
</tr>
<tr>
<td>1202</td>
<td>TLS</td>
<td>The certificate supplied by the peer is not valid.</td>
<td>Supply the TLS flag CertAcceptWithoutValidating to the DRC.Srv/DRC.Clt function (see Appendix C) to allow this connection and examine the certificate manually.</td>
</tr>
<tr>
<td>1203</td>
<td>TLS</td>
<td>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).</td>
<td>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.</td>
</tr>
<tr>
<td>1204</td>
<td>TLS</td>
<td>There was an error setting up the TLS libraries.</td>
<td>Ensure that all GnuTLS files are present and valid.</td>
</tr>
</tbody>
</table>
Change History

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

F.1 Version 3.2

Released with Dyalog version 17.1.

This version:

- Adds the ability to set the compression level when directly transmitting a file.
- Removes support for the FIFO mode server-only property.
- 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.

F.2 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.3 Version 3.0

Released with Dyalog version 16.0.
This version:

- adds:
  - two new server-only properties — FIFO Mode and Connection Only
  - integrated support for HTTP, including four new events (HTTP Body, HTTP Chunk, HTTP Header and HTTP Trailer), 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)
  - two new parameters to the DRC Srv function — Allow End Points and Deny End Points
  - 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, WSR esponse, 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 Comp Level)

- removes or relocates seldom-used or atrophied samples

**F.4 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).
- allows an empty left argument to be supplied to the Samples.HTTPGet function.

### F.5 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.

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.6 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.7 Version 2.4

An internal update incorporating features in support of the Remote Interactive Development Environment (RIDE).

F.8 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 connection is still live.

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.9 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](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.10  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 (-I) 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 .................................. 12
BlkText mode .................................. 12
Block events .................................. 12
BlockLast events .............................. 12

C
CAAs See Certificate authorities
Certificate authorities ................. 23
Certificate chains ......................... 30
Certificate revocation lists .......... 25
Certificate stores ......................... 25
Classes
  DRC.X509Cert .............................. 85
  Notation when calling ................. 53
Client-side WebSocket Upgrade .... 37
Closed events ................................ 51
Command mode ............................ 13
Compatibility
  Between object modes ................. 13
  Compression levels .................... 49
  With Dyalog .................. 4
Compression level ......................... 49
Conga object
  Modes .................................. 11
  Names .................................. 7
Properties ............................... 72
States .................................... 9
Types ..................................... 7
Conga object modes .................. 11
BlkRaw mode ............................. 12
BlkText mode ............................. 12
Command mode .......................... 13
Compatibility .......................... 13
HTTP mode ............................. 13, 32
Raw mode .................................. 12
Text mode .................................. 11
Conga object names .................. 7
Conga object properties ........... 62
Conga object states ................... 9
Conga object types ..................... 7
  Client .................................. 8
  Client-Server relationship .......... 8
  Command .............................. 8
  Connection ........................... 8
  Message ................................ 8
  Root .................................... 7
  Server .................................. 8
Conga objects ......................... 7
conga workspace ..................... 44
Conga.Init (function) ............. 54
Conga.New (function) .............. 55
Connect events ......................... 81

D
DRC.Certs (function) ................... 56
DRC.ClientAuth (function) ....... 57
DRC.Close (function) ............... 57
DRC.Clt (function) ................. 57
DRC.Describe (function) ........ 60
DRC.Error (function) ............... 61
DRC.Exists (function) ............. 61
DRC.GetProp (function) ........... 62