DYALOG

The tool of thought for expert programming

Dyalog™ for Windows

Conga User Guide

Version 12.0.3

Dyalog Limited

South Barn
Minchens Court
Minchens Lane
Bramley
Hampshire
RG26 5BH
United Kingdom

tel: +44 (0)1256 830030
fax: +44 (0)1256 830031
email: support@dyalog.com
http://www.dyalog.com

Version 12.0.3

First Edition May 2008

Contents

Introduction

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

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

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

From version 12.0 of Dyalog, Conga replaces the use of `TCPSocket` objects as the recommended mechanism for managing TCP-based communications from Dyalog. (Although Conga currently only uses the TCP protocol, the interface to Conga is at a level where other communication mechanisms could be added in the future.) The programming model for Conga is significantly simpler, and supports multi-threaded applications more easily than do `TCPSocket` objects. Conga also supports secure communication using TLS (Transport Layer Security), also known as SSL (Secure Socket Layer)[1].

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

[1] Starting with the versions of Conga included with Dyalog Version 12.0.3.

1. Conga fundamentals

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

1.1 A Simple Conga Client

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

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

```
      )COPY Conga DRC
…Conga saved… some time on some day…
      DRC.Init ''
0  Conga loaded from: …\bin\Conga10Uni
```

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

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

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

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

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

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

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

The result code of zero indicates that the client was successfully created. To receive incoming data, call `DRC.Wait` with the name of the object on which to wait, and (optionally) a timeout in milliseconds:

```
      DISP DRC.Wait 'C1' 1000
```

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

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

Finally, we should close the client object:

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

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

1.2. A Simple Server

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

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

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

```
      TODServer.Run&13
TOD Server started on port 13
```

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

```
      DRC.Client 'C1' 'localhost' 13 'Text' 1000
0
      DRC.Wait 'C1'
 0  C1  Block  14:36:23 06-05-2007
      DRC.Close 'C1'
0
```

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

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

The function `Run` works as follows:

`[3]` Call `DRC.Init` and set global flag `DONE` to zero.

`[4]` Create a Server object named `TOD` on selected port in Text mode with a 1,000 character buffer size.

`[8]` Repeat the following until `DONE` is set to `1`:

`[9]` Wait for any event and split the result into `rc` (return code), `obj` (object name), `event` and `data`. `obj` will be a string identifying a "child object" of `TOD`, with a name like `'TOD.127-0-0-1=55280'` (encoding the IP address and port from which the connection was made).

`[14]` If return code was `0` and the event was `Connect`, format the time of day

`[15]` Send the time of day to `obj`. The `1` in the 3rd element of the argument to `Send` instructs `Conga` to close the object as soon as the data has been sent.

`[17]` For any other event, we simply close the connection. (We are a very simple service.)

`[19]` If you want the service to periodically do housekeeping tasks, we will arrive here every 1000 milliseconds (specified in the argument to `Wait` on line 9).

`[21]` Any return code from `Wait` other than `0` or `100` will cause a shutdown of the service.

`[24]` When we are done, close the server object

1.3 Command Mode

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

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

Conga clients and servers support three different connection types:

`Text` Allows transmission of character strings, which *must* consist of characters with Unicode code points less than 256. To transmit characters outside this range, it is recommended that you UTF-8 encode the data (see `⎕UCS`).

`Raw` Essentially the same as a Text connection, except that data is represented as integers in the range `0` to `255` (for coding simplicity, negative integers `¯128` to `¯1` are also accepted and mapped to `128-255`).

`Command` Each transmission consists of a complete APL object in a binary format.

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

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

has arrived. For connections between APL clients and servers, Command mode is therefore more convenient.

When using Text and Raw connections, `Wait` will report incoming data each time a TCP packet arrives, or the receive buffer is full. The recipient may need to buffer incoming data in the workspace and analyze it to determine whether a complete message has arrived.

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

To do this, we need to make the following changes to `TODServer.Run`: Replace the `'Text'` parameter with `'Command'` on line `Run[4]`. Lines `[13-15]` can be replaced by the following:

```
[13]    :Case 'Connect'  ⍝ Ignore
[14]    :Case 'Receive'
[15]        ##.DRC.Respond obj ⎕TS
```

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

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

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

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

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

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

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

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

```
      DRC.Wait 'C1' 1000
0  C1.Auto00000000  Receive  2008 2 13 10 41 39 585
```

Element 4 of the result is now a 7-element integer vector rather than a formatted timestamp, which is more useful to an APL client. However, the server is of now unusable by other TCP client programs, if they are expecting a Text mode TOD

server. For this reason, it would be unwise to run the command mode service as a listener on port 13.

Note that the Command mode server also does not close the connection after sending a timestamp, so we can ask for the time of day again if we like:

```
      DRC.Send 'C1' ''
0  C1.Auto00000001
      DRC.Wait 'C1' 1000
0  C1.Auto00000001  Receive  2008 2 14 21 20 8 169
```

1.4 Parallel or Asynchronous Commands

It is not necessary to wait for the response to one command before the next is sent. You can also retrieve the results of each command in any order that you like. In the above examples, the command name was automatically generated, but you can also specify command names if you prefer:

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

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

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

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

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

1.5 More on Multi-Threading

Conga is specifically designed to support easy use of multi-threading. In particular, the ability to have a program work as both client and server simultaneously, without blocking other threads, has been a key design goal. All calls to Conga are

asynchronous calls to an external Windows DLL or Unix/Linux Shared Library. Waiting threads are suspended, but all other threads can continue execution.

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

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

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

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

The TOD service could be external, but it could even be running in the same workspace in a separate thread as described in section 1.2. Outside the workspace, Conga uses multiple operating system threads to handle TCP communications. It will handle communications independently of what the interpreter is doing, and return each result to the APL thread which is waiting for it, as appropriate.

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

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

1.6 Conga versus TCPSocket objects

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

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

Conga is designed to make it easy for APL developers to embed client or server components in APL applications. Conga hides many of the details of TCP socket handling, notifies the application of incoming data, connection events and errors – but the application does not need to do anything other than handle the data which arrives. Conga makes it straightforward to make remote calls in a multi-threaded client environment.

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

2. Secure Sockets

If you do not intend to use secure communications, you can safely skip to the next chapter which discusses the samples which are included with Conga.

From version 12.0.3 of Dyalog APL, Conga supports secure connections using SSL/TLS protocols. Secure connections allow client and server applications to:

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

SSL/TLS is a generic term for a set of related protocols used to add confidentiality and authentication to communications channels such as sockets. TLS, which stands for "Transport Layer Security" is the successor to SSL, the "Secure Socket Layer" protocol V3 designed by Netscape (http://wp.netscape.com/eng/ssl3/draft302.txt). TLS is defined by the IETF and described in RFC 2246. There are only minor differences between the two protocols, so their names are often used interchangeably.

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

http://developer.mozilla.org/en/docs/Introduction_to_Public-Key_Cryptography

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

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

2.1 CA Certificates

CAs (Certificate Authorities) are trusted third parties that sign certificates to indicate that a certificate belongs to who it claims to belong to. Assuming that you trust the CA that signed a certificate that some third party presents to you, and the CA certificate is still valid, you can use the certificate to verify the identity your communications partner, or "Peer[2]". To check the CAs signature on a certificate, you need to have access to the CAs public certificate (often called a root certificate).

[2] "Peer" comes from the description of TCP/IP as a "peer-to-peer" communications protocol.

Conga can be used to secure many different types of system, which may require different (and sometimes private) root certificates. Therefore, you may need to obtain the root certificates from several CAs. All public root certificates that you wish to use with Conga need to be placed in a root certificate directory. Conga need to be informed about the location of the root certificates with a call to the function `SetProp`. For example, you should be able to use the sample root certificates by typing:

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

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

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

The following table lists the download pages for root certificates for the most widely used CAs, and whether their main root certificates are shipped with Conga (in which case you can find them in the folder *PublicCACerts* below the main Dyalog program folder).

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

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

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

2.2 Client and Server Certificates

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

details of the public key algorithm and signing certificates signature to validate the contents of the certificate

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

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

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

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

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

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

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

2.3 Creating a Secure Client

Secure Conga Clients are created using the function `SecureClient`, which is very similar to the `Client` function, except that it takes additional argument elements containing the names of the relevant certificate and key files. A certificate is not required in order to create a secure client; many secure servers accept connections from clients without certificates. In this case, the server cannot verify the identity of the client, but the connection is still encrypted and safe from tampering. Most web commerce sites use this type of connection to protect sensitive used data transmitted over the internet without requiring that customers have a digital signature.

`SecureClient` has three additional arguments following the first three (name, address and port), which are shared with `Client`:

```
DRC.SecureClient name address port certfiles keyfile flags
```

The first additional argument, `certfiles`, contains a comma separated list of filenames. The first filename should be the public certificate used by the client, followed by one or more certificates in the “signing chain”.

`keyfile` contains the name of the private key for the client certificate. If no certificate exists, both `certfiles` and `keyfile` should be empty.

The third additional argument, `flags` contains the sum of TLS flags (see Appendix C for a complete list). A typical flag value used for a client connection would be 16 (accept the server certificate even if its hostname does not match the one we asked to

connect to), or 32 (accept without validating). The latter can be useful to determine the reason why a connection is failing. For example, if we try to connect toa secure site, and have not set `RootCertDir` to point to the required CA certificates, all attempts to make secure connections will fail:

```
    args←'' 'ssltest.dyalog.com'  443 '' ''
    args,←0 'Text' 100000
    DRC.SetProp '.' 'RootCertDir' 'c:\wrong'
    DRC.SecureClient args
1202  ERR_INVALID_PEER_CERTIFICATE
      /* The peers certificate is not valid */
```

Without access to the necessary CA certificate, validation fails:

```
    Args[6]←32 ⍝ Connect without validation

    DRC.SecureClient args
0  CLT00000051
```

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

```
    DRC.GetProp '.' 'CLT00000051' 'PeerCert'
0   Issuer              C=UK,ST=Hampshire,L=Bramley,O=Dyalog,
                        CN=Dyalog Ltd. Test Root CA,
                        EMAIL=jonathan@dyalog.com
    Subject             C=UK,O=JM \+ Dyalog Ltd.,OU=York unit,
                        ST=York,CN=Jonathan @ Dyalog
    CertificateVersion  3
    PublicKeyAlgorithm  RSA
    SerialNumber        01
    ValidFrom           2008 01 02 15 09 47 0
    ValidTo             2009 01 01 15 09 47 0
       …etc…
```

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

```
    DRC.SetProp '.' 'RootCertDir' 'C:\..\TestCertificates\ca'
0
    args[6]←0 ⍝ Require validation
    DRC.SecureClient args
0  CLT00000052
```

Once a Secure Client has been created, the rest of the communication works in exactly the same was as for a non-secure client. The only difference between the two is the use of `SecureClient` to make the connection, and the possibility of extracting the server's certificate information once the connection has been made.

2.3 Creating a Secure Server

Secure Servers are created using the function `SecureServer`, which is requires exactly the same additional information as does `SecureClient`:

```
DRC.SecureServer name port certfiles keyfile flags
```

Unlike a secure client, a secure server ***must*** have a certificate; the `certfiles` and `keyfile` arguments may not be empty when creating a secure server. See the previous section on creating secure clients for an explanation of these arguments. Once a secure server is created, it is managed in the same way as a non-secure server.

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

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

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

3. Samples

The distributed workspace `Conga` contains a number of working examples, which are intended to demonstrate how to use most of the capabilities of Conga. Although they are simple, many of the samples have enough functionality to be used as the starting point for communicating Dyalog applications. We hope to be able to provide more elaborate examples of Conga-based communications during 2008, on the APL Wiki or our own web site. We invite all users to submit examples!

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

3.1 Overview

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

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

3.2 The Samples Namespace

The `Samples` namespace contains a number of functions with names beginning with `Test`. Between them, these examples should show examples of most of the different ways that the above components can be used. In alphabetical order, the test functions supplied with Dyalog Version 12.0.3 are:

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

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

3.3 Web Client

The functions `Samples.HTTPGet` and `HTTPSGet` show how Conga can be used to retrieve the contents of a web page from an internet site (the latter function using a secure connection). For example:

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

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

```
      1⊃z
  0
```

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

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

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

```
      ⍴3⊃z
20715
      60↑3⊃z
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//E
```

Note that if the header has a content-type element containing the string `charset=utf-8`, the content part should be translated from UTF-8. In version 12, this can be done as follows:

```
      'UTF-8' ⎕UCS ⎕UCS 3⊃z
```

The rightmost `⎕UCS` translates the content into Unicode "code points, which will be numbers between `0` and `255`. Dyadic `⎕UCS` with a left argument of `'UTF-8'` interprets decodes this byte stream and returns (Unicode) characters.

The `HTTPGet` sample uses Conga as follows:

[4] Call `DRC.Init` to ensure `DRC` is initialised

`[7]` `HTTPUtils.HostPort` is used to detect a trailing port number (eg `:8080`) if one was supplied, otherwise the default HTTP port of `80` is used

`[9]` A Text-mode client is created, with a buffer size of 100k

`[10]` An HTTP "GET" command is sent to the Web Server

`[14]` We wait for blocks of data. As soon as some data has arrived, we call `HTTPUtils.DecodeHeader` to detect and extract the HTTP header

`[22]` We look for a Content-Length field in the header, as this will allow us to know how much data to expect

`[30-34]` Various logic is used to determine whether all data has arrived. If the declared Content-Length has been delivered, if the server delivered data and closed the socket (`BlockLast` event), or if the text contains an ending `</html>` tag (the latter is perhaps dubious)

`[44]` We close the client object

3.4 Web Server

The namespace `WebServer` contains an ultra-simple implementation of a web server. This does not provide any of the services that sophisticated web servers provide. Nonetheless, it shows just how little code is required to implement a web server that interfaces to a web browser. The `WebServer` example supports serving real files from the file system, or using APL functions to intercept requests and manufacture virtual pages on request.

During 2008, Dyalog hopes to port Stefano Lanzavecchia's WildServer example to Conga, and put this framework in the public domain. The WildServer provides a comprehensive framework for web application development, with support for sessions, cookies, authentication, forms validation, headers, and so on.

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

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

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

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

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

`[4]` Call `DRC.Init` to ensure `DRC` is initialised.

`[13]` Create a Raw-mode server. (Same as Text-mode except it returns byte numbers in the range `0-255`.)

`[24]` Loop on `Wait`, timing out every 10 seconds.

`[27]` Switch on the first element of the result of `Wait`.

`[28]` Code `0`: `Wait` returned an application event.

`[31]` The event was an error on the socket, which will have been closed. Clean up the data namespace for the client. (`SpaceName` generates a namespace name based on the IP address and port number of the client.)

`[38]` Data arrived. Find the client namespace and call `HandleRequest` in a new thread, passing the object name and input data to it. `HandleRequest` will eventually call `DRC.Send` to send the answer to the client.

`[46]` If the client closed the connection, expunge the namespace.

`[50]` Insert code here to react to `Connect` events.

`[56]` If the return code was `100`, nothing happened for 10 seconds. Insert timekeeping code here. (For a busy web server, we will need to do housekeeping even if we don't have any timeouts.)

`[60]` If `Wait` returned `1010` (Object not found), this means that the server object has disappeared. This probably means that another thread has closed it. Many of the test functions do this once they have completed client tests.

`[72]` If we get to the end of the while loop, this means that we shut down because some component of the server set stop to `1`. We close the server object ourselves.

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

3.5 RPC Client and Server

The RPC Server is similar in structure to the web server discussed in the previous section, except that Command sockets are used to transmit Remote Procedure Calls to the server, which validates and executes them, and returns the array result to the client. Both client and server need to be Conga users. A Command-mode client application for use by other languages is possible; similar tools have been built in the past. However, most non-APL clients already support SOAP/XML for remote procedure calls in the form of Web Services. A Web Service platform based on Conga will be available from Dyalog in the spring of 2008.

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

`[4]` Call `DRC.Init` to ensure `DRC` is initialised.

`[6-12]` In order to be able to return an error if it is unable to start the server, `Run` first creates the Command-mode server on line `[7]`, and only starts a new handling thread on line `[9]` by calling itself recursively with a left argument of `0` if the server could be created. The handler continues execution from the `:While` on line `[15]`.

`[16]` Loop on `Wait`, timing out every 5 seconds.

`[18]` Switch on the first element of the result of `Wait`.

`[19]` Code `0`: `Wait` returned an application event:

`[21]` The event was an error. If the object in error was the server itself, we close it and stop running. Otherwise, the error is ignored. (We might do some housekeeping if we were tracking client sessions.)

`[27]` Data arrived. Validate the format of the incoming array, and confirm the first element names a function that may be called. If all is OK, run `Process` in a new thread, passing the object name and input data to it. `Process` will eventually call `DRC.Progress` and subsequently `DRC.Respond` to first signal progress and finally send the answer to the client.

[38] We ignore connection events.

`[44, 46]` Codes `100` and `1010`: See `WebService.Run` in the previous section.

`[72]` If we get to the end of the while loop, this means that we shut down because some component of the server set stop to `1`. We close the server object ourselves.

3.6 FTP Client

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

The use of Conga by the class is as follows:

`Open[4]`	`Call DRC.Init` to ensure `DRC` is initialised.
`Open[6]`	Create a Text-mode client for issuing commands to the FTP server.
`Open[11-13]`	Use the function `Do` to enter user ID and password and check for the expected responses from the server.
`Do[5]`	Send a command to the server. `Do` returns the FTP state code following the command.
`ReadReply[106]`	Wait for a response, called by `Do`.
`GetData[3]`	Execute the PASV command to prepare for passive-mode data transfer; server returns a dataport that it has opened.

`GetData[4]`	Connect a Text- or Raw-mode client to the dataport identified by PASV.
`GetData[5-6]`	Set ASCII or Binary mode and issue the command which will return data.
`GetData[9-11]`	Wait and collect output response until the server closes connection.
`GetData[14]`	Confirm the server thinks transfer was completed.
`PutData[6-8]`	Same as `GetData[4-6]`
`PutData[10]`	Send all data in a single call to `DRC.Send`; Conga will break the data up into TCP Packets.

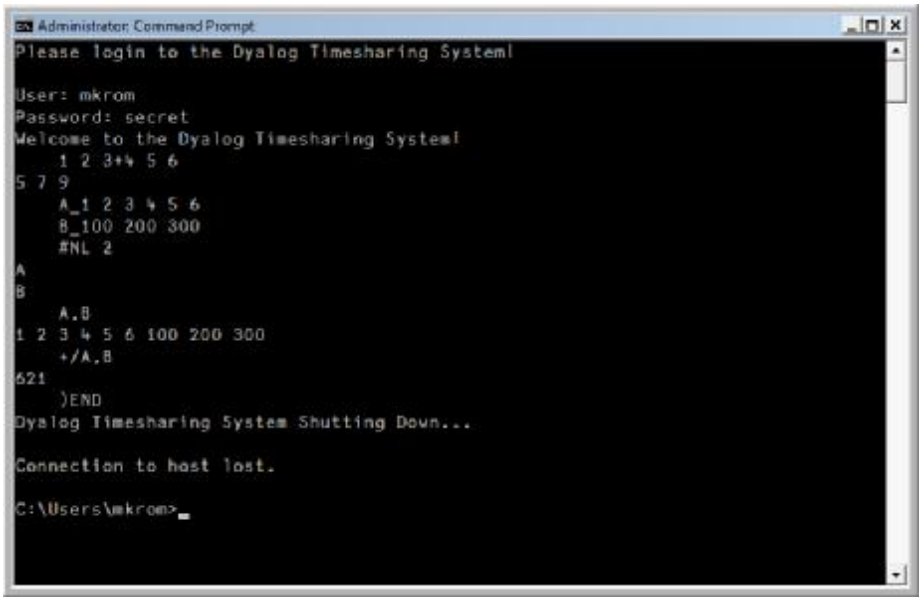
The FTP client protocol is surprisingly easy to implement.

3.7 Telnet Server

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

```
      TelnetServer.Run&0
Dyalog Timesharing System 'TELNETSRV' started on port 23
```

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



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

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

3.8 Telnet Client

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

3.9 TODServer

This example was discussed in detail in Chapter 1.

Appendix A: Function reference

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

DRC.Client

Purpose Creates a Conga Client

Syntax `rc name←DRC.Client name address port [mode size [stop]]`

`Rc`	return code	`0`	Non-zero on error
`name`	client name	`'C1'`	If empty, will be generated (see name in result).
`address`	IP address or name	`'192.168.1.1'` `'localhost'`	
`port`	service port	`80`	
`mode`	type of client	`'Command'|` `'Raw'|'Text'`	See `DRC.Server`
`size`	buffer size	`1000`	Maximum data size
`stop`	stop options	`('/' 1 0)`	Termination String, Cut to Right of string, Ignore Case

Watch Out The Stop Option settings described in this document have been withdrawn pending review, are being reviewed, and are likely to be different in future versions of Conga.

Examples

```
DRC.Client 'C1' '192.168.1.1' 5050
```

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

```
DRC.Client '' 'localhost' 13 'Text' 1000 ((⎕UCS 13 10) 1 0)
```

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

Typical Errors

`1009`	Object name already in use

1110, 1111	Nothing seems to be listening on the port

DRC.Close

Purpose Closes any Conga object

Syntax `rc ← DRC.Close name`

rc	return code	0	Non-zero on error
name	object name	'C1'	Should be the name of an existing Conga object

Example

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

Typical Errors

1010	Object name does not exist

DRC.Error

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

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

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

Example

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

DRC.Exists

Purpose Tests the existence of an object.

Syntax `bool ← DRC.Exists name`

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

Example

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

DRC.GetProp

Purpose Retrieving properties from a Conga object.

Syntax `res ← DRC.GetProp obj property`

`res` Depends on the object and the property requested – see table below.

`obj` Name of the Conga object

`property` Name of the property

Examples:

```
      DRC.GetProp '.' 'PropList'
 PropList RootCertDir

      DRC.GetProp '.' 'RootCertDir'
 '/usr/rootcerts'
```

Supported Properties:

Name	Objects	Description
`PropList`	All	Returns the list of property names
`CertRootDir`	Root ('.')	Name of the folder containing Certificate Authority root certificates (should be set using SetProp)
`OwnCert`	Server, Connection	Retrieves information about your own certificate.
`PeerCert`	Client, Connection	Retrieves information about the certificate used at the other end of the connection.

Certificate Information:

Certificate information is returned as a 2 column matrix of string property/value pairs, and the exact number may vary from certificate to certificate. This matrix will normally include the certificate issuer, subject, public key algorithm, certificate format version, serial number, valid from & to dates. If no certificate exists, or in the case of a Server object no certificate information has been requested (see the section on TLS Flags in Appendix C), the result will have zero rows.

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

DRC.Init

Purpose Loads and initializes the Conga DLL or Shared Library

Syntax `rc ← DRC.Init ''`

`rc`	return code	`0`	Non-zero on error

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

Example

```
      DRC.Init ''
  0
```

Typical Errors

`1000` Unable to load the library

DRC.Names

Purpose Returns names of existing objects

Syntax `names ← DRC.Names root`

`names`	object names	Names of children of root
`root`	root object	`''|'C1 '`

Example

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

See also `DRC.Tree`

DRC.Progress

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

Syntax `rc ← DRC.Progress name data`

`rc`	return code	`0`	Non-zero on error
`name`	command name	`'C1.CMD1'`	
`data`	any array		

Example

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

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

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

DRC.Respond

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

Syntax `rc ← DRC.Respond name data`

`rc`	return code	`0`	Non-zero on error
`name`	command name	`'C1.CMD1'`	
`data`	any array		

Example

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

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

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

DRC.SecureClient

Purpose Creates a Secure Conga Client

Syntax

```
rc name←DRC.Client name address port certfiles keyfile
                                    flags [mode size [stop]]
```

`rc`	return code	`0`	Non-zero on error
`name`	client name	`'C1'`	If empty, will be generated (see name in result).
`address`	IP address or name	`'192.168.1.1'` `'localhost'`	
`port`	service port	`80`	
`certfiles`	certificate files (optional)	`'client.pem,root.pem'`	Comma separated list of filenames for the public certificate used by the client & the certificates in its signing chain. The first filename will be the certificate for the client.
`keyfile`	key files (optional)	`'Key.pem'`	The path & filename for the private key for the first certificate specified in `certfiles`. This is the private key for the client certificate.
`flags`	option flags	`0`	Sum of all the flags required for the connection. See the TLS Flags section in Appendix C.

`mode`	type of client	`'Command'|` `'Raw'|'Text'`	See `DRC.Server`
`size`	buffer size	`1000`	Maximum data size
`stop`	stop options	`('/' 1 0)`	Termination String, Cut to Right of string, Ignore Case

Watch Out The Stop Option settings described in this document have been withdrawn pending review, are being reviewed, and are likely to be different in future versions of Conga.

Examples

```
cert← 'client-cert.pem,ca-cert.pem'
key←'client-key.pem'

DRC.Client 'C1' '192.168.1.1' 5050 cert key '' 0
```

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

```
DRC.Client '' 'localhost' 13 '' '' 0 'Text'
                                    1000 ((⎕UCS 13 10) 1 0)
```

(Text-mode client with no certificate, an auto-generated name, connected to server on port 13 on the same machine, with a maximum buffer size of 1000 characters, and termination sequence of CRLF)

Typical Errors

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

Plus all the TLS handshaking errors listed in Appendix C.

DRC.SecureServer

Purpose Create a Secure Conga Server to listen on a selected port.

Syntax `rc name ← DRC.SecureServer name port certfiles keyfile flags [mode size [stop]]`

`rc`	return code	`0`	Non-zero on error
`name`	client name	`'S1'`	If empty, will be generated (see name in result).
`certfiles`	certificate files	`'cert.pem,root.pem'`	Comma separated list of filenames for the public certificate used by the server & the certificates in its signing chain. The first filename will be the certificate for the server
`keykile`	private key file	`'Key.pem'`	The path & filename for the private key for the first certificate specified by `certiles`. This is the private key for the server certificate.
`flags`	TLS flags	`0`	Sum of all the flags required for the server. See the TLS Flags section in Appendix C.
`port`	service port	`80`	Port on which to listen for connections
`mode`	type of client	`'Command'| 'Raw'|'Text'`	See discussion below
`size`	buffer size	`1000`	Maximum data size
`stop`	stop options see below	`('/' 1 0)`	Termination String, Cut to Right of string, Ignore Case.

See `DRC.Server` for a discussion of `mode` and other options.

Example:

```
      cert←'server-cert.pem,ca-cert.pem'
      key←'server-key.pem'
      DRC.SecureServer 'APLRPC' 5050 'Command' cert key 64
0  APLRPC
```

Creates a secure Command server using the names certificate and key files, and the flag value 64 (RequestClientCertificate).

Typical Errors

In addition to the errors listed for `DRC.Server`, the TLS-specific errors listed in Appendix C.

DRC.Send

Purpose Send data to partner.

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

`rc`	return code	`0`	Non-zero on error
`command`	command name	`'C1.CMD1'`	Generated from Client name if necessary (in command mode)
`name`	client / command	`'C1'` `'C1.CMD1'`	Client or Command name. In command mode, if a client name is supplied, command name is generated automatically.
`data`	any array		
`close`	close flag	`1`	If 1, connection will be closed

Command-mode Client example

```
      DRC.Send 'C1' ('PlusReduce' (⍳10))
0  C1.Auto00000001
      DRC.Wait 'C1.Auto00000001'
0  C1.Auto00000001  Receive  55
```

Creates a command below the named client, and sends an APL array to the server. The command name is generated if a complete name (for example `'C1.CMD1'`) is not provided.

Command-mode Server example: Use `DRC.Respond`

Raw- or Text-mode example

```
      DRC.Send 'C1' ('Bye',CR) 1
0
```

The above example sends the text `'Bye'` on the Client `C1` and subsequently closes the connection. When replying to a recently received message (in a server), you would typically write use the result from wait and write something along the lines of:

```
      DRC.Send ((2⊃waitresult)) ('Bye',CR) 1
0
```

DRC.Server

Purpose Create a Conga Server to listen on a selected port.

Syntax `rc name ← DRC.Client name port [mode size [stop]]`

`rc`	return code	`0`	Non-zero on error
`name`	client name	`'C1'`	If empty, will be generated (see name in result).
`port`	service port	`80`	
`mode`	type of client	`'Command'| 'Raw'|'Text'`	See discussion below
`size`	buffer size	`1000`	Maximum data size
`stop`	stop options see below	`('/' 1 0)`	Termination String, Cut to Right of string, Ignore Case.

Watch Out The Stop Option settings described in this document have been withdrawn pending review, are being reviewed, and are likely to be different in future versions of Conga.

Mode / Stop

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

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

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

Examples

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

Creates a Command mode server listening on port 5050.

```
      DRC.Server '' 23 'Text' 1000 ((⎕UCS 13) 1 0)
0  SRV00000001
```

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

Typical Errors

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

DRC.SetProp

Purpose Updating properties of Conga objects.

Syntax `DRC.SetProp '.' property value`

`property`	Name of the property to set
`value`	New value for the property

Example

```
DRC.SetProp '.' 'RootCertDir' '/usr/rootcerts'
```

See GetProp for a complete list of properties supported by different Conga objects. `RootCertDir` is currently the only settable property, and is used to specify the location of the Certificate Authority Root Certificates.

DRC.Tree

Purpose Return state information about an object and all of its "children".

Syntax `rc tree ← DRC.Tree root`

`Rc`	return code	`0`	Non-zero on error
`Tree`	tree (see below)		Description of the object and its child nodes
`Root`	root object	`'.' | 'C1 '`	Root object

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

For all objects, the first three elements of the object description are: [1] Name, [2] Type, [3] State. The root object also has additional information: [4] Version, [5] Semaphore count

Object Type codes (2nd element)

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

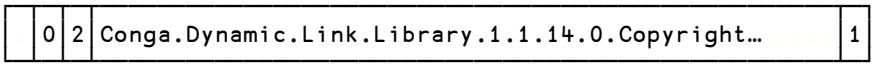
State codes (3rd element)

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

Example

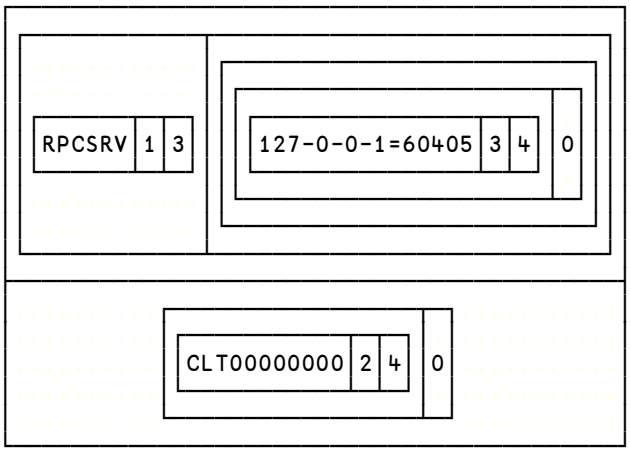
```
      (rc (root subtree))←##.DRC.Tree '.'
      rc
0

      DISP root
```



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

```
      DISP 2 1⍴subtree
```



Interpretation There are two children of the root, named `RPCSRV` and `CLT00000000` (this example was 'shot' while running `Samples.TestRPCserver`). `RPCSRV` is of type `1` (Server) and in state `3` (Listen). It has a child named `127-0-0-1=60405` which is a Connection (`3`) in state Connected (`4`). `CLT00000000` is a client (`2`) in state Connected (`4`).

See also `DRC.Names`

DRC.Wait

Purpose Wait for an event to occur.

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

`rc`	return code	`0` `100`	`0` means data received `100` means timeout
`obj`	object name	`'C1.CMD1'`	Object on which an event occurred
`event`	event name	`'block'`	See below
`data`	received data		
`name`	root name	`'C1'`	If a server or connection name is used, Wait will report events on the named object or any any of its children. If a command mode client wishes to wait on a specific command, the full command name can be given.
`timeout`	how long to wait	`1000`	Optional: Number of milliseconds to wait before timing out. If not specified, defaults to 1000 milliseconds.

Events

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

`Block`	A block of data was received (Text or Raw mode) and the connection is still open.
`BlockLast`	A block of data was received and the connection was closed; no more data can be expected. If the connection is closed while it is inactive, a `BlockLast` event will be reported with empty data.
`Connect`	`obj` is a newly created connection.
`Error`	An error occurred (data will contain an error message)
`Progress`	Command mode client only: The server transmitted data using the `DRC.Progress` function
`Receive`	Command mode only: data received

Command-mode server examples:

```
      DRC.Wait 'RPCSRV' 1000
0  RPCSRV.127-0-0-1=58070  Connect  0
```

A client connected to the server, the connection name is RPCSRV.127-0-0-1=58070 (generated from the IP address and port used by the client).

```
      DRC.Wait 'RPCSRV' 1000
0  RPCSRV.127-0-0-1=58070.Auto00000000   Foo 1
      DRC.Respond 'RPCSRV.127-0-0-1=58070.Auto00000000' (Foo 1)
```

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

Appendix B: Creating and Converting Certificates

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

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

PEM files begin and end with:

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

And contain a base 64 encoded version of the certificates.

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

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

http://www.gnu.org/software/gnutls/manual/html_node/Invoking-certtool.html

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

Create a self signed test CA certificate & key:

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

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

Create a private key & certificate request

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

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

Generate & Sign a certificate request

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

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

Appendix C: TLS Flags and Errors

TLS Flags

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

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

TLS Related error codes

Purpose This table lists the TLS specific error messages that can be returned when dealing with TLS connections. In addition any of the normal Conga or TCP errors may be returned.

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