Gcom®
Data Tunneling
User Guide

July, 2008

Gcom, Inc.

1800 Woodfield Drive
Savoy, IL 61874

Voice: 217.351.4241
Fax: 217.351.4240

Email: support@gcom.com
http://www.gcom.com

Dave Grothe was the subject matter expert for this manual.

Table of Contents

List of Tables

List of Figures

Tunneling vs. Gcom Data Tunneling

To most people, *tunneling over TCP* means transporting a particular protocol from one side of an IP network to the other using a TCP connection as the transport mechanism. It is the protocol that is tunneled. What emerges from the far end of the tunnel is a set of protocol objects that look just like the ones that entered the near end of the tunnel. That means the protocol that exits the tunnel must be the same protocol that enters the tunnel.

Gcom's *data tunneling* is quite different: The a Gcom Protocol Appliance 2G (GPA 2G) tunnels only the payload data carried by the legacy protocol. It does so by:

- Terminating the legacy protocol (SNA, X.25, Bisync, etc.)
- Applying a delimiter (header or header/trailer combination) to preserve logical message boundaries needed to send payload data over a byte stream connection such as TCP

The means you can use Gcom's Data Tunneling protocol to convert between legacy protocols on one side of the connection and payload data sent over TCP/IP on the other side.

For example: You can easily feed legacy protocol payload data into...

- Existing TCP/IP devices, such as ATMs, that use simple message-boundary-preserving encapsulations.
- Existing TCP/IP-based applications, such as EDI, that use similarly simple encapsulations.

An added bonus: By connecting a pair of encapsulated TCP connections back to back, you can even achieve protocol translation between any two legacy protocols. For example: If one legacy device talks SNA but another can only supply data via X.25, you can use Gcom's Data Tunneling protocol as a bridge between the two.

Gcom Data Tunneling Overview

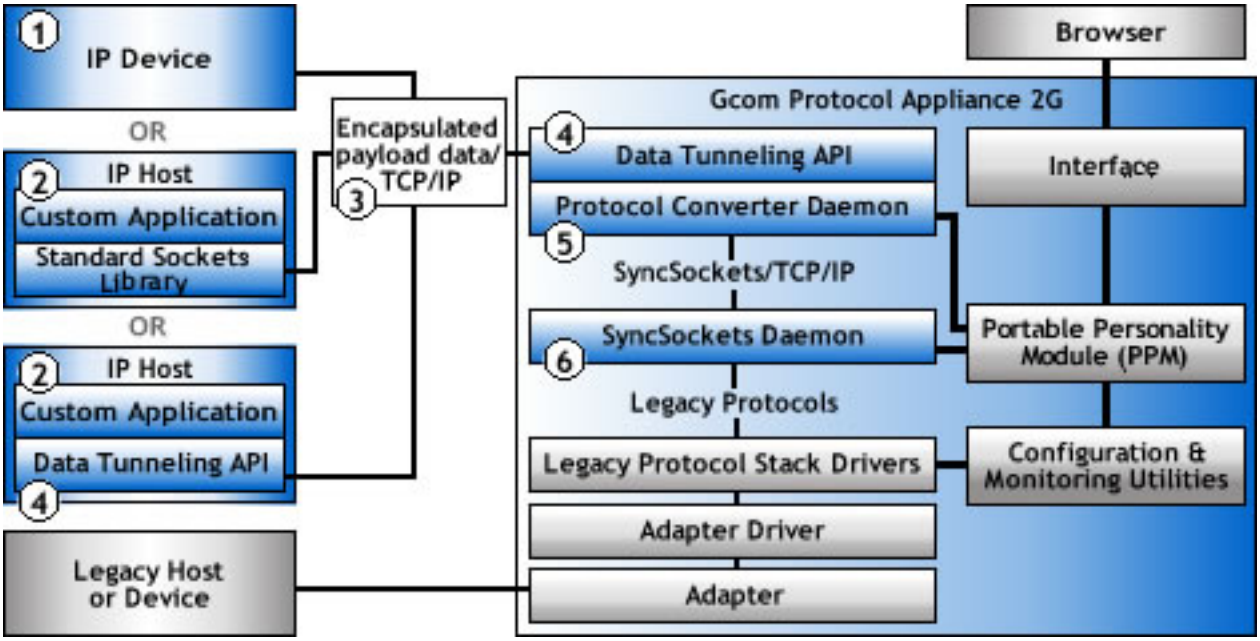


Figure 1 – Gcom Data Tunneling Overview

1. **IP Device** = Device, such as an ATM, TN3270 client, custom equipment, etc., that processes encapsulated payload data and communicates using TCP/IP
2. **Custom Application =** User-developed application that processes encapsulated payload data and communicates using TCP/IP
3. **Encapsulated Payload Data** = Payload data with delimiters added to preserve logical message boundaries needed to send payload data over a byte stream connection such as TCP
4. **Data Tunneling (DT) API** = Gcom® proprietary library of functions (provided in both C and Java):
   - Used by a developer to create a custom application
   - Used by a Gcom Protocol Appliance 2G (GPA 2G) to interface with an IP device or a custom application on an IP host
   - Built on a standard sockets library (an industry-standard repository of C functions to create and use IP communications) to give it a TCP/IP-based look and feel
   - Offers a wide variety of payload data delimiter formats called *encapsulations*
5. **Protocol Converter Daemon** = Gcom-proprietary, stand-alone application-level program that converts legacy protocol data streams from Gcom's SyncSockets® protocol to encapsulated payload data over TCP/IP

6. **SyncSockets Daemon** = Gcom-proprietary, stand-alone application-level program that converts legacy protocols to/from SyncSockets protocol over TCP/IP
7. **Gcom Protocol Appliance 2G (GPA 2G)** = Secure, closed, high-MTBF, small-footprint unit that can convert between legacy protocols on one side of a connection and encapsulated payload sent over TCP/IP on the other side

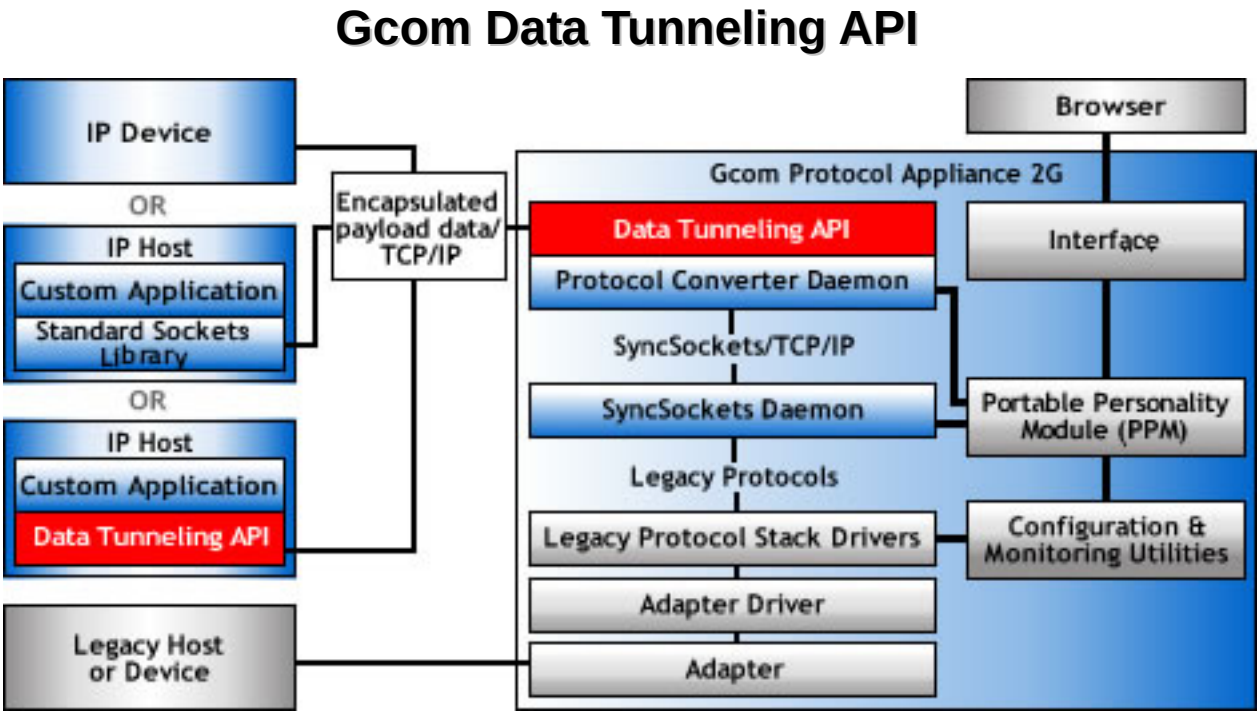


Figure 2 – Gcom Data Tunneling API

The Data Tunneling (DT) API is a Gcom-proprietary library of functions (provided in both C and Java):

- Used by a developer to create a custom application
- Used by a GPA 2G to interface with an IP device or a custom application on an IP host
- Built on a standard sockets library (an industry-standard repository of C functions to create and use IP communications) to give it a TCP/IP-based look and feel
- Offers a wide variety of payload data delimiter formats called *encapsulations*

Gcom's Data Tunneling API advantages:

- Using the Gcom Data Tunneling API to manage TCP connections in either client mode (outgoing connections) or server mode (incoming connections) is easier than using the standard sockets library.
- The Gcom Data Tunneling API has over a dozen built-in data encapsulation techniques that take care of all encapsulation details.
- You can use simple send and receive functions to operate on the payload data.

API Style

Gcom's Data Tunneling API uses:

- A simple subroutine calling style of interaction with the custom application
- A callback routine to the custom application to signal connection establishment

Thread Compatibility

Gcom's Data Tunneling API routines are all thread-safe and thread-efficient. You can handle connections using threads, poll lists, or any combination of the two that make sense in the context of solving the communication problem at hand.

Non-blocking Operation

By default, Gcom's Data Tunneling API assumes:

- The sockets used for data tunneling are operated in non-blocking mode. This means they must be used in conjunction with `poll` to assure they are readable or writeable when those operations are attempted.
- The custom application uses either threads or poll lists to discover when a socket has data to be read. When the custom application calls the `dt_recv` function, the function returns one encapsulated message with the encapsulation header/trailer removed. If the entire message is not received, the function returns `–1` with `errno` set to `EAGAIN` to read the socket again in order to receive the remainder of the message.
- The custom application uses either threads or poll lists to take care of flow control back pressure when writing to a socket. When the custom application calls the `dt_send` function, the function sends the encapsulated data unless TCP flow control does not permit transmission. A call to the `dt_send` function following a `poll`, which indicates the socket is write ready, should always succeed.

Data Tunneling API Functions

API Function	Description
dt_close	Close a socket.
dt_init	Initialize the API and set logging options.
dt_lock_print	Get the API's log printing mutex.
dt_open	Open a socket and: • Connect to a remote peer. • Or listen for incoming connections from remote peers.
dt_printf	Print into the log file.
dt_recv	Receive an encapsulated message from an open socket.
dt_reuseaddr	Set SO_REUSEADDR option on a socket. (Useful only for listening sockets.)
dt_send	Send an encapsulated message on an open socket.
dt_set_log_size	Set the maximum size of the log file.
dt_shutdown	De-initialize the API.
dt_unlock_print	Release the API's log printing mutex.

Table 1 – Data Tunneling API Functions

dt_close

Prototype

```
int   dt_close(int fd)
```

Parameters

Parameter	Description
fd	File descriptor for the socket connection

Table 2 – dt_close Parameters

Description

The dt_close function closes a socket connection. The passed file descriptor can be the descriptor for a data connection or for a listening socket, as returned by the dt_open function in host mode.

In the case of a:

- Data connection – The connection ceases to exist and data in transit may be lost.
- Listening socket – The listen ceases and the listening thread terminates.

Return Values

`dt_close` always returns `0`.

dt_init

Prototype

```
int  dt_init(int log_optns, char *log_name)
```

Parameters

Parameter	Description
`log_optns`	Logical OR of the following mnemonic options: **Options** / **Description** `DT_LO_ERRORS` — Print errors from failed system calls, allocation failures, etc. `DT_LO_CODE_PATH` — Print details of the execution path through the API code `DT_LO_WARNINGS` — Print warnings about inconsistent data structures `DT_LO_DATA` — Print the data portion of sent and received messages `DT_LO_DEBUG_LEVEL1` — Verbose debugging `DT_LO_DEBUG_LEVEL2` — More verbose debugging `DT_LO_DEBUG_LEVEL3` — Even more verbose debugging
`log_name`	ASCII name of the log file The names `stderr` and `stdout` are treated specially and result in directing the log to those standard file descriptors.

Table 3 – dt_init Parameters

Description

The `dt_init` function – the first API function called by a custom application– establishes the logging options and log name for the duration of the API library use. Usually this is the duration of the execution of the custom application, but you can use the `dt_shutdown` function to terminate use of the API library prior to termination of the custom application.

Return Values

`dt_init` returns `1`, indicating the API is initialized.

dt_lock_print

Prototype

```
void  dt_lock_print(void)
```

Parameters

None

Description

The `dt_lock_print` function acquires the mutex used to single-thread print into the API's log file. Holding this lock across multiple calls to printing routines keeps all the lines in a file together with no lines interspersed from other threads.

dt_open

Prototype

```
int dt_open(char *hp, int mode, int encap,
     int (*call_back)(void *usr_ptr, int fd),
     void *usr_ptr)
```

Parameters

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>hp</td><td>Pointer to an ASCII string denoting the host and port information for the connection

The value of this parameter varies depending upon the mode of the open.</td></tr>
<tr><td>mode</td><td>
<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>DT_O_HOST</td><td>Open the connection in host mode. This implies listening for an incoming connection from some remote host. This option is exclusive with the DT_O_CLIENT option.</td></tr>
<tr><td>DT_O_CLIENT</td><td>Open the connection in client mode. This implies actively connecting to a remote host. This option is exclusive with the DT_O_HOST option.</td></tr>
</table>
</td></tr>
<tr><td>encap</td><td>A mnemonic indicating the type of encapsulation to use for passing data via this connection</td></tr>
</table>

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>hlen_mode</td><td>
<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>HLEN_MODE_DEFAULT</td><td>Use default defined value. Normally set to not include header size in the size field.</td></tr>
<tr><td>HLEN_MODE_INCLUDE_HSIZE</td><td>Include header size in the size field.</td></tr>
<tr><td>HLEN_MODE_EXLUDE_HSIZE</td><td>Do not include header size in the size field.</td></tr>
</table>
</td></tr>
<tr><td>call_back</td><td>Pointer to a routine called at a later time when the connection completes
Used for both incoming and outgoing connections.</td></tr>
<tr><td>usr_ptr</td><td>Pointer passed to the callback routine</td></tr>
</table>

Table 4 – dt_open Parameters

Description

The dt_open function opens a socket-based connection to or from a remote or local host.

The dt_open function creates a thread, if one has not already been created, to monitor for incoming connections or a completed outgoing connection. When either of these events occurs, the thread calls the custom application's callback routine, passing it the usr_ptr and file descriptor for the opened data socket.

When a host mode connection setup completes, the listening thread continues to listen on the indicated port. As a result, there may be a number of thread instances invoking the callback routine with the same usr_ptr and different file descriptor values. Code the callback routine accordingly.

The callback routine is invoked from within the context of the listener or connection setup thread. It should either:

- Create a new thread/process to handle the newly established connection by using clone or fork.
- Or arrange to add the new file descriptor to a parent process's poll list or the equivalent

For incoming connections: Upon success (willingness to accept the connection), the callback routine returns 1 . Upon failure it returns 0.

For outgoing connections: The callback routine is called only upon successful completion of a connection. If an outgoing connection fails to complete, the callback routine is not called and cleanup related to the failure is handled entirely by the outgoing connect completion thread. The custom application may set a time limit for how long to wait before assuming the connection failed to complete.

Return Values

`dt_open` returns either `0` or the value `INVALID_SOCKET`.

`INVALID_SOCKET` does not necessarily indicate an error. If `errno` is set to `EWOULDBLOCK` or `EINPROGRESS`, `INVALID_SOCKET` means the socket is listening (host mode) or opening (client mode).

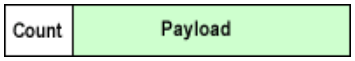
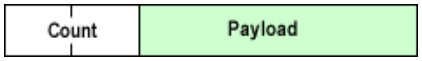
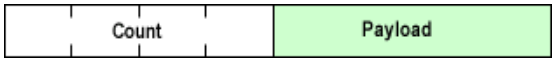
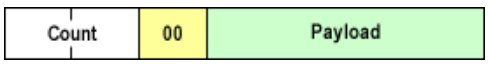
hp and mode Parameters

The `hp` parameter points to an ASCII string that is interpreted differently depending upon the settings of the `mode` parameter:

mode Parameter Value	hp Parameter Interpretation
`DT_O_HOST`	`port` – The API uses the port number as the local port on which to listen for incoming TCP connections. The client must connect to this same port number on this same host to complete the TCP connection. For example: `"9000"`
`DT_O_CLIENT`	`host:port` – • `host` = host name of the remote host and can be a mnemonic or an IP address in dot notation. • `port` = port number to which to connect. For example: `"GPA1:9000"`

Table 5 – hp Parameter Interpretations of mode Parameter for the dt_open Function

Encapsulation Mnemonics for dt_open Function:

encap Parameter Value	Description
`DT_ENCAP_LEN1`	
`DT_ENCAP_LEN2`	
`DT_ENCAP_LEN4`	
`DT_ENCAP_ATALLA`	
`DT_ENCAP_ATALLA_OPT23`	
`DT_ENCAP_BANCNET`	

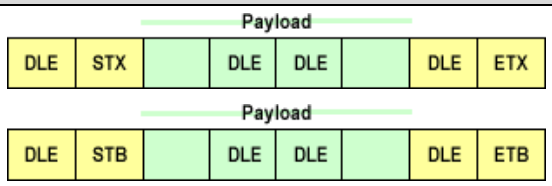
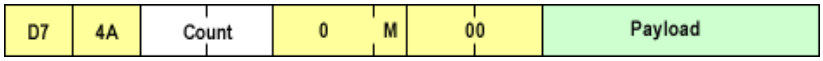
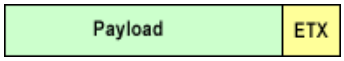
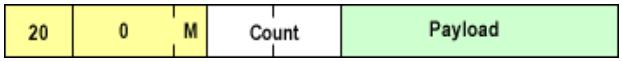
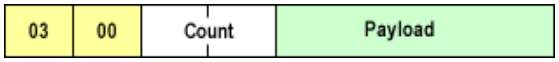
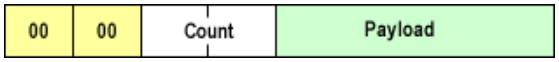
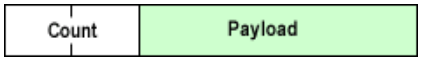
encap Parameter Value	Description
DT_ENCAP_BISYNC	
DT_ENCAP_CISCORBP	
DT_ENCAP_ETX	
DT_ENCAP_GCOM	
DT_ENCAP_RAW	
DT_ENCAP_RFC1006ISO	
DT_ENCAP_RFC1613XOT	
DT_ENCAP_SHELL	
DT_ENCAP_VISA	

Table 6 – Encapsulation Mnemonics for dt_open Function

dt_printf

Prototype

```
void  dt_printf(char *fmt, ...)
```

Parameters

Parameter	Description
`fmt`	Pointer to a character string with format information in the manner of the `printf` library routine
Remaining parameters	Values to be printed according to the format string

Table 7 – dt_printf Parameters

Description

The `dt_printf` function prints a message into the API log file. The location of the log file is set by the `dt_init` function.

dt_recv

Prototype

```
int dt_recv(SOCKET fd, int len, void *data, int *more)
```

Parameters

Parameter	Description
`fd`	File descriptor for the socket connection
`len`	Size of the custom application buffer
`data`	Pointer to the custom application buffer
`more`	Pointer to an integer that can hold the *more data indicator* that may accompany the received message May be NULL.

Table 8 – dt_recv Parameters

Description

The `dt_recv` function:

- Receives an encapsulated data message from the TCP socket.
- Deposits the data portion in the custom application's buffer.
- Returns its length.

When the custom application calls the `dt_recv` function, the function returns 0 if the entire message has not been received. Call the `dt_recv` function again when more data arrives, with the same parameters, to receive the entire message.

If the custom application's buffer is not large enough to hold the received message, the `dt_recv` function returns `–1` and sets `errno` to `ENOBUFS`.

Note: It is first necessary to call `poll` or the equivalent to ensure the file descriptor is read-ready.

The `more` pointer returns the *more data indicator* for encapsulation types using this indicator. If the custom application passes a NULL pointer to the `dt_recv` function, the `more` flag is not returned. If the `more` data indicator is non-zero upon return from the `dt_recv` function, regard the received message as one element of a longer message whose additional parts will be received via subsequent calls to the `dt_recv` function.

Return Values

Upon success, `dt_recv` returns the length of the received data.

Upon failure, `dt_recv` returns a negative number with `errno` set to reflect the error condition. If `errno` is set to:

- `ECONNRESET` – The underlying socket has experienced an error and should be closed by the custom application.
- `EAGAIN` – The incoming message is discarded; the custom application should return to invoking poll.

- ENOBUFS – The custom application buffer is not large enough for the incoming data.

dt_reuseaddr

Prototype

```
void dt_reuseaddr(void);
```

Parameters

None

Description

The dt_reuseaddr function enables setting of the SO_REUSEADDR socket option on listening sockets. After calling this function, any dt_open calls that specify a host mode connection result in enabling the SO_REUSEADDR socket option on the listening socket. This is provided primarily for server development purposes to prevent waiting for the TIME_WAIT time to expire before restarting the server.

Notes:

- If you enable this option, it is possible to receive garbage on the listener from any connections that were active when the server was shut down.
- If the option does not appear to work and you are using Linux, be aware this socket option is buggy in many Linux kernels.

Return Values

None

dt_send

Prototype

```
int dt_send(SOCKET fd, int len, void *data, int more)
```

Parameters

Parameter	Description
fd	File descriptor for the socket connection
len	Number of bytes of data to send
data	Pointer to the bytes of data to be sent after the header
more	Set the *more data indicator* if appropriate

Table 9 – dt_send Parameters

Description

The dt_send function sends encapsulated data on the indicated socket. The len parameter identifies the length of the payload data. The API adds the encapsulation information prior to sending the message on the socket.

If the API cannot send the entire message on the socket because of flow control constraints, the `dt_send` function returns `–1` and sets `errno` to either `EAGAIN`, `EINTR` or `EWOULDBLOCK`. In this case, simply call the `dt_send` function with the same parameters after `poll` indicates that the socket is write-ready again.

Return Values

Upon success, `dt_send` returns the value of the `len` parameter to indicate all bytes have been sent.

Upon failure, `dt_send` returns a negative number with `errno` set to reflect the error condition. If `errno` is set to:

- `ECONNRESET` – The underlying socket has experienced an error and should be closed by the custom application.
- `EAGAIN` – The API cannot send the entire message on the socket because of flow control constraints.
- `EINTR` – The API cannot send the entire message on the socket because of flow control constraints.
- `EWOULDBLOCK` – The API cannot send the entire message on the socket because of flow control constraints.

dt_set_log_size

Prototype

```
void dt_set_log_size(int wrap_point, int max_size)
```

Parameters

Parameter	Description	
`wrap_point`	**Value**	**Description**
	`-2`	Do not change the wrap point.
	`-1`	Set the wrap point to the current position.
	`0`	Set the wrap point to the beginning of the file.
	`>0`	Set the wrap point to this byte offset.
`max_size`	**Value**	**Description**
	`0`	Log can grow without bound.
	`>0`	Log size in bytes before wrap-around.

Table 10 – dt_set_log_size Parameters

Description

The `dt_set_log_size` function sets the boundaries of the log file. You can set the log file to grow without bound or to wrap when it reaches a size you define.

The most common usage: Write some initialization information into the log and then use dt_set_log_size(-1, max_size) to limit log file size and set the wrap point just beyond the initialization information.

dt_shutdown

Prototype

```
void  dt_shutdown(void)
```

Parameters

None

Description

The `dt_shutdown` function terminates all API operations. It kills any internal threads started by the API and closes the log file unless the log file is set to `stderr` or `stdout`.

dt_unlock_print

Prototype

```
void  dt_unlock_print(void)
```

Parameters

None.

Description

The `dt_unlock_print` function releases the mutex used to single-thread print into the API's log file.

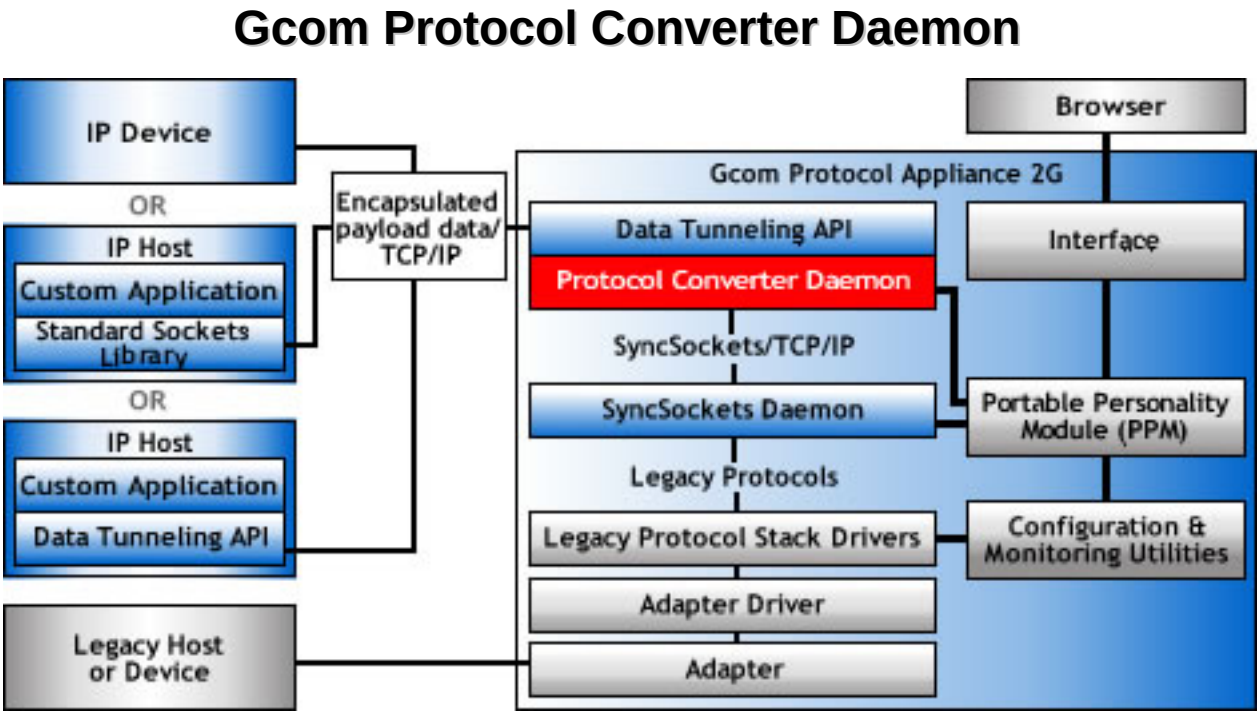


Figure 3 – Gcom Protocol Converter Daemon

The Protocol Converter Daemon is a Gcom-proprietary, stand-alone application-level program that converts legacy protocol data streams from Gcom's SyncSockets protocol to encapsulated payload data over TCP/IP.

The Protocol Converter Daemon uses:

Gcom API…	To…
SyncSockets API	• Manage SyncSockets (socket connections) to legacy protocol connections. • Exchange data with the SyncSockets Daemon, an intermediary to the legacy protocol connections.
Data Tunneling API	• Manage TCP connections to the custom application or IP device. • Perform encapsulation on the payload data.

Table 11 – Gcom Protocol Daemon and Gcom APIs

A single Protocol Converter Daemon can:

- Manage multiple connections – even if those connections are to different legacy protocols.
- Act as a client or a server (listening) for TCP/IP connections to/from the custom application or IP device on a connection-by-connection basis.

- Manage SyncSockets connections to the SyncSockets Daemon on a per-connection basis to initiate legacy connections or to wait for the legacy host/device to initiate connections.

You may run:

- Multiple instances of the Protocol Converter Daemon, each with its own configuration file/command line that identifies a set of non-interfering encapsulated TCP and SyncSockets connections
- Routes for different legacy protocol connections through a single instance of the Protocol Converter Daemon (because the Protocol Converter Daemon is not sensitive to the type of legacy protocol)
- Multiple custom applications with no restrictions on Protocol Converter Daemon usage

Gcom_pcd

`Gcom_pcd` – the Protocol Converter Daemon program – accepts the following command line options and arguments:

Option & Argument	Description	Optional/Required
`-B`	Run `Gcom_pcd` in the background and detach from its controlling tty.	Optional
`-c` *filename*	Name of the configuration file read by `Gcom_pcd`	Required if using a configuration file to configure `Gcom_pcd`
`-d` *mask*	Mask of debug bits	Optional
`-e [+|-]` *encap*	Default encapsulation You may have multiple occurrences of this option on a single command line.	Required if using a command line to configure `Gcom_pcd`
`-f` *filename*	Log file name	Optional
`-h`	Print option list	Optional
`-i` *#1,#2,#3,#4*	Keep alive settings in seconds. #1 is required. #2 - #4 are optional. Zero = infinite. #1 – Number of seconds of idle time before link is dropped. #2 – Send keep alive after #2 seconds. #3 – Number of seconds between keep alive pulse. #4 – Number of failed keep alive responses before considering link down.	Optional
`-n` *number*	Maximum number of connections Default: `100`	Optional
`-p` *msecs*	Poll timeout in milliseconds	Optional
`-q` *file-size*	Log file size in kilobytes	Optional
`-R` *secs*	Number of seconds to wait before reconnecting with the SSD (Default is 30 sec.)	Optional
`-r` *route-spec*	Define a connection route through `Gcom_pcd`. You may have multiple occurrences of this option on a single command line.	Required if using a command line to configure `Gcom_pcd`
`-t` *secs*	Connection setup time	Optional
`-v`	Print version information	Optional

Table 12 – All Gcom_pcd Command Line Options and Arguments

Notes:

- Recommended method for configuring `Gcom_pcd`: Configuration file.
- Gcom's current browser-based interface to the `Gcom_pcd` does not support creation of a configuration file; you must create the configuration file yourself.
- Do NOT mix the `-c` argument with the `-r` /`-e` arguments on a single command line.

Connection Setup and Cloned Connections

The Protocol Converter Daemon tries to synchronize connection setup between the TCP connection and the legacy connection when possible in the following ways:

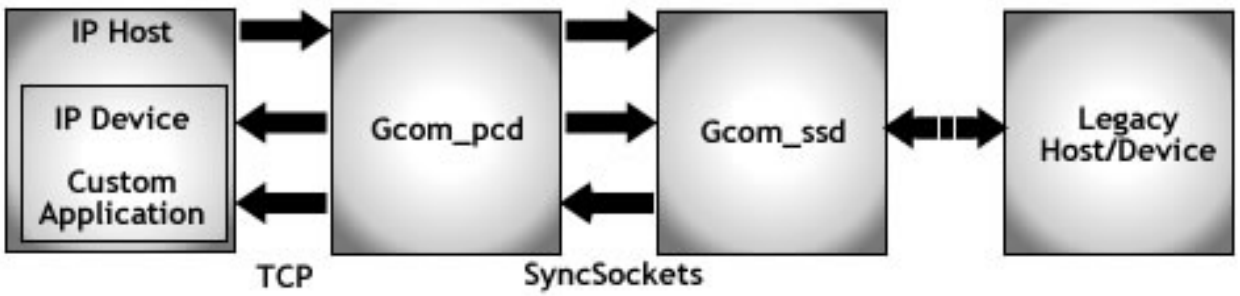


Figure 4 – Connection Initiation

TCP Connection	Legacy Connection	Gcom_pcd Synchronization Steps
Listening	Outgoing	1. Issue `listen` on a socket. 2. When a connection arrives, open a SyncSockets connection to `Gcom_ssd` (the SyncSockets Daemon) and issue an `SS_OP_CONN_REQ`. 3. Receive an `DT_OP_CONN_CONF` from `Gcom_ssd`. 4. Issue an `accept` on the socket.
Outgoing	Outgoing	1. Open the SyncSockets connection and issue an `SS_OP_CONN_REQ` to `Gcom_ssd`. 2. Open the TCP socket and issue a `connect`. 3. Block data flow until both connections are complete. The SyncSockets connection is complete upon receipt of an `DT_OP_CONN_CONF`. The TCP socket connection is complete upon receipt of a poll event.

TCP Connection	Legacy Connection	Gcom_pcd Synchronization Steps
Outgoing	Listening	1. Open a SyncSockets connection and send an `SS_OP_LISTEN` to `Gcom_ssd`. 2. Receive an `SS_OP_CONN_REQ` from `Gcom_ssd`. 3. Issue a `connect` on the TCP connection. 4. If the legacy connection is X.25, open a new SyncSockets connection to `Gcom_ssd` and send an `SS_OP_LISTEN`. 5. When the TCP connection completes, send an `SS_OP_CONN_CONF` to `Gcom_ssd` on the original SyncSockets connection.

Table 13 – Connection Synchronization

You may use a single connection definition to produce multiple actual connections.

All TCP connections can be cloned connections. Once you define an IP address and port number, you can open multiple actual connections to that destination. Similarly, if you are listening on a port number, you can receive multiple incoming connections to that port.

As for the different legacy protocols handled by the SyncSockets Daemon, only X.25 Switched Virtual Circuits (SVCs) can be cloned connections:

- Once you configure a route between an X.25 SVC and a TCP connection, this route can represent multiple actual connections. Keep this in mind when you set parameters for the maximum number of routes in the Protocol Converter Daemon.
- If you define a listening X.25 SVC connection paired with an outgoing TCP connection, the Protocol Converter Daemon keeps a listening SyncSockets connection in place to the SyncSockets Daemon. Thus, multiple incoming X.25 connections to the Protocol Converter Daemon trigger multiple outbound TCP connections.
- If you configure a listening TCP connection paired with an outgoing X.25 SVC connection, the Protocol Converter Daemon holds the listening socket open so that additional TCP connections can arrive and trigger outgoing SyncSocket connections.

Configuration Using the Gcom_pcd Configuration File

Use a `Gcom_pcd` configuration file to tell a Protocol Converter Daemon how to:

- Handle its log file.
- Handle its poll list.
- Set up connections between the custom application and the SyncSockets Daemon.
- Handle the encapsulations on the TCP connections to the custom application.

Each section of a `Gcom_pcd` configuration file consists of a left-justified label followed by indented lines containing parameters. The following table offers a small example, with annotations, that illustrates the general form of the file.

Label/Parameter	Description
`global`: `max_routes = 100`	The first label is always `global`, with parameters that define settings for `Gcom_pcd` as a whole. `max_routes` defines the maximum number of routes that can be: • Defined in the rest of the file • Active at any one time, which can be larger than the number defined in the file

`route.1`:
`left = "tcp_listen"`
`right = "sna_connect"`
`encap = "cisco-rbp"`
	`Gcom_pcd` scans the file for entries named `route.1`, `route.2`, up to `max_routes`. For any entries of that form found in the file, `Gcom_pcd` takes the descriptive information and adds it to its routing table.
	This sample route is:
	• A listening TCP connection using the Cisco RBP encapsulation…
	• Connected to an outgoing SNA connection.
	The connection definitions appear later in the file.

`tcp_listen:`
`host-port ="8001"`
`is_listen = 1`
`conn_name = ""`
	`Left` connection name
	This sample connection is a TCP connection listening on port 8001.

Label/Parameter	Description
sna_connect: host-port = "localhost:8000" is_listen = 0 conn_name = "sna"	Right connection name This sample connection goes to a SyncSockets Daemon on the local machine. Gcom_pcd sends an SS_OP_CONN_REQ to the SyncSockets connection named "sna" to establish this connection.

Table 14 – Gcom_pcd Configuration File Label and Parameter Excerpt

Gcom_pcd Configuration File Global Parameters

The following table shows the available parameters for the configuration file global: entry.

Parameter	Type	Description
connect_timeout	Number	Maximum time in seconds allowed for: • A deferred TCP connection to complete • Or for an SS_OP_CONN_CONF to be returned from Gcom_ssd (the SyncSockets Daemon) in response to Gcom_pcd sending an SS_OP_CONN_REQ. Default: 10
max_routes	Number	Maximum number of routes that can be serviced simultaneously The scanning for "route.n:" labels in the configuration file runs from 1 to this value. Default: 100
max_log_size	Number	Log file size in kilobytes Set to 0 for unlimited log file size. Default: 8000 (which yields an 8-MB circular log file)
poll_timeout	Number	Timeout parameter to use when calling poll Default: 1000 Recommendation: Do not change default value.
reconnect_timeout	Number	Maximum time in seconds at which connection reestablishment occurs. **NOTE**: Must be greater than or equal to connect_timeout. Default: 30

<table>
<tr><th>Parameter</th><th>Type</th><th>Description</th></tr>
<tr><td>reuse_addr</td><td>Number</td><td>
<table>
<tr><th>Value</th><th>Description</th></tr>
<tr><td>1
(Default)</td><td>Set the SO_REUSEADDR socket option on listening TCP connections to prevent the listening socket from entering the TIME_WAIT state when Gcom_pcd is terminated.</td></tr>
<tr><td>0</td><td>Do NOT prevent the listening socket from entering the TIME_WAIT state.</td></tr>
</table>
</td></tr>
<tr><td>run_in_background</td><td>Number</td><td>Set to non-zero to cause Gcom_pcd to run in background mode.</td></tr>
</table>

Table 15 – Gcom_pcd Configuration File Global Parameters

Gcom_pcd Configuration File Route Parameters

The following table shows the available parameters for each configuration file entry of the form route.n:

<table>
<tr><th>Parameter</th><th>Type</th><th>Description</th></tr>
<tr><td>left</td><td>String</td><td>The name of the connection definition for the left connection of the route

This name must appear elsewhere in the configuration file as a label (left justified and followed by a colon character).</td></tr>
<tr><td>right</td><td>String</td><td>The name of the connection definition for the right connection of the route

This name must appear elsewhere in the configuration file as a label (left justified and followed by a colon character).</td></tr>
<tr><td>encap</td><td>String</td><td>A mnemonic indicating the type of encapsulation type to apply to the TCP connection.</td></tr>
<tr><td>hlen_mode</td><td>Number</td><td>Inclusion of header in encapsulation length field
<table>
<tr><th>Value</th><th>Description</th></tr>
<tr><td>0</td><td>Use whatever protocol specification says.</td></tr>
<tr><td>1</td><td>Force header to be included in byte count.</td></tr>
<tr><td>2</td><td>Force header to NOT be included in byte count.</td></tr>
</table>
</td></tr>
</table>

Table 16 – Gcom_pcd Configuration File Route Parameters

Gcom_pcd Configuration File Connection Parameters

The following table shows the available parameters for each `left` and `right` connection name referenced in a `route.n:` entry.

Parameter	Type	Description
`host-port`	String	Format: "*host*:*port*" **For a listening TCP connection**: • *host* = Host name or dotted notation IP address of sending IP host • *port* = IP port number on which to listen **Note:** If *host* = `any` – Listen for an incoming connection from any IP host **For an outgoing TCP connection**: • *host* = Host name or dotted notation IP address of target host • *port* = Target host's port number **For a legacy connection**: • Host name and port number of the `Gcom_ssd` (SyncSockets Daemon) controlling the legacy connection identified in `conn_name` • Usually = "`localhost:8000`"
`is_listen`	Number	Set to `1` for a listening connection. Default: `0`
`conn_name`	String	**For a TCP connection**: Empty string "" **For a legacy connection**: Name of the connection to which `Gcom_pcd` directs an `SS_OP_CONN_REQ` or `SS_OP_LISTEN` message

Table 17 – Gcom_pcd Configuration File Connection Parameters

Configuration Using the Gcom_pcd Command Line

The following command line options are required if you use the command line to configure `Gcom_pcd`:

Option & Argument	Description
`-e [+|-]` *encap*	Default encapsulation You may have multiple occurrences of this option on a single command line.
`-r` *route-spec*	Define a connection route through `Gcom_pcd`. You may have multiple occurrences of this option on a single command line.

Table 18 – Gcom_pcd –r and –e Command Line Options and Arguments

Gcom_pcd –r Command Line Option

Use the `–r` option on a `Gcom_pcd` command line to tell a Protocol Converter Daemon how to:

- Set up connections between the custom application and the SyncSockets Daemon.
- Handle the encapsulations on the TCP connections to the custom application.

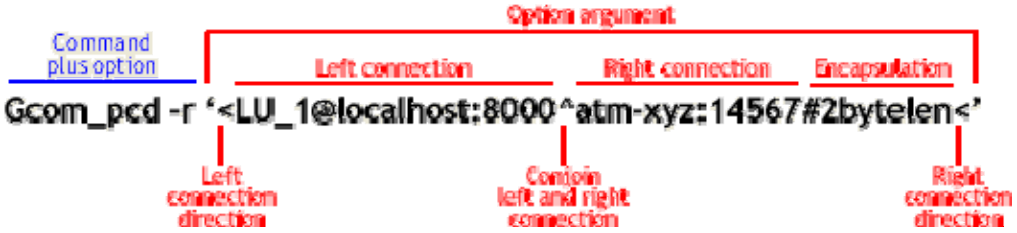


Figure 5 – Sample Gcom_pcd –r Command Line

Component	Purpose
Command line plus `-r` option	Configure a data connection route through a Protocol Converter Daemon. **Note**: You can configure multiple data routes on a single command by coding multiple occurrences of the `–r` option.

Component	Purpose
Option argument	• Identify a TCP connection – IP address, port number, connection direction, and encapsulation technique. • Identify a legacy connection – Name of the connection defined in the `Gcom_ssd` (SyncSockets Daemon) Configuration file, and connection direction. • Bind the two connections together in a data route through a Protocol Converter Daemon. **Recommendation**: Enclose the argument in single quote/apostrophe characters to prevent shell interpretation of the characters within the argument.
Left and right connection	Left connection – Identifies TCP connection parameters or a SyncSockets connection name. Right connection – Identifies TCP connection parameters or a SyncSockets connection name. It doesn't matter if you use left to identify a TCP or SyncSockets connection, or right to identify a TCP or SyncSockets connection, as long as you have one of each connection type. Identify a TCP connection in the following manner: *host*:*port* where: • *host* = Host name or a dotted notation IP address • *port* = IP port number Identify a SyncSockets connection in the following manner: *conn-name@host*:*port* where: • *host:port* = `Gcom_ssd` location • *conn-name* = Name of the legacy connection to target within that `Gcom_ssd`

Component	Purpose
Connection direction	Identify if `Gcom_pcd` should actively connect or passively listen. In the case of a TCP connection, this means `Gcom_pcd` uses either the `socket connect` routine or the `listen` routine. In the case of a SyncSockets connection, this means `Gcom_pcd` sends either an `SS_OP_CONN_REQ` or an `SS_OP_LISTEN` on the SyncSockets connection. Use the `<` or `>` character to identify connection direction for both the left and right connections. • Actively connect = An arrow pointing away from a connection definition • Passively listen = An arrow pointing toward a connection definition In the sample above: • The TCP connection (atm_xyz:14567) passively listens. • The legacy connection (LU_1@loclahost:8000) actively connects.
Encapsulation	Optional – Identifies a specific encapsulation for this data route (and overrides a default encapsulation identified in the `-e` command line option) All encapsulations use some form of header preceding the message payload data that includes a byte count within the header except as noted in Header Byte Count Exceptions. Encapsulation notation formats for the `-r` option: • #*encap* – Include/exclude byte count of the header itself in/from the header length depending upon the protocol specification. • #+*encap* – Always include byte count of the header itself in the header length. • #-*encap* – Always exclude byte count of the header itself from the header length.

Table 19 – Gcom_pcd –r Command Line Components

Gcom_pcd –e Command Line Option

Use the `-e` option on a `Gcom_pcd` command line to tell a Protocol Converter Daemon how to handle the encapsulations on the TCP connections to the custom application.

All encapsulations use some form of header preceding the message payload data that includes a byte count within the header except as noted in Header Byte Count Exceptions.

Encapsulation notation formats for the `-e` option:

- *encap* – Include/exclude byte count of the header itself in/from the header length depending upon the protocol specification.
- +*encap* – Always include byte count of the header itself in the header length.
- -*encap* – Always exclude byte count of the header itself from the header length.

You can use `-e` option multiple times on a single command line, in which case each `-e` option encapsulation applies to all `-r` options (that lack encapsulation information) to the right of the `-e` option until another `-e` option is encountered.

Header Byte Count Exceptions

- The Bisync and ETX encapsulations use bytes at the beginning and the end of each message instead a prefix containing a byte count.
- The Atalla encapsulations communicate with the HP Atalla encryption device and do not actually add and strip the Atalla prefix and suffix characters. These characters are assumed to be present in the legacy protocol data stream and are passed through transparently. Any bytes in the legacy data stream outside the Atalla prefix/suffix are discarded.

Encapsulation Mnemonics for Gcom_pcd

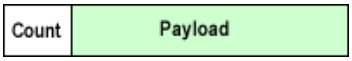
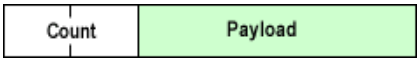
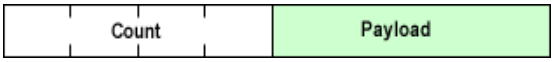
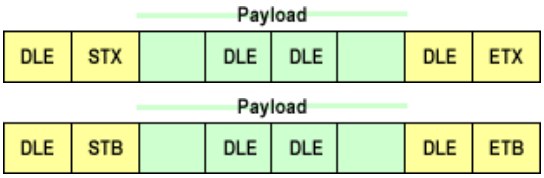
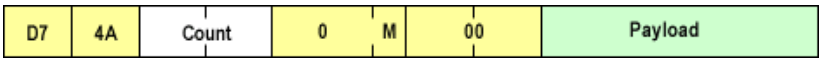
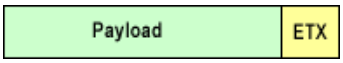
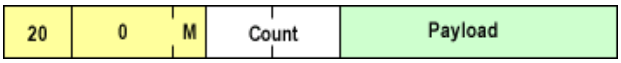
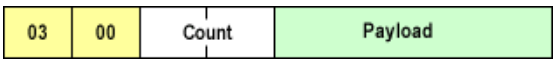
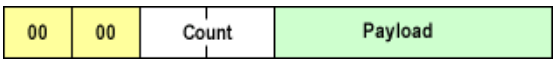
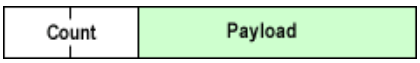
Mnemonic	Description
1bytelen	
2bytelen	
4bytelen	
atalla	
atalla-opt23	
bancnet	
bisync	
cisco-rbp	
etx	
gcom	
raw	
rfc1006-iso	
rfc1613-xot	
shell	
visa	

Table 20 – Encapsulation Mnemonics for Gcom_pcd