## comm.tcl - A remote communications facility for Tcl (7.6, 8.0, and later)

## SYNOPSIS

**package require Comm 3**

*chan* **send** ?*-async*? *id cmd* ?*arg arg ...*?

*chan* **interps**

*chan* **ids**

*chan* **self**

*chan* **connect** ?*id*?

*chan* **config**

*chan* **config** *name*

*chan* **config ?***name value ...*?

-**listen** ?*0|1*? -**local** ?*0|1*? -**port** ?*port*?

*chan* **new** *chan* ?*name value ...*?

*chan* **channels**

*chan* **shutdown** *id*

*chan* **abort**

*chan* **destroy**

*chan* **remoteid**

*chan* **hook** *event* ?**+**??*script*?

The package initializes **comm** as the default *chan*.

## INTRODUCTION

The **comm** command provides an inter-interpreter remote execution facility much like Tk's *send*(n)*,* except that it uses sockets rather than the X server for the communication path. As a result, **comm** works with multiple interpreters, works on Windows and Macintosh systems, and provides control over the remote execution path.

These commands work just like **send** and **winfo interps**:

**comm send** ?*-async*? *id cmd* ?*arg arg ...*?

**comm interps**

This is all that is really needed to know in order to use **comm**.

## DESCRIPTION

**comm** names communication endpoints with an *id* unique to each machine. Before sending commands, the *id* of another interpreter is needed. Unlike Tk's send, **comm** doesn't implicitly know the *id*'s of all the interpreters on the system.

**comm send** ?*-async*? *id cmd* ?*arg arg ...*? This invokes the given command in the interpreter named by *id*. The command waits for the result and remote errors are returned unless the **-async** option is given. **comm self** Returns the *id* for this channel. **comm interps** Returns a list of all the remote *id*'s to which this channel is connected. **comm** learns a new remote *id* when a command is first issued it, or when a remote *id* first issues a command to this comm channel. **comm ids** is an alias for this method. **comm connect** ?*id*? Whereas **comm send** will automatically connect to the given *id*, this forces a connection to a remote *id* without sending a command. After this, the remote *id* will appear in **comm interps**.

These four methods make up the basic **comm** interface.

## EVAL SEMANTICS

The evaluation semantics of **comm send** are intended to match Tk's **send** *exactly*. This means that **comm** evaluates arguments on the remote side.

If you find that **comm send** doesn't work for a particular command, try the same thing with Tk's send and see if the result is different. If there is a problem, please report it. For instance, there was had one report that this command produced an error. Note that the equivalent **send** command also produces the same error.

% **comm send** *id* **llength {a b c}**

**wrong # args: should be "llength list"**

% **send** *name* **llength {a b c}**

**wrong # args: should be "llength list"**

The **eval** hook (described below) can be used to change from **send**'s double eval semantics to single eval semantics.

## MULTIPLE CHANNELS

More than one **comm** channel (or *listener*) can be created in each Tcl interpeter. This allows flexibility to create full and restricted channels. For instance, **hook** scripts are specific to the channel they are defined against.

**comm new** *chan* ?*name value ...*? This creates a new channel and Tcl command with the given channel name. This new command controls the new channel and takes all the same arguments as **comm**. Any remaining arguments are passed to the **config** method. **comm channels** This lists all the channels allocated in this Tcl interpreter.

The default configuration parameters for a new channel are:

**-port 0 -local 1 -listen 0**  The default channel **comm** is created with: **comm new comm -port 0 -local 1 -listen 1**

## CHANNEL CONFIGURATION

The **config** method acts similar to **fconfigure** in that it sets or queries configuration variables associated with a channel.

**comm config**

**comm config** *name*

**comm config ?***name value ...*? When given no arguments, **config** returns a list of all variables and their value With one argument, **config** returns the value of just that argument. With an even number of arguments, the given variables are set to the given values.

These configuration variables can be changed (descriptions of them are elsewhere in this manual page):

-**listen** ?*0|1*? -**local** ?*0|1*? -**port** ?*port*?

These configuration variables are readonly:

-**chan** *chan* -**serial** *n* -**socket** sock*n*

When **config** changes the parameters of an existing channel, it closes and reopens the listening socket. An automatically assigned channel *id* will change when this happens. Recycling the socket is done by invoking **comm abort**, which causes all active sends to terminate.

## ID/PORT ASSIGNMENTS

**comm** uses a TCP port for endpoint *id*. The **interps** (or **ids**) method merely lists all the TCP ports to which the channel is connected. By default, each channel's *id* is randomly assigned by the operating system (but usually starts at a low value around 1024 and increases each time a new socket is opened). This behavior is accomplished by giving the **-port** config option a value of 0. Alternately, a specific TCP port number may be provided for a given channel. As a special case, comm contains code to allocate a a high-numbered TCP port (>10000) by using **-port {}**. Note that a channel won't be created and initialized unless the specific port can be allocated.

As a special case, if the channel is configured with **-listen 0**, then it will not create a listening socket and will use an id of *0* for itself. Such a channel is only good for outgoing connections (although once a connection is established, it can carry send traffic in both directions).

## REMOTE INTERPRETERS

By default, each channel is restricted to accepting connections from the local system. This can be overriden by using the **-local 0** configuration option For such channels, the *id* parameter takes the form **{***id host***}** .

**WARNING**: The *host* must always be specified in the same form (e.g., as either a fully qualified domain name, plain hostname or an IP address).

## CLOSING CONNECTIONS

These methods give control over closing connections:

**comm shutdown** *id* This closes the connection to *id*, aborting all outstanding commands in progress. Note that nothing prevents the connection from being immediately reopened by another incoming or outgoing command. **comm abort** This invokes shutdown on all open connections in this comm channel. **comm destroy** This aborts all connections and then destroys the this comm channel itself, including closing the listening socket. Special code allows the default **comm** channel to be closed such that the **comm** command it is not destroyed. Doing so closes the listening socket, preventing both incoming and outgoing commands on the channel. This sequence reinitializes the default channel: **comm destroy; comm new comm**

When a remote connection is lost (because the remote exited or called **shutdown**), **comm** can invoke an application callback. This can be used to cleanup or restart an ancillary process, for instance. See the **lost** callback below.

## CALLBACKS

This is a mechanism for setting hooks for particular events:

**comm hook** *event* ?**+**??*script*?

This uses a syntax similar to Tk's **bind** command. Prefixing *script* with a + causes the new script to be appended. Without this, a new *script* replaces any existing script. When invoked without a script, no change is made. In all cases, the new hook script is returned by the command.

When an *event* occurs, the *script* associated with it is evaluated with the listed variables in scope and available. The return code (**not** the return value) of the script is commonly used decide how to further process after the hook.

Common variables include:

**chan** the name of the comm channel (and command) **id** the id of the remote in question **fid** the file id for the socket of the connection These are the defined *events*: **connecting**  Variables: *chan id host port*

This hook is invoked before making a connection to the remote named in *id*. An error return (via **error**) will abort the connection attempt with the error. Example:

% comm hook connecting {

if [string match {\*[02468]} $id] {

error "Can't connect to even ids"

}

}

% comm send 10000 puts ok

Connect to remote failed: Can't connect to even ids

%

**connected**  Variables: *chan fid id host port*

This hook is invoked immediately after making a remote connection to *id*, allowing arbitrary authentication over the socket named by *fid*. An error return (via **error**) will close the connection with the error. *host* and *port* are merely extracted from the *id*; changing any of these will have no effect on the connection, however. It is also possible to substitute and replace *fid .*  **incoming**  Variables: *chan fid addr remport*

Hook invoked when receiving an incoming connection, allowing arbitrary authentication over socket named by *fid*. An error return (via **error**) will close the connection with the error. Note that the peer is named by *remport* and *addr* but that the remote *id* is still unknown. Example:

comm hook incoming {

if [string match 127.0.0.1 $addr] {

error "I don't talk to myself"

}

}

**eval**  Variables: *chan id cmd buffer*

This hook is invoked after collecting a complete script from a remote but **before** evalutating it. This allows complete control over the processing of incoming commands. *cmd* contains either **send** or **async**. *buffer* holds the script to evaluate. At the time the hook is called, **$chan remoteid** is identical in value to **id.**

By changing *buffer*, the hook can change the script to be evaluated. The hook can short circuit evaluation and cause a value to be immediately returned by using **return** *result* (or, from within a procedure, **return -code return** *result*). An error return (via **error**) will return an error result, as is if the script caused the error. Any other return will evaluate the script in *buffer* as normal. For compatibility with 3.2, **break** and **return -code break** *result* is supported, acting similarly to **return {}** and **return -code return** *result*.

Examples:

1. augmenting a command % comm send [comm self] pid

5013

% comm hook eval {puts "going to execute $buffer"}

% comm send [comm self] pid

going to execute pid

5013

2. short circuting a command % comm hook eval {puts "would have executed $buffer"; return 0}

% comm send [comm self] pid

would have executed pid

0

3. Replacing double eval semantics % comm send [comm self] llength {a b c}

wrong # args: should be "llength list"

% comm hook eval {return [uplevel #0 $buffer]}

return [uplevel #0 $buffer]

% comm send [comm self] llength {a b c}

3

4. Using a slave interpreter % interp create foo

% comm hook eval {return [foo eval $buffer]}

% comm send [comm self] set myvar 123

123

% set myvar

can't read "myvar": no such variable

% foo eval set myvar

123

5. Using a slave interpreter (double eval) % comm hook eval {return [eval foo eval $buffer]}

6. Subverting the script to execute % comm hook eval {

switch -- $buffer {

a {return A-OK} b {return B-OK} default {error "$buffer is a no-no"}

}

}

% comm send [comm self] pid

pid is a no-no

% comm send [comm self] a

A-OK

**reply**  Variables: *chan id buffer ret return()*

This hook is invoked after collecting a complete reply script from a remote but **before** evalutating it. This allows complete control over the processing of replies to sent commands. The reply *buffer* is in one of the following forms return *result*

return -code *code* *result*

return -code *code* -errorinfo *info* -errorcode *ecode* *msg*

For safety reasons, this is decomposed. The return result is in *ret*, and the return switches are in the return array: *return(-code) return(-errorinfo) return(-errordcode)*  Any of these may be the empty string. Modifying these four variables can change the return value, whereas modifying *buffer* has no effect. **lost**  Variables: *chan id reason*

This hook is invoked when the connection to *id* is lost. Return value (or thrown error) is ignored. *reason* is an explanatory string indicating why the connection was lost. Example:

comm hook lost {

global myvar

if {$myvar(id) == $id} {

myfunc

return

}

}

## UNSUPPORTED

These interfaces may change or go away in subsequence releases.

**comm remoteid** Returns the *id* of the sender of the last remote command executed on this channel. If used by a proc being invoked remotely, it must be called before any events are processed. Otherwise, another command may get invoked and change the value. **comm\_send**  Invoking this procedure will substitute the Tk **send** and **winfo interps** commands with these equivalents that use **comm**.

proc send {args} {

eval comm send $args

}

rename winfo tk\_winfo

proc winfo {cmd args} {

if ![string match in\* $cmd] {return [eval [list tk\_winfo $cmd] $args]}

return [comm interps]

}

## SECURITY

Something here soon.

## BLOCKING SEMANTICS

There is one outstanding difference between **comm** and **send**. When blocking in a synchronous remote command, **send** uses an internal C hook (Tk\_RestrictEvents) to the event loop to look ahead for send-related events and only process those without processing any other events. In contrast, **comm** uses the **vwait** command as a semaphore to indicate the return message has arrived. The difference is that a synchornous **send** will block the application and prevent all events (including window related ones) from being processed, while a synchronous **comm** will block the application but still allow other events will still get processed. In particular, **after idle** handlers will fire immediately when comm blocks.

What can be done about this? First, note that this behavior will come from any code using **vwait** to block and wait for an event to occur. At the cost of multiple channel support, **comm** could be changed to do blocking I/O on the socket, givng send-like blocking semantics. However, multiple channel support is a very useful feature of comm that it is deemed too important to lose. The remaining approaches involve a new loadable module written in C (which is somewhat against the philosophy of **comm**) One way would be to create a modified version of the **vwait** command that allow the event flags passed to Tcl\_DoOneEvent to be specified. For **comm**, just the TCL\_FILE\_EVENTS would be processed. Another way would be to implement a mechanism like Tk\_RestrictEvents, but apply it to the Tcl event loop (since **comm** doesn't require Tk). One of these approaches will be available in a future **comm** release as an optional component.

## COMPATIBILITY

**Comm** exports itself as a package. The package version number is in the form *major*.*minor*, where the major version will only change when a non-compatible change happens to the API or protocol. Minor bug fixes and changes will only affect the minor version. To load **comm** this command is usually used:

**package require Comm 3**

Note that requiring no version (or a specific version) can also be done.

The revision history of **comm** includes these releases:

3.6 A bug in the looking up of the remoteid for a executed command could be triggered when the connection was closed while several asynchronous sends were queued to be executed. 3.5 Internal change to how reply messages from a **send** are handled. Reply messages are now decoded into the *value* to pass to **return**; a new return statement is then cons'd up to with this value. Previously, the return code was passed in from the remote as a command to evaluate. Since the wire protocol has not changed, this is still the case. Instead, the reply handling code decodes the **reply** message. 3.4 Added more source commentary, as well as documenting config variables in this man page. Fixed bug were loss of connection would give error about a variable named rather than the message about the lost connection. **comm ids** is now an alias for **comm interps** (previously, it an alias for **comm chans**). Since the method invocation change of 3.0, break and other exceptional conditions were not being returned correctly from **comm send**. This has been fixed by removing the extra level of indirection into the internal procedure **commSend**. Also added propogation of the *errorCode* variable. This means that these commands return exactly as they would with **send**: comm send *id* break

catch {comm send *id* break}

comm send *id* expr 1 / 0

Added a new hook for reply messages. Reworked method invocation to avoid the use of comm:\* procedures; this also cut the invocation time down by 40%. Documented **comm config** (as this manual page still listed the defunct **comm init**!) 3.3 Some minor bugs were corrected and the documentation was cleaned up. Added some examples for hooks. The return semantics of the **eval** hook were changed. 3.2 A new wire protocol, version 3, was added. This is backwards compatible with version 2 but adds an exchange of supported protocol versions to allow protocol negotiation in the future. Several bugs with the hook implementation were fixed. A new section of the man page on blocking semantics was added. 3.1 All the documented hooks were implemented. **commLostHook** was removed. A bug in **comm new** was fixed. 3.0 This is a new version of **comm** with several major changes. There is a new way of creating the methods available under the **comm** command. The **comm init** method has been retired and is replaced by **comm configure** which allows access to many of the well-defined internal variables. This also generalizes the options available to **comm new**. Finally, there is now a protocol version exchanged when a connection is established. This will allow for future on-wire protocol changes. Currently, the protocol version is set to 2. 2.3 **comm ids** was renamed to **comm channels .** General support for **comm hook** was fully implemented, but only the **lost** hook exists, and it was changed to follow the general hook API. **commLostHook** was unsupported (replaced by **comm hook lost )** and **commLost** was removed. 2.2 The **died** hook was renamed **lost**, to be accessed by **commLostHook** and an early implementation of **comm lost hook**. As such, **commDied** is now **commLost**. 2.1 Unsupported method **comm remoteid** was added. 2.0 **comm** has been rewritten from scratch (but is fully compatible with Comm 1.0, without the requirement to use obTcl).

## SEE ALSO

*send*(n)

## AUTHOR

John LoVerso, John@LoVerso.Southborough.MA.US

*http://www.opengroup.org/~loverso/tcl-tk/#comm*

## COPYRIGHT

Copyright (C) 1995-1998 The Open Group. All Rights Reserved. Please see the file *comm.LICENSE* that accompanied this source, or *http://www.opengroup.org/www/dist\_client/caubweb/COPYRIGHT.free.html*.

This license for **comm**, new as of version 3.2, allows it to be used for free, without any licensing fee or royalty.

## BUGS

* If there is a failure initializing a channel created with **comm new**, then the channel should be destroyed. Currently, it is left in an inconsistent state.
* There should be a way to force a channel to quiesce when changing the configuration.

The following items can be implemented with the existing hooks and are listed here as a reminder to provide a sample hook in a future version.

* Allow easier use of a slave interp for actual command execution (especially when operating in "not local" mode).
* Add host list (xhost-like) or "magic cookie" (xauth-like) authentication to initial handshake.

The following are outstanding todo items.

* Add an interp discovery and name->port mapping. This is likely to be in a separate, optional nameserver. (See also the related work, below.)
* Fix the *{id host}* form so as not to be dependent upon canonical hostnames. This requires fixes to Tcl to resolve hostnames!

This man page is bigger than the source file.

## ON USING OLD VERSIONS OF TCL

Tcl7.5 under Windows contains a bug that causes the interpreter to hang when EOF is reached on non-blocking sockets. This can be triggered with a command such as this:

**comm send $other exit**  Always make sure the channel is quiescent before closing/exiting or use at least Tcl7.6 under Windows.

Tcl7.6 on the Mac contains several bugs. It is recommended you use at least Tcl7.6p2.

Tcl8.0 on UNIX contains a socket bug that can crash Tcl. It is recommended you use Tcl8.0p1 (or Tcl7.6p2).

## RELATED WORK

Tcl-DP provides an RPC-based remote execution interface, but is a compiled Tcl extension. See *http://www.cs.cornell.edu/Info/Projects/zeno/Projects/Tcl-DP.html*.

Michael Doyle <miked@eolas.com> has code that implements the Tcl-DP RPC interface using standard Tcl sockets, much like **comm**.

Andreas Kupries <a.kupries@westend.com> uses **comm** and has built a simple nameserver as part of his Pool library. See *http://www.westend.com/~kupries/doc/pool/index.htm*.

Markup created by *unroff* 1.0,    May 30, 1998.