Reflecting and Transforming Channels

Andreas Kupries ActiveState Software Inc. 409 Granville Vancouver, BC CA
andreask@ActiveState.com

ABSTRACT

This paper describes the history of the reflected and transformed channels exposed at the script level in Tcl 8.5 and higher, provides insight into their implementation, and demonstrates various applications of this feature.

1. OVERVIEW

While the script-level appearance of reflected and transformed channels in Tcl 8.5 and Tcl 8.6 (See TIPs 219 [1] and 230 [2]) make them appear to be a very new feature of the core, the technology underneath is actually quite old with a rich history behind it. This paper was written to shed some light on the technology, its applications, and its history.

The paper is structured as follows: in chapter 2 we provide an anecdotal overview of the history of this feature. In the next chapter, 3 we present its implementation in the Tcl core. Chapters 4 and 5 then discuss how to use this feature, i.e. how to write a channel, and what applications are possible. At last, chapter 6 discusses our conclusions.

2. HISTORY

The first attempts at providing user-specified transformation of data flowing through channels were made in 1996 (Tcl 7.6 was current at that time), by writing new channel drivers in C (the "transformers") which manipulated the data as required and talked to a second channel (the "base"), referenced by either pointer or name (See figure 1).

This quickly ran into trouble. One problem was ownership. Is the base now owned by the transformer or not ? Both possibilities have their drawbacks. In a specific situation this is handled easily, weighing the drawbacks and deciding on one. Not so much if we are trying to work on a general framework for such channels. Another problem, the base is still visible at script level, allowing all sorts of mischief to be done on it the transformer will not be aware of, such as injection of arbitrary data.

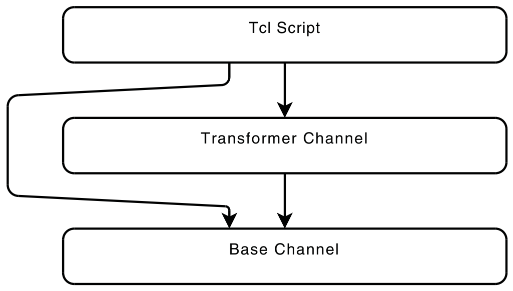


Figure 1: Transform with separate channels

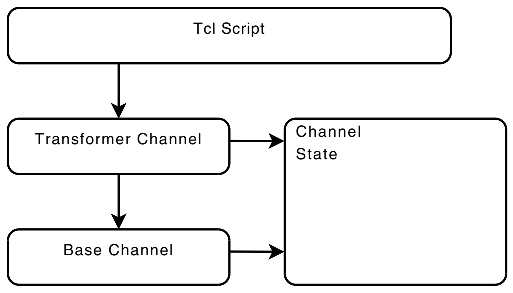


Figure 2: Stackable transforming channels

Because of these issues the framework quickly evolved to have a much tighter coupling between transformer and base, the transformer effectively assuming the identity of the base, see figure 2. Internally this caused a split in the Channel structure, with the identity information and other major state moving to the new ChannelState structure, shared between the base and transformers in the stack.

This model then went forward, with the first series of example transformers becoming the Trf package [16] in Oct 1996, and the necessary low-level support in the core distributed as patches with Trf. The subdirectory "patches/" of the Trf source distribution still contains them, from Tcl 7.6 up to Tcl 8.1b3.

In 1999 the patch and feature was then accepted into Tcl 8.2, under the name "Stacked Channels".

The primary limitation was that transformers had to be written as C extensions, which possibly slowed acceptance and use by the community. After Trf the only packages I am aware of which provide transformers are the Matt Newman's TLS [15] binding to OpenSSL (currently maintained by Dan

Razell and Jeff Hobbs), and my own TrfCrypt [17].

The next step, breaking out of this limitation and exposing transformers at the script level, was already done in 1997, via Trf's "transform" command [19], and again in 1999 with the separate gIOT package [9]. The latter ceased development in 2000. Trf's variant of this on the other hand continued to be developed, notably it became more complex in the area of seeking, allowing transformation ratios and such. For TIP 230 [2] this complexity was abandoned again.

Then nothing more was done for four years, except for the maintenance of Trf, TLS, etc. until near the end of 2004 exposing transformers and base channels was proposed in the TIPs 230 [2] (transformer) and 219 [1] (base). The latter proposed something new, but in the same line of development, i.e. exposing more of the core's internals to the script level. The new feature tracked faster and was integrated into the core August 2005, in time for Tcl 8.5. TIP 230 [2] then languished quite a bit more, with another three year gap between May 2005 and April 2008, until it finally was integrated into the core in June 2008, Tcl 8.6b1.

Now users of the core are free to write both transformers and base channels in Tcl, moving only performance critical parts into C as needed. And one of the first base channels written based on Tcl 8.6, by Colin McCormack for his Wub [18] is a plain identity channel talking to a separate socket channel, to dissociate reused socket names from the channels names seen by the Wub internals. In other words, the very scheme which started us on this road.

We have come full circle.

3. IMPLEMENTATION

The implementation of reflected and transformed channels resides in the two files

generic/tclIORChan.c Reflected Base Channel

generic/tclIORTrans.c Transformed Channel

of the Tcl core. They are part of the general Tcl I/O core and implement, in essence, a channel driver each.

3.1 Driver Basics

At the center of these two drivers is code mapping the calls to the channel driver functions by the Tcl core into invocations of the specified/associated script level handler command and its methods, and then to process the results before delivering them as the driver's result. This processing includes, for example, the unpacking of the returned `Tcl_Obj`'s into C values, and the delivery of errors. For details on the latter see section 3.2.

See the tables 1 and 2 for a quick and compact overview of these mappings. The full details can be found in TIPs 219 [1] and 230 [2], and also in the manpage `[refchan]` [7].

As can be seen, the mapping for transformers is more complex than for base channels. This is due to the necessity of having to handle corner cases introduced by seeking and end of file which do not appear in base channels.

Also note that the semantics of "read" and "write" differ between base and transformer channels, despite the identical method names.

[0]This driver function is actually never called by Tcl's I/O core.

<channel creation>	→"initialize"
closeProc	→"finalize"
inputProc	→"read"
outputProc	→"write"
seekProc	→"seek"
setOptionProc	→"configure"
getOptionProc	→"cget", "cgetall"
watchProc	→"watch"
getHandleProc	Not mapped. See section 3.4.
close2Proc	Not mapped.
blockModeProc	→"blocking"
flushProc	Not mapped.[0]
handlerProc	Not mapped.
wideSeekProc	→"seek"
threadActionProc	Not mapped. See section 3.3
truncateProc	Not mapped.

Table 1: tclrchannel – Driver functions ↔ methods

<transform creation>	→"initialize"
closeProc	→"drain", "flush", "finalize"
inputProc	→"drain", "limit?", "read"
outputProc	→"clear", "write"
seekProc	→"clear", "flush"
setOptionProc	Passed to base.
getOptionProc	Passed to base.
watchProc	Not mapped.
getHandleProc	Passed to base.
close2Proc	Not mapped.
blockModeProc	Not mapped.
flushProc	Not mapped.[0]
handlerProc	Pass up.
wideSeekProc	→"seek"
threadActionProc	Not mapped. See section 3.3
truncateProc	Not mapped.

Table 2: tclrtransform – Driver functions ↔ methods

Method	tclrchannel	tclrtransform
"read"	Read more data and return it	Transform the provided data for reading and return results, if any
"write"	Write the provided data	Transform the provided data for writing and return results, if any

The main point for transformers is that these method always take data for transforming, and return results, if any, possibly partial. The actual reading from and writing to the base is handled at the C-level. In the case of base channels there is no base, and the script-level has to implement what it means when to read or write data.

3.2 Error delivery

The delivery of errors reported by the script level handler of a reflected channel is another significant portion of the code, because it can raise arbitrary messages, whereas the channel driver functions as designed at the time of their introduction basically deal only with POSIX error codes, i.e. "errno", and nothing else.

A redesign of the API to incorporate proper error delivery was not possible, because it is a public part of the core's API and had to stay backwards compatible.

The solution was to introduce four additional functions in the public API (see listing 1) to store and retrieve the full error information in either per-channel or per-interp state, and then modify the implementations of the higher functions in Tcl's I/O core to query this state before falling back to the delivery of a POSIX based error. This change enables all channel drivers inside or out of the core to report arbitrary errors, although only the drivers for the reflected channels make use of this facility at this time.

Listing 1: API for the delivery of arbitrary errors

```
void Tcl_SetChannelError
     (Tcl_Channel chan, Tcl_Obj* msg)

void Tcl_SetChannelErrorInterp
     (Tcl_Interp* ip,    Tcl_Obj* msg)

void Tcl_GetChannelError
     (Tcl_Channel chan, Tcl_Obj** msg)

void Tcl_GetChannelErrorInterp
     (Tcl_Interp* ip,    Tcl_Obj** msg)
```

In essence these functions construct and maintain an area through which information can be passed between driver and higher layers which passes the narrow API of the driver functions themselves by, as shown in figure 3. This is the source of the references to the "bypass area" in various comments throughout the implementation. The solution is not very elegant, but anything else would have requires an incompatible redefinition of the whole channel driver structure and of the driver functions.

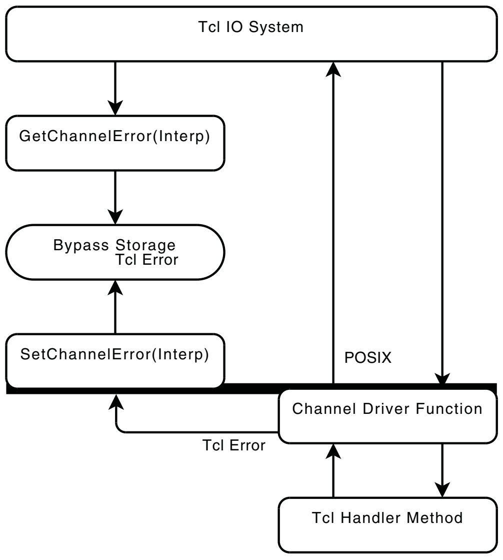


Figure 3: Delivery of arbitrary errors, bypassing the narrow API

Further things of note

1. The usage of `Tcl_Obj`'s for the information storage binds the information to a single thread, making a transfer across thread boundaries impossible.

2. The "msg" argument of the function does not contain a plain string, but has to be a list of uneven length.

   The last element is interpreted as the actual error message in question, and the preceding elements are considered as option/value pairs containing additional information about the error, like the errorCode, etc. I.e. they are an extensible dictionary containing the details of the error beyond the basic message.

   As a safety precaution any -level specification submitted by the driver and a non-zero value is rewritten to a value of 0 to prevent the driver from being able to force the user application into the execution of arbitrary multi-level returns, i.e. from arbitrarily changing the control-flow of the application itself. Analogously any -code specification with a non-zero value which is not error is rewritten to value 1 (i.e. error).

3. All the functions are necessary. The regular variants because many driver functions have only a Channel*. The Interp variants, because some of the driver functions, notably "closeProc", have no channel to put their error messages into.

3.3 Threading

An important part of the implementation and the design was the ability to handle threading, and the movement of a base or transformed channel between threads. While the latter cannot move by themselves they are moved together with their base channel, should it be moved.

The issue is that while during a move of a reflected channel C from Thread A to thread B the Channel structure can and is moved to B the handler command servicing C cannot move. Because it depends on an unknown set of state information held by A, and is also supported by an unknown number of packages loaded into A. Unknown to the I/O core that is, making it impossible for the core to move the handler.

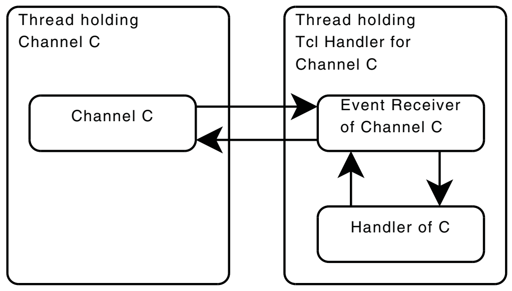
Instead of simply forbidding the movements of reflected channels a different solution was found.

First, instead of just keeping a reference to the Tcl interpreter holding the handler command and its state we keep a reference to its thread as well. Then, whenever a driver function is invoked and finds that it is called from a different thread it uses the event system to forward its method invocations to the proper thread for processing. In the implementations all identifiers having "Forward" as prefix are related to this.

It is this case-by-case checking of thread-location in the driver functions which makes the use of "threadActionProc" driver function superfluous, which is why it is not mapped to anything, see the tables 1 and 2.

Important things to be aware of:

1. If the thread holding the handler command is not processing events then the channel in question will block waiting for the result of its method invocation, stopping the sending thread as well. This may lead to dead-locks. Beware!



Thread holding Channel C
Thread holding Tcl Handler for Channel C
Channel C
Event Receiver of Channel C
Handler of C

Figure 4: Reflection in Threads

This cannot be avoided, because the event system has to be used for this, we have no other system in the Tcl core for cross-thread execution of Tcl scripts and commands.

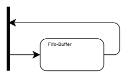
2. When a thread or interpreter is deleted all reflected channels created in this thread/interpreter are deleted as well, in all interpreters they have been shared with or moved into, and in whatever thread they have been moved to. While this pulls the rug out under the other thread(s) and/or interpreter(s), this cannot be avoided. Trying to use such a channel will cause the core to throw errors about unknown channel handles.

3.4 Limitations

The base channels suffer from one limitation inherent in their design. As they are virtual, i.e. in memory, and, more importantly, not known to the operating system they do not have an OS-specific handle either. It is because of this that their getHandleProc is not implemented, see table 1, but another consequence is that they cannot be used as targets for I/O redirection in command pipelines created by either `[exec]` or `[open|]`.

4. WRITING A REFLECTION

4.1 Base channel – Fifo



In this section we will now walk through and explain the implementation of a base channel using the reflection API. The full sources of this channel can be found in
`examples/walkthrough/fifo.tcl`
The class in question provides a channel with the same semantics as Memchan's [10] fifo channel.

```
proc ::fifo {} {
    return [::chan create {read write} \
                [fifo::implementation new]]
}
```

```
oo::class create ::fifo::implementation {
```

First a convenient wrapper around the creation process, so that the user can create new fifos in the most simple way possible, i.e. a single command invocation, no arguments. The "new" method is used because do not really care about the name of the object which is our handler command for the channel.

```
    constructor {} {
        set data {}
        array set allowed {read 0 write 0}
        set requested {}
        set delay 10
        return
    }

    destructor {
        if {$channel eq {}} return
        close $channel
        return
    }
```

The life-cycle management is pretty simple. On construction we initialize our state variables, and on destruction we close the channel, except if we were called from the "finalize" method. This is explained when we come to that method too.

```
    method initialize {c mode} {
        set channel $c
        my Allow write
        return {
            initialize finalize watch
            read write
            configure cget cgetall
        }
    }
```

This is the first method the IO system will call. Its result is expected to be a list of all the methods the handler supports. In a proper framework this list would likely be computed automatically in some way.

The first two shown here are mandatory. The third, "watch", is also pretty much mandatory. While it can be left out if the channel does not have to support file events, I can't think of any channel which would not.

As our channel is both readable and writable we have to declare this here, and provide the necessary methods (see below). As fifos are always created read/write, as shown in the `[::fifo]` wrapper command above, we can ignore the mode argument. Otherwise this tells us if user wishes the channel to be readable, writable, or both[1].

The last three methods we declare here as supported are for the handling of channel options. The methods "cget" and "cgetall" always go together, if present channel options can be queried. The "configure" method can be used alone, and enables the setting of channel options. In most cases we wish to support both setting and querying options, requiring all three of them.

Our channel here supports a single option "-delay" with which the user can configure the interval between events generated by the channel, in milliseconds.

```
    method finalize {c} {
        my Disallow read write
        set channel {}
        my destroy
        return
    }
```

This is the last method the IO system will call, when closing the reflected channel. We have to stop the generation of any events which may still be active at this point, and then destroy the handler object.

The channel variable is reset to prevent the destructor (shown later) from recursively calling `[close]`. This means

[1]List of "read", "write"

that the code here allows the closing of the fifo through the destruction of the handler object as well. One could think that this does not matter given that the handler is an anonymous object whose name is not made public when the channel is created. Not true however, as the object destruction can also be triggered indirectly, through the destruction of the containing namespace, interpreter, thread, or process.

```
method configure {c option value} {
    my CheckOption $o
    my CheckDelay  $value
    set delay $value
    return
}

method cget {c option} {
    my CheckOption $o
    return $delay
}

method cgetall {c} {
    return [list -delay $delay]
}

method CheckOption {o} {
    if {$option eq {-delay}} {
        return
    }
    return -code error \
      "Unknown option $option, expected -delay"
}

method CheckDelay {value} {
    if {[string is integer -strict $value] &&
        ($value > 0)} {
        return
    }
    return -code error \
      "Expected positive integer, got $value"
}
```

As said above, the fifo supports a single option "-delay". The code of the methods to handle setting and querying it, including the error handling, should be pretty much self-explanatory.

```
method read {c n} {
    set read [string range $data 0 ${n}-1]
    set data [string range $data $n end]
    if {$data eq {}} {
        my Disallow read
    }
    if {$read eq {}} {
        return -code error EAGAIN
    }
    return $read
}

method write {c bytes} {
    append data $bytes
    if {$data ne {}} {
        my Allow read
    }
    return [string length $bytes]
}
```

For the sake of simplicity of implementation a single scalar variable "data" is used to hold the bytes currently written to the fifo and not yet read back. In a more performance-conscious implementation we would manage at least two buffers and a read index to avoid copying data as much as possible. For an example of that see the fifo in file `examples/base/fifo.tcl`.

An important thing to remember, the buffer given to "write" will contain a byte-array, and the same is expected as the result of "read". We are working with bytes here, not characters.

The interesting parts, from the perspective of file event support are the checks of "data" being empty, and the subsequent allowance signals to the event manager for readable events. This is done because when the fifo falls empty it must not generate readable events any longer, and when data is added we can start again with that. Assuming, of course, that Tcl's IO system requested them via a call to "watch".

Another important part is the check for the read result being empty. When the fifo is empty we are **not** at EOF, as more data can be added to the channel later. This is signaled through the error. Just returning the empty string in this situation would tell the IO system that the channel has reached EOF.

```
method watch {c requestmask} {
    if {$requestmask eq $requested} {
        return
    }
    set requested $requestmask
    my Update
    return
}
```

This is the main entry point for the handling of file events. Through this method the IO system declares the set of events it is ready to receive. We ignore calls which do not change anything, then remember the information in the event managers state and at last combines this information with the set of events the channel is able to generate at this point. More information later, when we are looking at the variables and methods of the event manager itself, i.e. "Allow" and after.

```
variable \
    data \
    channel \
    timer \
    delay \
    allowed \
    requested \
    posting
```

Now we are coming to innards for the handling of file events. As this channel is entirely in memory, with no connection to the OS as a source of events it is necessary to generate any events by ourselves. Which of the events are posted, if any, depends on two pieces of information, namely

allowed the set of events the channel is able to generate per its own state,

requested and the set of events the IO system is ready to receive.

The first we manage as an array variable which maps the event names to a boolean flag, where True indicates that the channel is able to generate the associated event. The second is simply a copy of the list of events given as argument to the method "watch". The intersection of both, again a list, is stored in the variable "posting". See also figure 5 showing the flow of data just described, and the methods involved.

The other three variables hold the Tcl handle of the channel the events will be posted to, the handle of the currently

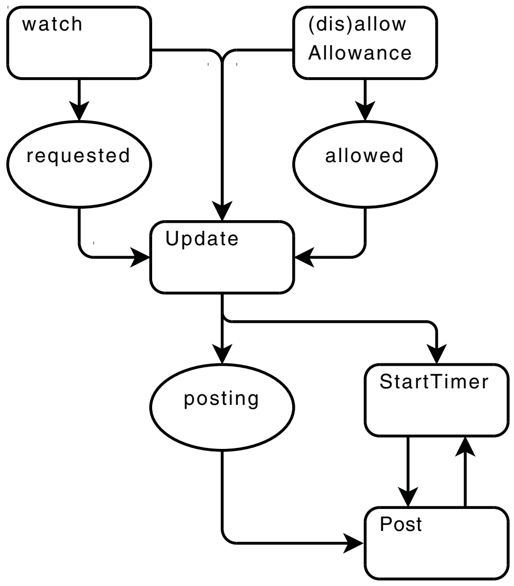


Figure 5: Event management dataflow

pending timer, if any, and the interval between timer invocations, in milliseconds. The first is initialized by method "initialize", the last has a default of 10 milliseconds set during construction and is accessible through the channel option. The timer is managed by the upcoming methods "Update" and "Post".

```
method Allow {args} {
    my Allowance $args yes
    return
}

method Disallow {args} {
    my Allowance $args no
    return
}

method Allowance {events enable} {
    set changed no
    foreach event $events {
        if {$allowed($event) != $enable} {
            set allowed($event) $enable
            set changed yes
        }
    }
    if {!$changed} return
    my Update
    return
}
```

The first two methods in this block are the API by which the channel implementation signals the event manager which events it is able to generate. They are just thin shim over the third method which simply checks the requested against the current state, setting only the actual changes. At last this method, like "watch", calls the method "Update", i.e.

```
method Update {} {
    catch { after cancel $timer }
    set posting {}
    foreach event $requested {
        if {!$allowed($event)} continue
        lappend posting $event
    }
    if {[llength $posting]} {
        my StartTimer
    } else {
        catch { unset timer }
    }
    return
}
```

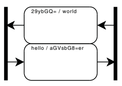
which combines the two sources of information, "allowed" and "requested", into "posting", see also figure 5. If we have events to post after that a periodic timer is started to do so. To prevent multiple timers from going on in parallel any pending timer is canceled. This automatically also handles the case when no events can be posted, by just killing any pending timer.

```
method Post {} {
    my StartTimer
    chan postevent $channel $posting
    return
}

method StartTimer {} {
    set timer \
        [after $delay \
            [namespace code \
                [list my Post]]]
}
```

At last we have the method which is called when our timer triggers. It uses the standard pattern for re-scheduling itself to make the timer periodic, and beyond that uses `[chan postevent]` to inject the events into the IO system.

4.2 Transform channels – Base64



In this section we will now walk through and explain the implementation of a transformation channel using the reflection API. The full sources of this channel can be found in
`examples/walkthrough/base64.tcl`

The class in question provides a transformation using base64 (RFC 4648 [6]) to en- and decode the data flowing through a channel. It uses Tcl 8.6's new `[binary encode/decode base64]` facility (see TIP 317 [4]).

```
proc ::base64 {chan} {
    ::chan push $chan [base64::implementation new]
    return
}

oo::class create ::base64::implementation {
```

First a convenient wrapper around the creation process, so that the user can set the transformation up in the most simple way possible, i.e. a single command invocation, no arguments. This is equivalent to what we did for the fifo base channel shown in the previous section.

```
    constructor {} {
        set encodebuf {}
        set decodebuf {}
        return
    }

    destructor {
        if {$channel eq {}} return
        close $channel
        return
    }
```

The life-cycle management is pretty simple. On construction we initialize our state variables, and on destruction we close the channel, except if we were called from the "finalize" method. This was explained at that method too.

```
    variable channel encodebuf decodebuf
```

The state of the transformation is pretty simple as well. First a reference to the channel it is stacked on, used to control the interaction of destructor and "finalize", then the two buffers to hold partial, not-yet-transformed data.

```
    method initialize {c mode} {
        set channel $c
        return {
            initialize finalize
            read write
            clear drain flush
        }
    }
```

This is the first method the IO system will call. Its result is expected to be a list of all the methods the handler supports. In a proper framework this list would likely be computed automatically in some way.

The first two shown here are mandatory.

As we want our transformation to support both reading and writing we have to declare this here, and provide the necessary methods (see below).

As our transformation has to deal with partial data we declare the last two methods as well, as extensions to "read" ("drain") and "write"-handling ("flush"), and provide their implementations (see below).

```
    method finalize {c} {
        set channel {}
        my destroy
        return
    }
```

This is the last method the IO system will call, when closing the channel the transformation is stacked on. We have to clean up and release any resources used by the transformation which may still be active at this point, and then destroy the handler object. In our example we have no resources to free.

The channel variable is reset to prevent the destructor (shown later) from recursively calling `[close]`. This means that the code here allows the closing of the channel the transformation is stacked on through the destruction of the handler object as well. One could think that this does not matter given that the handler is an anonymous object whose name is not made public when the channel is created. Not true however, as the object destruction can also be triggered indirectly, through the destruction of the containing namespace, interpreter, thread, or process.

```
    method write {c data} {
        my Code encodebuf encode $data 3
    }

    method read {c data} {
        my Code decodebuf decode $data 4
    }

    method Code {bufvar op data n} {
        upvar 1 $bufvar buffer

        append buffer $data

        set n [Complete $buffer $n]
        if {$n < 0} {
            return {}
        }

        set result \
            [binary $op base64 \
                [string range $buffer 0 $n]]
        incr n
        set buffer \
            [string range $buffer $n end]

        return $result
    }

    method Complete {buffer n} {
        set len [string length $buffer]
        return [expr {(($len / $n) * $n)-1}]
    }
```

The "read" and "write" methods are structurally pretty much identical. The differences are in the en/decoding direction, buffer used to hold partial data, and some constants.

Each extends their internal buffer of partial data with the data offered for transformation by the C-level, then determines how much full data is ready to be transformed, runs the transformation on them and returns the result of that after removing the transformation input from its buffer.

This leaves the C-level of transformations with some transformed data (possibly nothing!), and the remaining partial data in the buffer.

An important thing to remember, the buffers given to "read" and "write" will contain byte-arrays. We are working with bytes here, not characters. This is true for "flush" and "drain" as well.

```
    method flush {c} {
        set data \
            [binary encode base64 $encodebuf]
        set encodebuf {}
        return $data
    }

    method drain {c} {
        set data \
            [binary decode base64 $decodebuf]
        set decodebuf {}
        return $data
    }
```

In some situations even buffered partial data **must** be transformed. This is handled by the methods "flush" and "drain", for the write- and read-side of the transformation, respectively.

If a transformation doesn't support these methods then that is an indication that the transformation doesn't have to care about partial data and everything will be done in the "read" and "write" methods. Examples for this latter are the counter, observe, and one-time-pad transformations mentioned in the next section.

```
method clear {} {
    set decodebuf {}
    return
}
```

This method is used to discard partial data on the read side of a transformation. This is done when upon `[seek]`, and when writing to the channel, as this is an implicit seek.

This brings us to an issue which was not mentioned in the walk-through so far, the handling of seeking. Its handling can be seen as either a limitation of transformations, or an application of the KISS principle.

The Tcl command handler for a transformation can sort of infer when the channel it is stacked on is seeked by the user by looking for invocations of "flush" and "clear". This however is not unambiguous and definitely not a supported mode of operation. In general transformations should be seen as being unaware of seeking.

At the C-level seeking is essentially a pass-through to the base channel of the whole stack, using the methods "flush" and "clear" to reset transformation internals. Ditto for `[tell]`, except that it does not reset internals. In other words, for a channel with reflected transformations on top `[seek]` modifies, and `[tell]` reports, the **physical** location in the file or other base channel, and not some virtual location in the transformed stream.

While doing the latter is possible in principle, it needs support from the transformations, and adds considerable complexity to the internals of handling transformations. The Trf package is an example of both the possibility, and the complexity of this support [20].

At last note that C-level transformations can intercept seek requests and implement their own semantics as they see fit, so the above is not the general picture, just for reflected transformation. See again Trf for an example of more complex seek semantics.

5. EXAMPLE CHANNELS AND IDEAS

This section provides examples demonstrating various applications for reflected channels, and ideas for more. All examples use Tcl 8.6's OO facility.

5.1 Base Channels

A base channel is in essence a bridge between Tcl's I/O system and whatever we are able to bend into being represented by either a block or (in)finite stream of bytes (effectively files and pipes).

The simplest possibilities for this are

1. Another channel, and we can do anything to the data read from and/or written into it. This is the original idea which started us on our path, as detailed in section 2 . Anything from section 5.2 applies here as well.

   This is what Colin McCormack actually does in his Wub server [18], with the intent to decouple the socket names as generated by the core from the names seen by his code[2].

   A copy of his code can be found online [21], and under

   - `examples/colin/Chan.tcl`

2. Then, of course, channels which do essentially nothing, i.e. null and zero devices. These cannot be used in command pipelines tough, see section 3.4 for the explanation.

   - `examples/base/null.tcl`,
   - `examples/base/zero.tcl`, and
   - `examples/base/nullzero.tcl`.

3. Next, deliver random bytes when reading, using any of the many possible random number generators. This should be useful in fuzz tests.

   - `examples/base/random.tcl`

   The implementation uses a very simple linear feedback shift register. We are not using `[expr]`'s rand(), and srand() functions because they are a global random generator, and usage would be shared across all random channels in existence. We wish to have separate channels however, each independently seed- and usable.

Another series of relatively obvious possibilities, have the channel represent something which is in memory, i.e. part of the state of the Tcl interpreter.

1. A fixed string configured at construction time. In essence a read-only in-memory file.

   - `examples/base/string.tcl`

2. The content of some variable. This may be random access or more like a queue and/or fifo, depending on the exact implementation of the method for reading, writing, and seeking. Further, the variable used for the storage may be part of the channel, or outside of it.

   - `examples/base/fifo.tcl`,
   - `examples/base/memchan.tcl`, and
   - `examples/base/variable.tcl`.

3. A text widget. This could be random access, or stream based for a log or terminal emulation[3].

   - `examples/base/textwindow.tcl`

   It should be noted that the current implementation plays a bit fast and loose with the data. Unfortunately it cannot avoid doing so. The problem is in the encodings. The "write" method is given bytes, which may encode characters, as per the current channel encoding. For display these have to be converted back into characters. This conversion breaks down in case of multi-byte characters (like utf8). Partial characters at the end of the buffer are semi-lost, because `[encoding convertfrom]` processes only the complete characters

[2]Tcl reuses names for sockets (as the OS handle is part of the name), whereas Colin wished to have unique names which are not reused ever.

[3]Terminals take a stream of bytes, interpreting them as characters and embedded commands, like ANSI color codes.

fully, and the last partial one is considered invalid and replaced by a fallback. We are not given an indication that this happened at script level, nor the option to save the last partial character for a future call. Which means that the remainder of the partial character at the beginning of the next buffer will be another invalid character, and, depending on the encoding, may even completely break the conversion from then on.

Further note that while the full API of reflected channels looks to be quite complex for the general case, for streams it is possible to encapsulate it with only two or three elements exposed, for a radically simpler form:

1. A single method to put data into the channel which can then be read from it.
2. A single method to set a callback to be invoked whenever data has been written to the channel.
3. A single method to set a callback to be invoked whenever the channel changes between empty and not-empty states.

This makes it possible to create more complex communication networks from any number of such 1/2-pipes. The simplest is a full bidirectional pipe with two heads. This enables, for example, stream-based communication between threads, with the actual transfer of data hidden in the C-level of the reflected channels (Remember section 3.3).

- `examples/base/halfpipe.tcl`, and
- `examples/base/fifo2.tcl`

5.2 Transform Channels

In contrast to the base channels demonstrated in the previous section transform channels are all about manipulating the data flowing through them.

The simplest possibilities for this are

1. To do nothing.
   - `examples/transform/identity.tcl`
2. Or, don't change the data but divert it (or a just copy) somewhere else. In general, just an observer, be it of the data itself, or to compute information derived from it, like the number of bytes, words, i.e. statistics, checksums, etc.
   - `examples/transform/observe.tcl`
   - `examples/transform/counter.tcl`
   - `examples/transform/adler32.tcl` [25]
3. Similar to the last item, do not perform modifications, but limit how much data is let through it. I.e. let the transformation introduce an artificial end-of-file signal via the result returned by method "limit?". This can be based on size, patterns in the data, externals signals, etc.
   - `examples/transform/limitsize.tcl`

Beyond this we have the choice among a vast richness of algorithms which actually change the data going through the transformation, giving truth to the name, some of which are listed below, as general categories, and a few more specific examples in the categories.

1. Encodings.

   For example transfer encodings like base64 (RFC 4648 [6]), uuencode, ascii85, but also character encodings. Tcl's IO system handles these usually on its own, but there may be situations where it makes sense to do this deeper, in a transformation.
   - `examples/transform/base64.tcl`
2. Encryption.

   Lots of specific algorithms are possible in this category. Tcllib [14] for example provides implementations of DES, AES, Blowfish, and RC4.

   Other sources of encryption algorithms to wrap in transformations are OpenSSL [22], either directly, or indirectly through TLS [4] [15], and Steve Lander's CryptKit [23] for CryptLib [24].

   To keep the code accompanying the paper relatively self-contained our example is the one-time-pad. It simply xors the data flowing through it with the key bytes it takes from two other channels (One per direction, read and write).
   - `examples/transform/otp.tcl`
3. Compression.

   A nice example for this is zlib, which is also part of Tcl 8.6 as per TIP 234 [3]. The transformation support the TIP specifies is implemented at the C-level, however we can implement a simpler form[5] as reflection using the stream API.
   - `examples/transform/zlib.tcl`

   This transformation also shows us another limitation of the transformation system, actually of the whole IO system.

   To perform packet- or message-based compression it is necessary to pop and re-push the transformation after every packet/message to properly flush the compressed data to the underlying channel. This causes us to lose the whole compressor state, and reducing the effectiveness of the compressor. This despite the fact that the zlib library supports an API through which it can be forced to flush partially compressed data without losing its state. Compression effectiveness is reduced in that mode too, but not as severely as caused by a full restart.

[4]While its functionality is implemented as a C-level transformation, it might make sense to re-implement it as a reflected transformation, with the (possibly event-driven) initial key and cipher negotiation parts separated from the actual transformation.

[5]The C implementation in the core provides additional data useful to scripts, like the crc, the gzip header data, etc.

Without going into the full details (which can be found in the comments at [8]), the main problem is that drivers do not have information about flushes performed by the user, they only see a series of write requests. This prevents any special behavior we might wish to perform on a flush, like for zlib above.

4. Error correction.

   An example for this category, albeit implemented at the C-level, is Trf's Reed-Solomon Coder command (`[rs_ecc]`).

5. Chunking.

   The example transformation inserts and removes a configurable string every n characters, again configurable.

   - `examples/transform/spacer.tcl`

   Pat Thoyts is using them in his work on the core's http package to transparently handle HTTP's chunked data transfer mode.

5.3 File Transfer Example

This section describes a larger example which brings most of the channels and transformations discussed in the previous two sections together into a toy application for the "secure" transfer of files.

The core of the sender part can be seen in figure 6. A `[fcopy]` command moves the contents of a "string" channel into a socket which sends it to some receiver. This transfer is secured by a "one-time-pad" transformation (OTP) pushed on the socket and encrypting the data. The key bytes for the OTP are read from an instance of the "random" base channel type. Transmitter and receiver do a toy key exchange protocol[6] first, ensuring the identical initialization of their "random" channels. This is not part of the transforms, and thus not shown in the figure either.

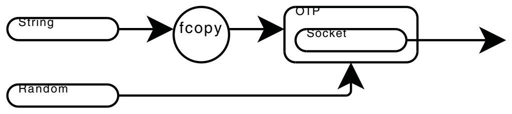


Figure 6: Transmitter core

The receiver, seen in figure 7, simply operates in reverse, and uses a "variable" channel to store the incoming text.

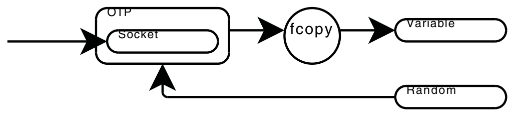


Figure 7: Receiver core

Now, as we not only wish to perform the transfer in our demonstration, but also see the operation of the internals "observer" transformations are added at strategic points, feeding "textwindow" channels visibly logging the bytes. As the deeper layers carry binary data additional "hex" and "spacer" transformations are used to format the data into something more legible.

[6]This is why the application is a only a toy. In a real application something like Diffie-Hellman [26] would be used instead. Even better would be the use of TLS/SSL instead of inventing their own crypto.

The resulting transmitter can be seen in figure 8. The extended receiver looks analogous.

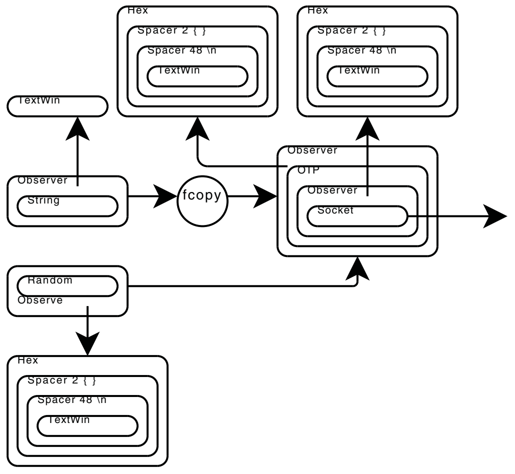


Figure 8: Extended transmitter core, with observers

Going back to the key exchange, while the actual protocol is a toy, the architecture used for it and its interaction with the encryption transformation itself, or rather its initialization, is of general interest. The listing below shows the relevant part of the code.

```
proc crypt {chan} {
    set key [keyexchange $chan]

    tcl::transform::otp $chan \
        [tcl::chan::random $key] \
        [tcl::chan::random $key]
}

proc keyexchange {chan} {
    set myseed [tcl::randomseed]

    puts  $chan $myseed
    flush $chan

    set peerseed [gets $chan]
    set key      [tcl::combine \
                      $myseed \
                      $peerseed]
    return $key
}
```

The main point in the code above is the existence of two distinct phases.

The first is running outside of the transformation, actually even before the transformation exists, i.e. is pushed. It negotiates all the configuration the transformation later needs. Here this is just the key. In a proper protocol this would also be stuff like the cipher to use, exchange and validation of certificates, etc.

The second phase is the transformation itself, configured with the data the first phase negotiated.

I alluded to this two-phase architecture already, in footnote 4 on how the TLS package could possibly be restructured.

Note that while in this example the negotiation is done synchronously (-blocking 1) this could be done in an event-driven manner as well.

6. CLOSING THOUGHTS

We have demonstrated the usefulness of reflected and transformed channels in a variety of ways, and hopefully also managed to get the reader thinking about there these features of the core could be useful to them.

We have also shown that there a still problem zones, in the area of IO alone (See the issue with partial flushing of zlib), and in the intersection of IO and encodings (See the textwindow multi-byte conversion problems).

Another problem was getting used to TclOO [11]. The main problem with it was not its usability in general, that is quite fine. However, it currently does not have a package for processing options as nice as provided by snit [13] with its standard/custom accessors, validation types, delegation, etc. The example transformations suffer from that, as they either use positional arguments, i.e. do not use options at all, or only have very primitive processing without validating anything.

APPENDIX

A. REFERENCES

[1] Andreas Kupries, TIP 219, *Tcl Channel Reflection API.* `http://tip.tcl.tk/219` , Sep 2004

[2] Andreas Kupries, TIP 230, *Tcl Channel Transformation Reflection API.* `http://tip.tcl.tk/230` , Nov 2004

[3] Pascal Scheffers, TIP 234, *Add Support for Zlib Compression.* `http://tip.tcl.tk/234` , Dec 2004

[4] Pat Thoyts, TIP 317, *Extend binary Ensemble with Binary Encodings.* `http://tip.tcl.tk/317` , May 2008

[5] Bryan Oakley, *Read-Only Text Megawidget with TclOO.* `http://wiki.tcl.tk/22036`

[6] Simon Josefsson, RFC 4648, *The Base16, Base32, and Base64 Data Encodings.* `http://www.ietf.org/rfc/rfc4648.txt`

[7] Andreas Kupries, Donal Fellows, *refchan - Command handler API of reflected channels, version 1.* `http://www.tcl.tk/man/tcl8.5/TclCmd/refchan.htm`

[8] Andreas Kupries *Flushing doesn't work on stacked channel.* `https://sourceforge.net/support/tracker.php?aid=1785438`

[9] Andreas Kupries, *gIOT package.* `http://www.purl.org/net/akupries/soft/giot`

[10] Andreas Kupries, *Memchan package.* `http://memchan.sf.net`

[11] Donal Fellows, *Tcl Core Source Repository.* `http://tcl.sf.net`

[12] John Ousterhout and others, *Tcl Core Source Repository.* `http://tcl.sf.net`

[13] Will Duquette, *snit package.* `http://tcllib.sourceforge.net/doc/snit.html` `http://tcllib.sf.net`

[14] Multiple *Tcllib Bundle of Packages.* `http://tcllib.sf.net`

[15] Matt Newman, Jeff Hobbs, Dan Razell, *TLS package.* `http://tls.sf.net`

[16] Andreas Kupries, *Trf package.* `http://tcltrf.sf.net` `http://www.purl.org/net/akupries/soft/trf`

[17] Andreas Kupries, *TrfCrypt package.* `http://www.purl.org/net/akupries/soft/trfcrypt`

[18] Colin McCormack, *Wub Web Server.* `http://code.google.com/p/wub/`

[19] Andreas Kupries, *The "transform" command in Trf.* `http://www.purl.org/net/akupries/soft/trf/trf_transform.html`

[20] Andreas Kupries, *Seeking in Trf.* `http://www.purl.org/net/akupries/soft/trf/trf_seek.html`

[21] Colin McCormack, *Wub Chan Code.* `http://wub.googlecode.com/svn/trunk/Wub/Chan.tcl`

[22] Multiple, *OpenSSL.* `http://www.openssl.org`

[23] Steve Landers et al., *CryptKit.* `http://wiki.tcl.tk/13191`

[24] Peter Gutmann et al., *CryptLib.* `http://www.cs.auckland.ac.nz/~pgut001/cryptlib/`

[25] Mark Adler, *Adler-32 checksum,* `http://en.wikipedia.org/wiki/Adler-32`

[26] Whitfield Diffie, Martin Hellman *Diffie-Hellman key exchange.* `http://en.wikipedia.org/wiki/Diffie-Hellman_key_exchange`