Tcl Commands as Media in a Distributed Multimedia Toolkit*

Jonathan L. Herlocker
Joseph A. Konstan

Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

{herlocke,konstan}@cs.umn.edu

Abstract

This paper discusses the design and implementation of a command stream based on Tcl. A command stream is a series of arbitrary commands that can be tightly synchronized with other media in a distributed multimedia presentation. In TclStream, we represent an arbitrary command as a collection of fragments of Tcl code. The command stream medium supports the standard manipulation functions of multimedia environments: reverse, fast-forward, random access, and variable speed. The ability to specify arbitrary actions, combined with fine playback control, make TclStream an extremely flexible and powerful presentation medium.

Introduction

Conventional multimedia applications focus on audio, video, image, and text media. In this paper, we introduce a more flexible and more powerful medium--a stream of commands. This stream of commands--Tcl[1] commands in particular-can be used to implement animation, device control, user-interfaces, and many other less-conventional media types.

A command stream is a real-time medium composed of discrete commands. The commands may reside anywhere on the network, but they are executed locally on the machine where other medias are displayed. More generally, they may reside and be executed on any network-connected machines. Since a command stream will be only one media type in an integrated multimedia toolkit, we must be able to operate on it like any other media stream. Therefore a command stream must support the following operations:

- Playback in reverse.
- Playback at variable speeds (normal, fast-forward, fast-rewind).
- Random access to any point of the stream.
- Synchronization with discrete elements of other media streams, such as video frames and audio samples.

Tcl commands are particularly useful as the basis for the command stream. They are general, placing few restrictions on the actions we can perform. Tcl also provides a ready-to-use interpreter and integration into a networked environment [2]. Tcl commands can be used to access powerful libraries such as Tk[1], to generate user interfaces, and Expect[3], to operate interactive processes.

We implemented the Tcl command stream as a new medium for the Berkeley Continuous Media Toolkit (CMT) [4]. CMT provides support for several media types (including audio & video), network transmission of media, and a timeline based synchronization mechanism (shared logical clock). By adding the Tcl stream to CMT, we are able to integrate Tcl streams with other media in presentations.

This paper presents the Tcl command stream and its implementation. We begin with a brief introduction to the Continuous Media Toolkit, followed by a description and discussion of experiences with TclStream 1.0, the initial implementation of the command stream. We follow this with a discussion of one of the fundamental challenges in command streams - developing an authoring interface that allows real people to create them. We conclude the paper with our plans for TclStream 2.0, some observations about Tcl features we would like to have, and some general conclusions.

Brief Introduction to CMT

The Continuous Media Toolkit is a distributed real-time multimedia system, developed at the University of California Berkeley by the Plateau project. It is implemented in a combination of C and Tcl, with the API being in Tcl. CMT runs with a Tcl interpreter whose core has been augmented to better support real time scheduling, as well as the TclDP[2] and CMT command extensions.

*This work was supported in part by grants from the National Science Foundation (IRI-9410470) and the Graduate School of the University of Minnesota.

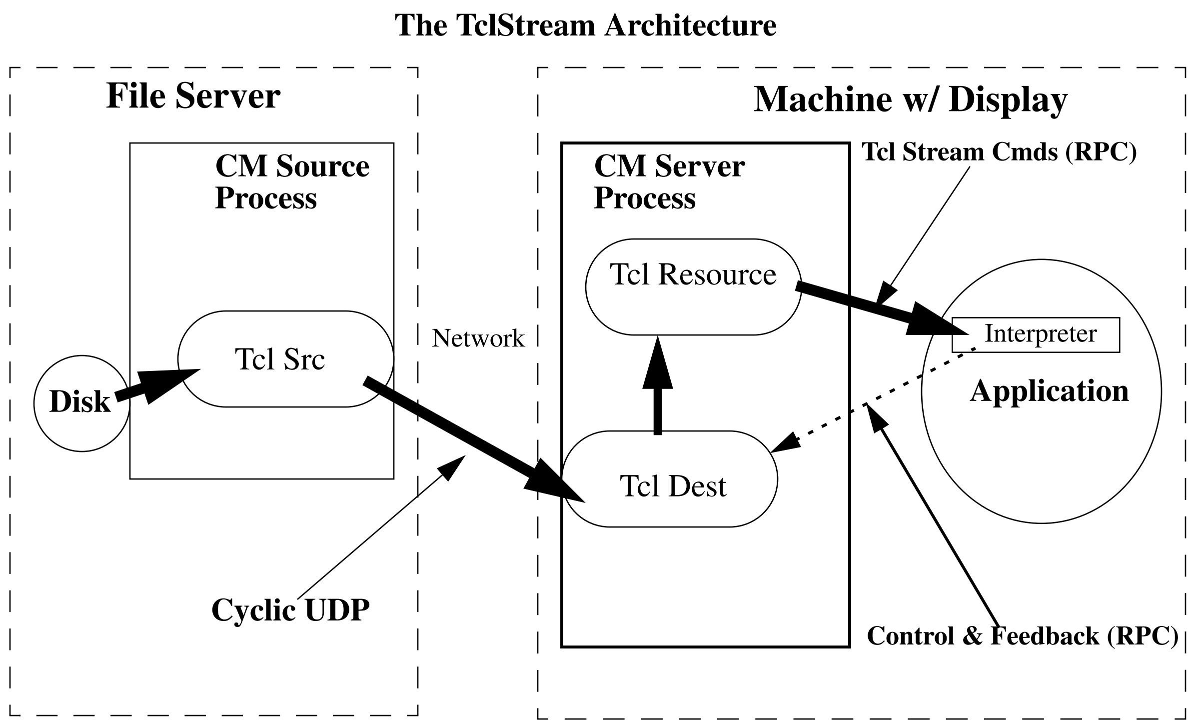

Figure 1: The TclStream consists of three key objects in the CMT environment: The TclSrc, the TclDest, and the TclResource

In the CMT architecture, there are three main processes: the *application*, the *server*, and the *source*. The source process runs on fileservers that hold the data (e.g. video, audio, or commands) and is responsible for reading the data from disk and sending it across the network. A server process runs on the machine where the media is to be displayed. The server process is responsible for receiving network data and either placing it on the display or passing it to the application. The application is responsible for building the initial user interface, and initializing the media streams from the source.

Synchronization in CMT is fine-grained and based on the timeline model. An exact mapping of media and tight time tolerances are achieved through a logical clock which is shared by all of the CMT processes. This shared logical clock is called the *Logical Time System,* commonly known as the *LTS*. The LTS is implemented as a simple distributed object, with two slots: speed and offset. The mapping of logical time to system time is LogicalTime = LTSspeed * (Systemclock - LTSOffset). A speed of 1 indicates a relation of one system second to one logical second. Fast forwards, fast rewinds or slow motion can be attained by setting the speed to a value other than 1 or 0. A value of 2.5 would be a fast forward, -2.5 a fast rewind, and 0.5 a slow motion forward. Jumps to a specific logical time can be achieved by setting the offset to (SystemClock - NewLogicalTime). The application's view of the LTS is two slots: value and speed. All offset calculations and system clock accesses are handled by the CM toolkit.

Media are synchronized in CMT by assigning them logical play times. Individual frames can be assigned their own times, or more commonly a stream will have a start time and a frame rate. Internally, each frame (e.g. video frame or audio segment) has a designated start time and duration that are used by both the source (to supply the server) and the server (to play the frames). The Tcl command stream uses the same LTS/timeline model and can therefore by closely synchronized with other media.

Implementation of TclStream 1.0

The overview of the design goals for the implementation was to keep Tcl Stream data files as similar as possible to a standard Tcl program. This would allow command stream authors to tap into the large collective experience and code of Tcl programmers as a basis for their work, and minimize the learning curve for those who already have experience programming in Tcl.

The Tcl Stream stream consists of three CMT objects, named the TclSrc, the TclDest, and the TclResource (see Figure 1). The TclSrc object resides in the source process (on the machine with the data) and is responsible

```
{0.1 {move_arm left_arm 100 100},\                         # Primary
     {},\                                                  # Rush-ahead
     {move_arm left_arm 100 300},\                         # Inverse
     {}                                                    # Rush Behind
}

{ 100.0a { button .b -text "Press here to skip to the Polka!"\
             -command {set jumped 1 ; setf .lts 300; destroy .b\},\
      { set jumped 0 },\
      { catch {destroy .b} ;  catch {unset jumped} },\
      {!!}
}
```

Figure 2: Two examples of chunks that might be used in a dance animation program.

for reading commands from disk and scheduling transmissions. The TclDest manages the reassembly of network buffers, network postprocessing, and local scheduling on the machine with the display. The TclResource object is the device driver of the Tcl stream. It arranges for the TclStream commands to be executed in the application's Tcl interpreter.

A basic atomic unit of a Tcl stream is termed a *chunk.* Each chunk corresponds to a piece of Tcl code to be executed. In its simplest implementation, a chunk will correspond to a single Tcl command, such as "`set x 1`" or "`gets $file`." However it is possible and usually useful to define chunks as more complicated actions or meta-actions to provide a greater level of abstraction. For example, in a dance animation presentation, a chunk might correspond to the action "`rotate-torso`," or "`high-kick`." There is no limit to the amount of code in a chunk. Since chunks are atomic relative to the Tcl command stream, chunks are "sized" at the granularity that provides the needed synchronization with other media. Large chunks can be problematic because the amount of time to execute those chunks may be much harder to predict, and their length may overlap the start times of later chunks.

Tcl Stream data files are either preprocessed or converted on the fly to CMT's native *ClipFile* format when the Tcl Stream is accessed from a CMT object. The ClipFile format contains time and chunk offset tables at the beginning of the file which allow the TclSrc to quickly locate a needed chunk based on a time offset.

Chunk Implementation

In version 1 of the TclStream, a chunk consists of a 5-tuple. The first element is the logical time in seconds at which that chunk should be executed. The logical time is a floating point number that can be expressed relative to the previous command or absolutely (relative to the origin of the logical time system). The ability to specify relative values of execution allows collections of chunks to be relocated as a whole, without recalculating the logical execution time of each chunk. This feature supports the development of libraries of chunk sequences, analogous to CMT "clips".

The remaining four elements are fragments of Tcl code. The first fragment is the *primary* Tcl code that will be executed if the Tcl stream reaches the chunk while moving forward (speed positive) in "normal" execution. The definition of "normal" execution becomes clear as we explain the *rush-ahead* code fragment. The rush-ahead code fragment is executed in one of two situations: 1) A chunk's start time has been skipped because the previous chunk took a long time to execute, or 2) The application may change the offset of the logical time system, so that the scheduled logical time for that chunk is skipped over.*

The last two elements are the equivalent fragments of Tcl code that will be executed if the speed of the logical time is negative (reverse). The first of these is termed the *inverse* code fragment, and the second is called the *rush-behind.*

Figure 2 provides two examples of possible chunks in a dance animation program.

The value of 0.1 seconds specifies that the first chunk is to be executed a tenth of a second after the previous chunk. The primary code will move the end of the arm to (100,100) on the screen. Notice that there is no rush-ahead. Since this chunk is in the middle of a sequence of arm movements, it is unnecessary to execute it if we are

*The rush-ahead code also addresses the case where the chunk arrived at the destination too late to play. This case, which is common for video on low bandwidth contention networks, is uncommon since chunk sizes are relatively small and the aggregate bandwidth needed for a common stream is similarly small.

rushing-ahead, because the following arm movement will simply cancel it out. The inverse code moves the arm back to it's original position. The rush-behind code is empty for the same reason as the rush-ahead.

In the second code fragment, the logical time is specified absolutely by appending an 'a' to the end of the logical time. This chunk will be scheduled exactly 100 seconds into the presentation. During normal execution moving forward, this chunk will pop up a button, asking if the user wishes to jump to a later point in the presentation. If the user pushes the button, then the Tcl stream will change the value of the logical time system itself (using the setf command which propagates the distributed object's slots properly), moving the entire presentation to a point 300 seconds in, where presumably the Polka demonstration starts. If the chunk is skipped for whatever reason moving forward, the button is not created, but a variable is set so that later chunks can react to the fact that the button was not displayed. The inverse and rush-behind chunks both remove the button if it exists and undo all changes to variables that might have occurred.

Implementation in CMT

The TclSrc schedules chunks to be sent to the server based on the value and speed of the current timeline. Whenever possible, the source sends chunks before they are actually needed. When it determines that a chunk needs to be sent, the source sends the two fragments of Tcl code that are appropriate to the direction of the logical time system (the primary and rush-ahead if speed is positive, inverse and rush-behind if negative), and the logical start time. During normal operation, after sending a chunk, the source will schedule a timer to wake itself up when the next chunk needs to be sent across the network. Any change in the offset or speed of the logical time line will immediately cause the TclSrc object to wake up, reassess it's situation based on the new offset and speed, and reschedule chunks accordingly.

The server (a combined effort of the TclDest and TclResource objects) receives the Tcl command fragments and schedules them to be executed at the correct start time in the application via TclDP RPC (remember that there is send-ahead). Whenever the logical time system is changed, the server dumps all scheduled code fragments that have not been played and waits for further information. If the logical time system has not been changed, and a chunk is skipped (due to a previous chunk which runs too long), the server will execute the rush-ahead code fragment for that chunk.

While the ease and portability of the Tk send command was very appealing for the communication between the TclDest and the application, we found that it could not keep up with the rate of commands that we wished to send, and ended up being a large bottleneck. Version 1 was quickly re-implemented with Tcl-DP's RPC communication.

In TclStream 1.0, all Tcl commands from the stream are executed at the global level of the application's Tcl interpreter. Since the application and the Tcl Stream are both executing commands in the same Tcl interpreter, this implementation is subject to a wide range of variable or function name conflicts between the two processes. We have been able to avoid conflicts by careful programming but future versions will address this in a more substantive way.

Through direct manipulation of the logical time system, the application or the Tcl Stream may change the value of logical time. Whenever the value of the logical time system is, the source determines all chunks that were skipped by the sudden change in time (backwards or forwards). For each of these chunks, the source sends only the rush-ahead (or rush-behind if speed is negative) code fragment. Only one fragment per chunk is sent because the rush-ahead has to be executed (there is no alternative). The source does not schedule these, but rather sends them all as quickly as possible, relying on the server to make sure that each chunk waits for it's predecessors to complete before being executed. The server will not drop explicit rush-aheads or rush-behinds and will play them as quickly as possible.

In order to construct a Tcl stream, a media author codes each 5-tuple chunk by hand. For each chunk, the author determines what the four different fragments of Tcl code will be. Given a primary code fragment, the programmer will have to come up with appropriate inverse, rush-ahead, and rush-behind.

Evaluation of TclStream 1.0

Several demonstration streams were written for the Tcl stream in order to exercise the system and assess its potential.

To our pleasure, it proved to be very easy to create streams of Tcl chunks that were synchronized with other media streams. In the matter of a day or two, we created a simple animated "karaoke" program in which an animated stick-figure danced in a window and visually sang in sync with the music. No more reference was needed to create the program than the Tcl/Tk reference card, though much trial and error was needed to perfect the synchronization.

The ability to interact with the user, and modify the behavior of the presentation based on that interaction was a great gain for very little programming effort. The Tcl stream used the Tk toolkit to place buttons and other controls in the application. These buttons would change the value and speed of the logical time system. The ability to jump to a completely different point in logical time provided an extremely simple and straightforward method to adapt to user input. Alternative execution lines can be placed in segments of the logical time line that would be automatically skipped over if the application didn't jump directly into them. Consider the dance animation program. At one point, we can have the stream create two Tk buttons, with the messages: "Press here to continue practicing the Polka," and "Press here to move on to the Waltz." Pressing the first button will cause no change in the timeline, and the dance program will continue with the Polka. Pressing the second button will change the offset of the logical time line so that it resumes execution just after Polka demo. Note that with the second button, the change in timeline will affect all other medias, so the correct music will be played.

The Tcl's stream's full access to all of CMT's API during runtime brings about the true power of the Tcl stream as a presentation interface. Since the Tcl Stream has access to the applications's CMT interpreter, it has as much control over the multimedia interface as the application. Upon determination of the application's environment or upon cue from the user, the Tcl stream can initiate completely new media streams, choosing audio and video clips that are appropriate to the occasion. The Tcl stream can also control the flow of logical time through the CMT API. The dance example code given above allowed the dance animation program to actually skip itself forward in logical time based on a event generated by the user. An entirely new dimension is gained when combined with the new CMT nameserver Now the Tcl stream will also be able to determine exactly what audio and video servers and services are available on the network at runtime by querying the nameserver. A Tcl stream author need not know where or what media clips will be available at the time the multimedia presentation is created. The dance animation program could begin by prompting the user to select a language for narration, then open up a stream customized to the chosen language. Note that with the new CMT nameserver, a the dance animation Tcl Stream would not have to be preprogrammed with the list of available languages, but could determine them at runtime.

A problem did appear when the value of the logical time system was changed by the application or the TclStream. Because the Tcl stream is based on a distributed system, the source has no way to be sure of what events have happened at the server which is normally located remotely (the source and the server can be located on the same machine, but that scenario is trivial). While a source may schedule a chunk to be executed at a specific time and send that chunk across the network, it has no way of knowing whether a chunk has been executed by the server, dropped by the network, or dropped by the server (the server drops all pending chunks when the logical time system is changed). In version 1, the source did it's best to guess what frames had been played and what frames hadn't been played, but occasionally changes in the value of the logical time system resulted in glitches where a chunk was not executed or a chunk being executed twice.

For audio and video, losing occasional frames was not an issue, especially right after a change in the value of the logical time line. But a missing line of Tcl code can have disastrous effects. Consider for example, the creation of a canvas widget, on which the Tcl stream application will draw. If the canvas is not created, then all of the later commands that draw to the canvas will fail without warning. Playing a piece of Tcl code twice can also have disastrous effects.

Any attempt to track the state of the stream must begin with the server, because it is the only process that actually knows whether a chunk in the stream has actually been executed or is executing (we are assuming that the RPC on the local machine is reliable). In its simplest case, if the source knows which was the last chunk played by the server, it can correctly handle any number of changes in the value of the logical time system. This is based on the assumption that the server will never truly "skip" a chunk, but will always play one of the four Tcl code fragments. To implement this would be simple, by having the server send a feedback packet to the source every time the value of the logical time system was changed. The amount of network traffic generated by a single feedback packet every time the LTS is changed will be insignificant compared to the network traffic required to re-send the media.

Another problem in the current implementation of the Tcl stream arises from large changes in the logical time. If a Tcl stream is large then a large change in the value of the logical time system will sometimes result in a considerable delay before the stream returns to normal execution. The delay results from the fact that the rush-ahead or rush-behind code for every chunk that was skipped must be executed. It is impossible to get around the delay resulting just from executing a large amount of Tcl code fragments (since the rush-aheads must be played), but it is hoped that rush-aheads and rush-behinds will be programmed with speed in mind. The major bottleneck of the rush-aheads however is the amount of time that it takes to send a large collection of code fragments from the source to the server across the network.

These two problems listed above led to a new design. In the new design, the chunk abstraction is extended into a object abstraction. Chunks will be given data and handler functions with enough control logic so that they can determine their own state and respond intelligently to events with minimal interaction with other Tcl stream objects. In this model, when a chunk is first encountered, it is migrated in whole to the application. Once a chunk is in the application, the source will control the execution of the Tcl stream by sending messages to the chunks. Chunks will determine how they should respond to an action based on their current state and the state of neighboring chunks.

The chunk-object abstraction described above will provide solutions for the two key problems. The source will no longer care about the exact state of the program. It will make it's best guess, and send a message to the first chunk it believes to have been skipped. The chunk which receives the message will make sure that it has not been played already and that all chunks preceding have been played. If the previous chunk has not played, the message is forwarded to it. If the current chunk has already played, the chunk forwards the message on to the next chunk in logical time order. Once a chunk plays a code fragment during a rush-ahead sequence, the message is passed on to the next chunk in order. The message contains the ending logical time of the skip, so the message will propagate until it reaches a chunk that was not skipped.

Since all chunks in the past are essentially cached, skipping backward can be resolved quickly with no network transmission at all. A Tcl stream can also use a extensive send-ahead to speed up rush aheads, caching future chunks in the application for access.

An important observation is that often when large amounts of time are skipped over, collections of chunks become irrelevant and can safely be ignored. Consider the dance animation interface again. Suppose that the stick figure is only used for a period of 5 minutes in the middle of the presentation. If the application skips from a point before those 5 minutes to a point after those 5 minutes, then there is probably no point in executing any of the chunks involved with the stick figure, either in primary or rush-ahead mode. To support this operation, chunks would have a "lifetime" attribute associated with them, so a chunk would simply defer if it's entire lifetime has been skipped. For example the lifetime of a chunk that drew on the screen would be from the moment it drew on the screen until the screen was erased or drawn over. Likewise in the other direction, a widget-destroy chunk would have a lifetime that extended backward to the point where the widget was first created. This should allow for considerable speed-ups when a large number of chunks are skipped and rush-ahead actions need to be played.

Another possible solution was inspired by an observation of the MPEG encoding standard for video [5]. The basic idea is to designate certain time spots as "key points" and to pre-compute the easiest way to jump from each key point to the next and the previous one. When making a large jump, we would execute the rush-ahead or rush-behind code to get to the nearest key point, then use the pre-computed jump code to jump along the key point chain to the key point closest to the destination, and finally execute the rush-ahead (or behind) code to reach the destination. This mechanism can be implemented (trivially) by combining all of the rush-ahead/rush-behind code into the key point jumps, but could be optimized by placing key points strategically at logical separations in a presentation.

The Authoring Interface

The real challenge to making a command stream a useful tool is to minimize the complexity of creating day-to-day command streams. While the Tcl Stream appears to offer flexibility and power, Tcl streams will not be a useful tool if they are daunting to write. Therefore one of the key focuses of continuing research will be into the evolution of the Tcl stream as a programming language and into aides for the authoring environment. These will help to combat some of the inherent complexities of using commands as media.

One of the inherent complexities is the need to determine the "inverse" of a piece of Tcl code. What action will the Tcl stream take when it reverses over a chunk? The chunk must undo any changes that it made moving forward that might affect execution in chunks earlier in the time line. In version 1, the media programmer had to determine for himself the inverse of a piece of Tcl code. This proved to be extremely time consuming, considering that for the "karaoke" stream, every chunk corresponded to a single line of Tcl code.

Also of concern is the determining the rush-ahead and rush-behind pieces of the chunk. When a sequence of chunks is skipped over, sometimes it makes sense only to execute the code of the last chunk (such as in an animation sequence), while at other times, each chunk must execute some code. Determining the exact action that is to be taken is not always easy for a Tcl stream author.

If we wish to explore the concept of the "lifetime" of a chunk, we must develop some way to determine that

```
object button { name args start length } {
    chunk A {
         time { $start }
         primary { button  $name $args ; set exists_$name 1}
         rush-ahead { !! }
         inverse { destroy $name; unset  exists_$name 1}
         rush-behind { !! }
         lifetime {this B }
    }
    chunk B {
         time { $start + $length }
         primary { destroy $name; unset  exists_$name 1}
         rush-ahead { !! }
         inverse { button  $name $args ; set exists_$name 1}
         lifetime { A this }
    }
}
```

Figure 3: An prototype for a chunk-object template.

lifetime. This is not a critical element because while we may suffer performance-wise from not specifying it, everything will work fine without the lifetimes specified. For some chunks, such as those that create and destroy widgets, lifetime is easy to determine. For others it may be more complicated or impossible.

It is our hope that we can build a library of commonly used chunks or chunk templates that will easily allow the Tcl stream author to build powerful interactive multimedia applications quickly, yet still allow him the complete flexibility to create his chunks from scratch. Figure 3 is an example of a possible template for a generic button widget. Creating a button object would result in the addition of two new chunks to the Tcl stream. The two chunks would initially be assigned logical times, but note that the lifetime is relative and allows the relocation of either without having to recalculate the lifetime. Note that '!!' means to repeat the previous command (c-shell notation).

As we work to simplify the interface for the media programmer, the Tcl stream data format will undoubtedly change and become specialized for the Tcl stream. But we will attempt to maintain this specialization as an abstraction of the more general Tcl code interface, allowing the media programmer the full flexibility of the Tcl interpreter. We don't want to define a entirely new language because we would then lose much of the benefit that we gained from starting with Tcl in the first place.

Some Tcl Issues

There are two general issues that need to be addressed at the Tcl level: security and extension management.

Security is a concern because commands are being sent from a remote, possibly untrusted server to be executed in your application's Tcl interpreter. This is not an issue specific to the Tcl stream application, and these security problems are being addressed by Ousterhout with the integration of Safe-Tcl into the Tcl core[ref]. In the Safe-Tcl model, untrusted Tcl commands would be executed in a crippled Tcl interpreter.

Extension management is a problem with the Tcl stream. A Tcl stream has no knowledge beforehand of what extensions the application's interpreter will have. Dynamic loading of extensions would allow a Tcl Stream to rely on extensions, without requiring the user to know what extended wish he needed to run before starting the Tcl Stream. If dynamic loading was not available, a standardized extension "registry" would help a Tcl Stream to quickly determine what extensions are installed and adapt to the situation.

Status and Future Plans

The next version of the Tcl stream is still in the design process. First of all, it will be based on a new version of CMT, version 3.0, with a cleaner API, more stable networking and buffering code, and support for a service nameserver. CMT 3.0 should increase the speed and efficiency of the Tcl stream as well as making it easier to combine with other media types.

Chunks in Tcl stream 2.0 will probably be more abstract and independent objects as described above. The exact data attributes and handler functions are not yet determined. Chunks will probably migrate to the application, in a separate Tcl interpreter to avoid name space conflicts. An associative array indexed by logical time will allow quick access to each chunk but each chunk will also have a pointer to the chunks following and previous to it, so that messages can be passed along the chain of chunks.

A graphical user interface for the development of Tcl streams is also in the planning stages. We hope it will make development using the chunk libraries and templates a simple task. We are also exploring several different applications of command streams in interface training and presentations.

We plan to release a beta version of TclStream in July 1995 and also hope to include TclStream as part of the CMT release planned for late summer 1995. For further information on the status and availability of TclStream papers and software, please see our World Wide Web page at "http://www.cs.umn.edu/research/GIMME/".

References

1. John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Massachusetts, 1994.
2. Brian C. Smith, Lawrence A. Rowe, Stephen C. Yen. Tcl Distributed Programming, *Proc. of the 1993 Tcl/TK Workshop*, Berkeley, CA, June 1993.
3. Don Libes. Expect: Scripts for Controlling Interactive Processes. *Computing Systems: the Journal of the USENIX Association.* Volume 4, Number 2. Spring 1991.
4. Lawrence A. Rowe and Brian C. Smith. A continuous media player. *Proceedings of the Third International Workshop on Network and Operating Systems Support for Digital Audio and Video*, p. x+416, 376-86.
5. Frank Gadegast. The MPEG-FAQ. Version 3.2, Aug 1994. http://www.cs.tu-berlin.de/~phade/mpeg/faq/mpegfa32.zip.