The following paper was originally published in the
Proceedings of the Sixth Annual Tcl/Tk Workshop
San Diego, California, September 14–18, 1998

Charity Telethon Supported by Tcl/Tk

Dave Griffin
Compaq Computer Corporation

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

Charity Telethon Supported by Tcl/Tk

Dave Griffin
Compaq Computer Corporation

Abstract

A set of Tcl/Tk Version 8 scripts supports a telethon that raises money for needy families. This paper describes the system and how it was constructed. Because the system uses nearly every major feature of the Tcl and Tk system, including the Tcl web server, this paper serves as an example that emphasizes the power of universal scripting in both the development and deployment of a distributed system.

Introduction

For the past 19 years in Maynard, Massachusetts, the local high school radio and television station, WAVM, stages a telethon to raise money for a local charity supporting needy families in the area. This annual community event culminates in a 40-hour, non-stop television telethon, which includes live auction segments taking place throughout the event. The telethon weekend is staffed by over 100 students (from 6th through 12th grade), parents, and other community members.

In 1992 a (then) new network and VAX/VMS computer system had been installed at the high school where the telethon originates. The author wrote the first telethon auction support program in some earlier scripting languages: DCL (Digital Command Language) and GNU awk. This program served the telethon well, but was an absolute nightmare to maintain. It was also dependent on hardware (VAX 4000 and VT420 "dumb" terminals) that has a limited future within the school system.

For the 1997 WAVM Telethon, I decided that it was time to free ourselves from the trusty VMS-based system and move to a system that took advantage of the PCs, servers, and network infrastructure that has grown over the past 5 years. WAVM's Web department was also growing rapidly and, for the first time, taking a major role in the telethon's presentation to the public. So, we were interested in integrating facets of the auction to an unknown Internet audience.

I was prepared to develop the whole system myself, but a fellow Tcl hacker wanted to learn more about Tk and offered to help. However, his time was limited. This situation was further complicated by his taking a new job at another company–which meant that our communication would be primarily through e-mail. We opted for a simple-but-flexible architecture that takes advantage of the computing infrastructure at the school, as shown in the following diagram:

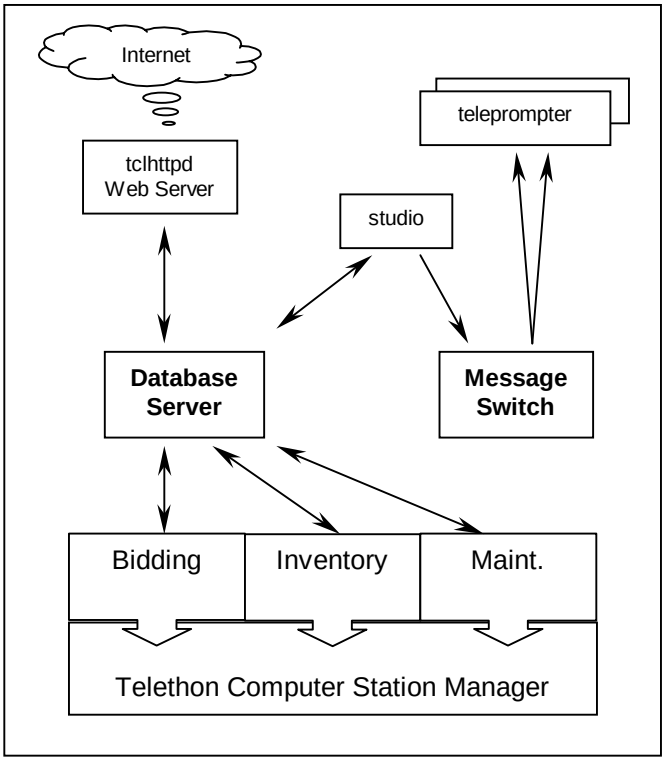

Fig. 1 – Architectural View

Even a "midnight" project needs priorities and we set them early on, as follows:

1. Support the auction. Provide programs to maintain the inventory of donated items and accept bids made over telephones during the telethon.

2. Keep the studios in touch with the auction. The first priority here was a way of letting the telethon hosts know the current top bids.

3. Make managing the telethon easier on the studio floor managers and on the adult support team. This includes reports, real-time monitors, etc.

4. Integrate the World-Wide Web with telethon activities--particularly the auction. (In parallel with all this work, we were preparing to webcast the full 40-hour telethon live via RealVideo.)

The Database Server

The first component to be constructed was the database server. I initially considered using the AltaVista Forum Toolkit [1], because it had a built-in database manager with journalling, and I was also very familiar with the interface. I did not consider other database packages (for example, MySQL) because I had no prior experience with them (and I didn't really have time to evaluate and select one). Plus it would require compilation and testing for my target platforms (Intel & Alpha). The AltaVista Forum software is based on Tcl 7.6–another minus. I opted to use a 100% Tcl approach, using the Tcl 8.0 release to construct a simple in-memory database with journalling for reliability [2].

The auction database consisted of three datasets: the items, the callers, and the bids. Each dataset had unique characteristics in terms of access and processing, but they all required common storage management and journalling/recovery features. (A windstorm in 1993 knocked out power during the middle of a telethon, so reliability features were historically mandated.)

Each dataset was given its own Tcl namespace within which to work. Database operations such as Add, Remove, etc., were implemented as Tcl procedures within the namespace. Because the total estimated size of the database was under a megabyte, the data for each dataset was stored in memory (as a Tcl array or list). This decision provided the server with a very high-performance base on which the reliability features were built.

It is interesting to note that while the Items and Callers datasets were implemented as one or more arrays that provided keyed access to the records, the Bids dataset was implemented as a simple Tcl list. Tcl 8 list operations are orders of magnitude more efficient than previous versions, and accessing elements of a list with 1000+ members consumes very little processing time. Based on historical data from the old system, we calculated an upper limit of 2,000 elements for any one list or array. So, we were well within what Tcl could efficiently manage using this simple approach to databases.

A single set of Tcl procedures implemented common operations such as Open, Close, Create Checkpoint, and Recover. These procedures were entered into each dataset's namespace (**eval**), where they merged with the other dataset-specific operations. This technique simulates an object-oriented inheritance operation, greatly reducing the amount of code needed for each dataset. The following figure shows all of these common methods as implemented in the database server:

```
Set CommonDBMethods {

  proc Checkpoint {} {
    variable dbInfo
    variable fileBase
    variable tlogFile
    close $tlogFile
    incr dbInfo(version)
    Save $fileBase.check$dbInfo(version)
    set_file $fileBase.tlog$dbInfo(version)
""
    set tlogFile [open $file-
Base.tlog$dbInfo(version) a]
    set_file $fileBase.info [array get
dbInfo]
  }

  proc Open { dbname } {
    variable fileBase
    variable tlogFile
    variable dbInfo
    variable inRecovery
    set inRecovery 0
    set fileBase $dbname
    array set dbInfo [get_file $dbname.info]
    Load $dbname.check$dbInfo(version)
    set tlogFile [open
$dbname.tlog$dbInfo(version) a+]
    if {[file size
$dbname.tlog$dbInfo(version)] > 0} {
        Recover
        Checkpoint
    }
  }

  proc Create { dbname } {
    variable fileBase
    set fileBase $dbname
    set_file $dbname.info "version 1"
    set_file $dbname.check1 ""
    set_file $dbname.tlog1 ""
    Open $dbname
  }
```

```
  proc Close {} {
    variable tlogFile
    Checkpoint
    close $tlogFile
  }

  proc TLog { tx } {
    variable tlogFile
    variable inRecovery
    if {$inRecovery} return
    puts $tlogFile $tx
    flush $tlogFile
  }

  proc Recover { } {
    variable fileBase
    variable tlogFile
    variable dbInfo
    variable inRecovery
    log recover "Recovering a database..."
    set inRecovery 1
    seek $tlogFile 0 start
    set tx ""
    while {[gets $tlogFile tLine] >= 0} {
      append tx $tLine
      if {![info complete $tx]} {
         append tx "\n"
         continue
      }
      log recover "/$tx/"
      eval $tx
      set tx ""
    }
    set inRecovery 0
  }

}
```

The common methods maintain a so-called `.info` file for each dataset, which coordinates the versioning of journal and checkpoint files.

The journalling and recovery feature exploits the "data represented as scripts" philosophy [3]. The lowest level database operations are implemented as Tcl procedures. The procedures that modify the database simply "record themselves" in the journal file as a Tcl script, as in the following example:

```
proc Items::SetRaw { key data } {
    variable items
    variable topKey
    set items($key) $data
    TLog [list SetRaw $key $data]
    if {$key > $topKey} {set topKey $key}
    return ""
}
```

When the database server shuts down, each data set will save a copy of itself to disk (checkpoint). This is as simple as `puts -nonewline $chan [array get items]`. When a database is opened, the latest checkpoint file is read into memory. Any transactions in the journal are then replayed by reading in the procedure calls and evaluating them, thus reapplying the missing transactions.

A fourth namespace, called "Tx", holds all the public transaction methods, again implemented as Tcl procedures. A Tcl **socket** connection manager accepts incoming TCP/IP connections. Upon receipt of a connection it then sets up a **fileevent** with the `Tx::tx` procedure called whenever the channel is readable. The database server implements a very simple protocol that allows the movement of arbitrary amounts of data. For both requests and responses, the data length is sent first (terminated by a linefeed), followed by the data.

The entire database server is implemented as an 850-line Tk script, which also provides a logging window for database events, a checkbox to enable debugging instrumentation, and a master switch to enable or disable bidding. The latter is used to assure that none of the bidding stations can accept bids until the auction officially starts. (The author could be seen running across the room in a panic at the start of the auction, because the first phone call had arrived but this "master switch" had not yet been thrown.)

The Auction

An auction is divided into 3 phases and there is specific software devoted to each phase.

Prior to the 40-hour telethon event, the focus is on entering new auction items into the database and tuning up the caller database; we developed some Tk scripts to handle these activities.

During the auction three to five "bidding stations" are active. These stations are manned by students who answer phone calls and process the bids. Special care was taken in the design of the bidding station's Tk interface to keep the flow of questions and responses easy to manage with a minimum of typing. This is particularly important when you have a phone in the one hand and the knowledge that many people are trying to call it at the same time. Care was also taken to allow backtracking and quick resetting in case the caller (or the student at the station) "got lost."

A number of Tk scripts support the auction by allowing the auction management team the ability to keep track of the bidding, providing statistics to the

telethon managers, and opening and closing bidding on individual or groups of auction items. The management scripts were fine-tuned during the telethon (one or two reporting scripts were actually written during lulls in the action).

After the telethon signs off the air, a team of adults and students use the various parts of the system to reconcile the paper and computer-based records to designate winners for all items.

All of these scripts were under the control of an "umbrella" script, which did a simple password check and gave access to a subset of the capabilities as needed. This script would dynamically load (**source**) in the particular program needed from a master network file share.

Supporting the Studios

Two television studios used during the telethon. We needed a way of relaying current bid information to the on-camera telethon hosts in a smooth and inobtrusive manner regardless of which studio they might be in. During the course of developing this capability it became apparent that a powerful and flexible system could easily be constructed to provide this critical information and potentially much more.

A "message switch" Tk script was designed to run during the entire telethon, much the same way the database server was always available. The message switch would accept messages via the standard **socket** and **fileevent** mechanisms. These messages could either listen for messages or leave a message. Every time a message was left, its payload would be the response for any outstanding `listen` messages. This was done by the `listen` transactions doing a **vwait** on the message number, which was incremented by the `leave` message. The following procs implement the core functions of the message switch:

```
proc connHandler { chan ipAddr port } {
    global activeChannels
    log conn "Starting channel $chan"
    fconfigure $chan -blocking no
    fileevent $chan readable "request $chan"
    set activeChannels($chan) "open"
}

proc request { chan } {
    global activeChannels
    fconfigure $chan -blocking no
    if {[gets $chan rqst] < 0} {
        fileevent $chan readable ""
        close $chan
        log conn "Closing channel $chan"
        unset activeChannels($chan)
        return
    }
    log debug "Tx $chan: $rqst"
    # Execute transaction (unsafely)
    if {[catch {eval $rqst} err]} {
        response $chan [list ERROR $err]
    }
    catch {flush $chan}
}

proc response { chan data } {
    if {[catch {puts $chan $data} err]} {
        log debug "Response Error: $chan"
    } else {
        log debug "Sent $chan data: $data"
    }
}


proc listen {} {
    upvar chan chan
    global messageNumber
    global lastMessage
    vwait messageNumber
    response $chan $lastMessage
}

proc leave { text } {
    upvar chan chan
    global messageNumber
    global lastMessage
    set lastMessage $text
    incr messageNumber
    response $chan \
        "Message $messageNumber sent"
}
```

The mechanism was very simple and did not attempt to provide any guarantees regarding message delivery, but it was more than sufficient for the task.

Two other scripts were written to communicate with the message switch. The first script acted as a transmitter, which could relay arbitrary text from the keyboard to the switch, or it could perform a simple database lookup (to the database server) and send the results to the message switch.

The final script was the Tk "teleprompter." Copies of this script were run in the two television studios with the computer monitors placed on top of the television "talent" monitors. Each teleprompter had a pull-down menu which allowed it to designate where it was located. This allowed it to screen out messages that were not intended for it (the transmitter scripts could optionally designate where it wanted the messages to appear). Using very large fonts, the teleprompter script displayed the current bids and any other text messages for the on-camera students. Once the PC was set up with this script, it was left to run unattended for the duration of the telethon.

The transmitter scripts were run on a few computers in the studio area and in the main control room. These were staffed by students allowing the on-camera students to call out the number of an auction item that was being discussed and have the latest bid information immediately available on the teleprompter.

When no messages were received by the teleprompter for approximately a minute, an **after** event would display a "countdown clock" of how many hours were left in the telethon.

Bringing the Auction to the Web

Although providing a Web interface was desirable, it was pretty much last on the priority list. Fortunately, I had the opportunity to be using the Tcl web server [4] in a separate project, so it appeared that integrating an auction inventory query facility would be relatively straightforward.

A "Telethon" package was constructed that provided the code for a small set of URL mappings. Like the database server, a separate namespace was established to hold the Telethon package's context (e.g., a shared TCP/IP connection to the database server). It's primary job was to issue database transactions and to format the results in HTML.

The ability to look up individual items, or get reports on multiple items turned out to be convenient for everyone. Because Tcl/Tk's print support is a bit weak, we made our reports available via the web server (or generated them in HTML), and let the browsers do the formatting and printing for us.

An interface to the message switch was also created to allow Internet viewers to interact with the students. However, there was insufficient time to fully develop and test the capability. So, we dropped back to using e-mail for feedback.

Results: Trial By Fire

The telethon would eventually utilize at least thirteen computers deployed in various rooms and studios. Given our limited resources and fixed schedule there was not much time for testing the system in the "all up" configuration. Critical components like the database server were unit tested by test harness scripts, to flush out any bugs and to validate that the performance was acceptable.

Much to our delight, the software performed nearly flawlessly. A few glitches in the bidding station program were corrected on the fly during a lull in the auction. The database and message servers ran non-stop and easily handled the load placed upon them. Nearly 500 items were entered into the auction inventory, and we accepted about 1,500 bids.

During the 40-hour telethon, the database server processed 25,213 transactions. The message switch routed 2,848 transactions to the various stations in the studios.

Our custom web server was an unexpected bright spot. We really were not sure if the idea of using the Internet for a local event such as ours would attract anyone. During the telethon, just over 7,800 web transactions were processed, with a number of small updates to the program applied on-the-fly and no interruptions in service. Feedback received during the auction was universally positive for this service, and we reached people far outside the normal broadcast range. For the short amount of time it took to make it all work, this was well worth the effort.

Construction Notes

The true "script once, execute anywhere" was critical to the timely development of the software. This is highlighted by two observations:

1. The author has no information about the platform on which his partner developed and tested the bidding program. The scripts arrived as attachments to e-mail messages, which were saved to the testing area and used immediately.

2. While the systems used for actual telethon operations varied between Intel and Alpha, and between Windows NT and Windows 95, a major portion of the development of this distributed system was done while in the passenger seat of the family car on a standalone Win95 laptop. Also, portions of the custom web server were tested on a Digital UNIX system.

We never had to think twice about whether or not the scripts would behave properly when they ran on the different platforms. The only difference that had to be accounted for was font-size specifications required by different screen resolutions. This problem was taken care of by implementing a pull-down

menu of base font-size preferences on the programs where it mattered.

Because all of the auction stations accessed the scripts via a network file share, on-the-fly bug fixes and updates were immediately available to the simply by closing and re-opening the appropriate window and letting the umbrella script re-source the application. At least one important bug fix was "distributed" to the stations in between phone calls.

Coding started almost exactly one month before the telethon. It took roughly 40-60 hours of work to design, code, and test the system up to the time the telethon went on the air. Another 2-3 hours of work was invested during and after the telethon to handle unanticipated issues.

Reuse of existing scripts and code, such as the Tcl web server, made it possible to add new features quickly as they became necessary. The decision to build the database server "from scratch" rather than to use a more traditional database server did not adversely affect the project. An analysis of the database server shows that roughly 325 lines of code actually implement the core database functions; the remaining code would have been necessary even if an SQL database was used instead.

Summary

Our telethon raised over $36,700 for local needy families, with the auction bringing in nearly $9,600, both amounts were new all-time records.

Tcl/Tk was instrumental in providing a modern, robust system on which future telethons will build. Tcl 8.0's performance, customizable web server, and platform independence were key assets in allowing an organization without significant resources to construct a distributed auction manager.

With the success of the telethon behind us, we are exploring how to apply this technology to day-to-day station operations. Our first task was to create a way to switch our webcasting services on and off via the web server. For upcoming telethons we will explore increasing the interaction between the web audience and the studios (messages, challenges, etc.), perhaps with limited auctioning occurring on the web itself (without SSL we need to find a way of easily authenticating bidders). One thing is for certain, the WAVM Telethon will continue to be "Tcl Powered".

References

[1] David Griffin. "Tcl in AltaVista Forum", 5th Annual Tcl/Tk Workshop Proceedings, 1997

[2] Andrew Birrell, Michael Jones, and Edward Wobber. "A Simple and Efficient Implementation for Small Databases", Digital SRC Research Report 24, 1988

[3] John Ousterhout. "Tcl and the Tk Toolkit", Section 28.4., Addison-Wesley 1994

[4] Brent Welch and Steven Uhler. "Web Enabling Applications", 5th Annual Tcl/Tk Workshop Proceedings, 1997