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

An Extensible Remote Graphical Interface for an ATM Network Simulator

Michael D. Santos, P. M. Melliar-Smith, and L. E. Moser
University of California, Santa Barbara

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/

An Extensible Remote Graphical Interface for an ATM Network Simulator *

Michael D. Santos, P. M. Melliar-Smith, L. E. Moser
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106
michael@alpha.ece.ucsb.edu, pmms@ece.ucsb.edu, moser@ece.ucsb.edu

Abstract

The University of California, Santa Barbara, is developing a 40 gigabit per second ATM switch as part of its Thunder and Lightning network project. To use the high bandwidth efficiently, new network protocols are being developed and simulated on the Thunder and Lightning network protocol simulator. Due to the extreme memory and computational requirements of the simulator, the display, unlike most Tcl/Tk interfaces, must be implemented as a distinct process capable of running on a remote machine. This paper discusses some of the issues that arise with such a physical separation of application and interface, and describes the implementation of the simulator's display application, with an emphasis on the use of the Tcl language. One module of the GUI for the simulator is discussed in detail, demonstrating the use of XDR (external data representation) with Tcl sockets to provide for cross-platform binary data exchange between the simulator and its display application. We also discuss our experience in building the simulator GUI and propose ways in which XDR might be incorporated into Tcl. We discuss some shortcomings of the canvas widget and describe mechanisms to overcome them.

1 Introduction

Advances in fiber-optic and VLSI technology have led to the emergence of very high-speed networks based on Asynchronous Transfer Mode (ATM) [1]. The Electrical and Computer Engineering Department of the University of California, Santa Barbara, in conjunction with Rockwell International Science Center, is currently building a 40 gigabit per second ATM switch as part of its Thunder and Lightning network project [2].

With the rapid increase in network bandwidth come new challenges for protocol development. Consider, for example, the situation in Figure 1 in which a user wishes to transmit data from San Diego, California to Boston, Massachusetts. In a typical ATM network, the user makes a request to reserve bandwidth from San Diego to Boston. The user's request traverses the network from San Diego to Boston, reserving bandwidth, and then returns to San Diego, informing the user of the capacity reserved, at which point the user can begin transmission. Due to the finite speed of light, this process takes approximately 40 milliseconds during which time 200 megabytes of data could have been transmitted into the Thunder and Lightning network if adequate capacity had been available.

Similarly, an ATM cell which is lost due to buffer overflow imposes a minimum delay of 40 ms before the retransmitted cell can be received at the destination. During this time, the sender could have transmitted another 200 megabytes of data into the network. To insert the retransmitted cell into the data stream properly (ATM guarantees in-order delivery of cells), the receiver must buffer this 200 megabytes of data while it waits for retransmission of the one lost cell. The sender faces similar buffering requirements as it must be able to retransmit a cell it sent at least 40 ms in the past (or 200 megabytes prior in the data stream). This is not a problem if the data source is a stable storage device, but may become a problem if, for example, the source of the data is a real-time measuring instrument.

As part of the Thunder and Lightning project, we are developing protocols [7, 8, 13, 14] that allow a sender to begin transmission without the lengthy reservation delay and yet provide lossless transmission. A network simulator [6] developed specifically

*This work was supported by DARPA Contract No. DABT63-93-C-0039, a National Science Foundation Graduate Research Fellowship, and a University of California Dissertation Year Fellowship.

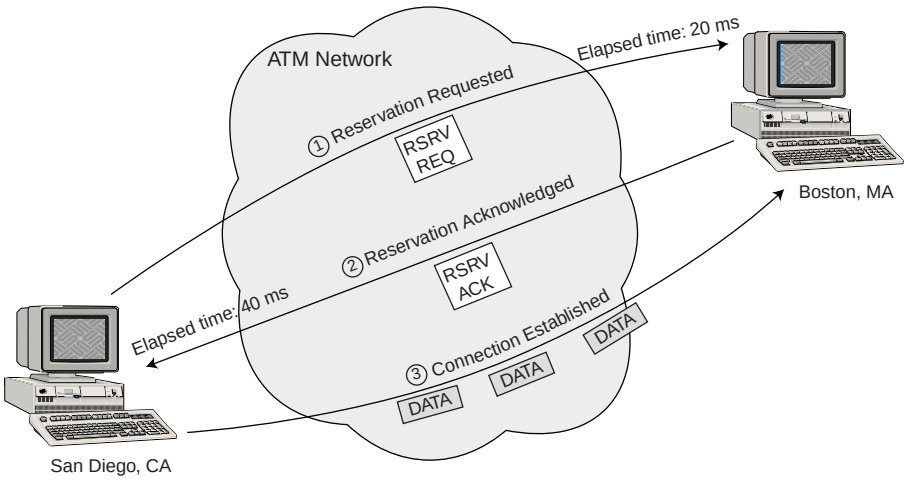

Figure 1: Connection establishment in a typical ATM network.

for Thunder and Lightning provides a testbed for the rapid prototyping, development, debugging and demonstration of these protocols.

To aid the protocol developer further, we have developed a graphical interface application which provides a view into the simulated environment. The display allows the user to manipulate the simulation in progress by inserting new events and altering existing events. Detailed protocol state information is also made available for debugging the protocol. This paper describes the design and development of this display program, emphasizing the use of Tcl/Tk. In addition to describing how Tcl/Tk has been an invaluable tool in rapidly developing this interface, we outline the mechanisms we developed to overcome the absence of certain features in the language.

2 Design Issues for the Display Application

The Thunder and Lightning protocol simulator has stringent memory requirements as it must track every ATM cell in the simulated network to ensure that no cells are lost. When simulating a relatively simple 4×2 mesh of ATM switches and the data sources connected to it, the network simulator must keep track of over 2 million ATM cells, or 100 MB of data. While novel data representation techniques [6] allow us to reduce the memory requirements of the simulator significantly, it is clear that, while simulating more complex networks, the simulator may consume all of the available memory in its host machine. In addition, the simulation process is entirely compute-bound. Thus, any extra computation the host machine must perform negatively impacts the speed of the simulation.

Therefore, in designing the simulator display, we had the following goals:

- The display should reduce the memory available to the simulator as little as possible,
- The computational impact on the simulator should be negligible when the display is not in use,
- The display should be as flexible as possible so as to allow features to be added or modified as the protocol specification is changed, and
- The display should be reusable as a monitor for the real Thunder and Lightning switch.

We minimized the memory requirement of display code within the simulator by making the user interface a separate application. Whereas a typical Tcl/Tk application integrates the user interface into the application, we split the application and the user interface into two separate processes using TCP/IP sockets to provide the interprocess communication. As a result, the display process can run on a separate machine. This not only reduces the size of the simulation code but also implies that the window system (*e.g.*, X Windows) does not have to run on the machine performing the simulation, thereby increasing the amount of memory available for use in the simulation.

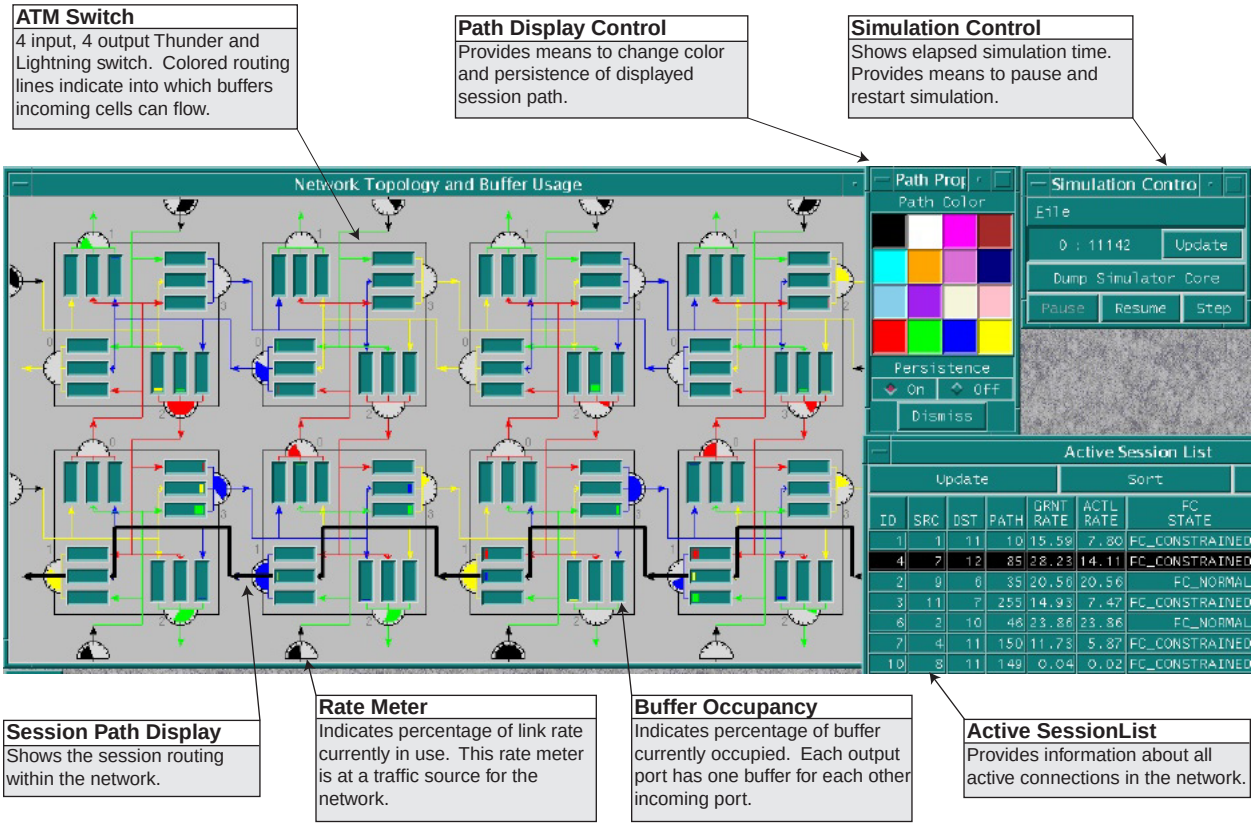

Figure 2: Sample simulation display.

While executing the user interface on a separate machine helps reduce the computational load on the machine running the simulation, employing a client/server relationship between the display and the simulator further reduces the computational cost. The simulation does not constantly send status information to the display over the communication channel. Instead, the display requests (only) the information needed to satisfy the user demands. The simulator responds to display requests with the appropriate information, and the display process then manipulates the received data into a form suitable for display. In this way, the simulator is interrupted from normal processing only when a specific request is received; if the display is idle or not running, no processing time is wasted.

Finally, display flexibility is provided through the use of the Tcl/Tk scripting language [4]. As described in Section 3 below, almost all of the graphics-related code is written as compact Tcl/Tk scripts, which allow for rapid coding of new window types. The display uses only a small amount of C code to initiate requests and to convert the binary response data into string lists that the scripts can use.

3 Implementation Details

To provide as much flexibility as possible, the display application is implemented as multiple nearly-independent modules. Each module is responsible for the display of a particular kind of data and is implemented using both a C file and a Tcl/Tk script. Transmission of requests and reception of responses related to that display type are handled in the C file. The C code converts the simulator response into a form useful by Tcl and then invokes a Tcl procedure to perform the display action. The Tcl file implements the user interface for the data display maintained by the module.

One such display module is used to illustrate the path a simulated connection takes through the network, as shown in Figure 2. These path displays are either transient, in which case they are automatically removed from the display after a short delay,

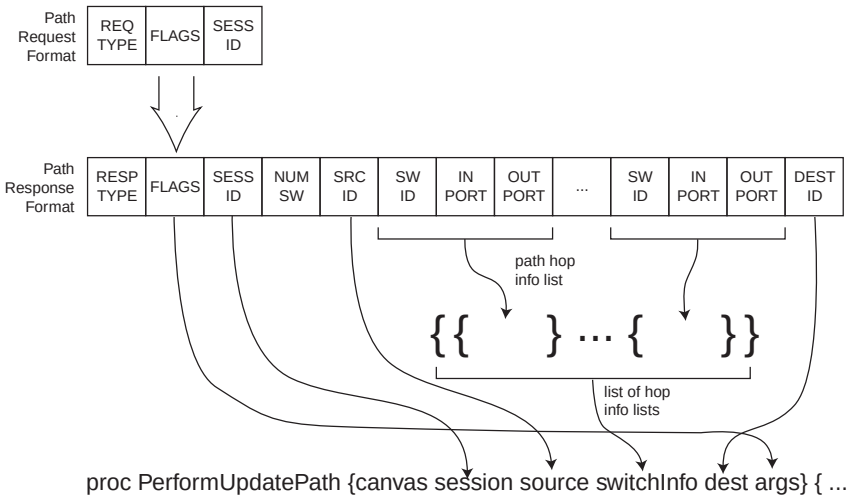

Figure 3: Mapping `D_PATH` response data into function call arguments.

or persistent. Because multiple routes may be displayed simultaneously, the path module automatically chooses a different color for subsequent path displays. A small control panel provides a means to change the persistence and color of displayed routes.

In the following sections, we describe both the Tcl and C code for the path module to provide a concrete example of the ease with which modules can be created.

3.1 Simulator-Display Interface

When the user wishes to display a particular piece of information, the display sends a request to the simulator. Each request consists of a single integer request-type field followed by a request-type-specific number of arguments. Figure 3 illustrates the format of the request message used to obtain session routing information from the simulator. The request-type field holds the constant integer value, `D_PATH`, assigned to path requests. Routing information requests have two arguments. The flags field argument indicates whether the path display should be persistent or transient, while the session ID field indicates the simulated connection that should be displayed.

To aid in the concurrent development of the simulator and the display, the display application is stateless with respect to communication. When the display makes a request on behalf of the user, it does not maintain any record of the request. This enables the simulator to ignore unrecognized requests, while not causing the display to hang awaiting a response that will never be received. Similarly, if the display cannot handle a response from the simulator, the response may be discarded without affecting the simulator.

Stateless communication also allows the display application to be used virtually unchanged with the real Thunder and Lightning switch. The Thunder and Lightning switch runs a small daemon process which implements the subset of simulator responses that apply within the context of a real switch. For example, the daemon responds to buffer occupancy requests, but silently ignores requests to edit or create simulation events. The interface presented to the user is the same whether the display is connected to a simulation process or a real switch.

The fact that communication is stateless, however, requires all responses from the simulator to contain sufficient information about the original request to enable the display to process the response appropriately. Figure 3 shows the format of a path response message from the simulator. Because no request state is maintained, the response includes both the session ID and the flags fields transmitted in the original request message. In fact, the simulator completely ignores the flags field when processing the request; the persistence information for the path display must be transmitted to the simulator in the request message for the sole purpose of returning it in the response message. In addition to information from the original request, the response also contains the source and destination IDs of the path, a set of triples containing information about each hop along the path, and a field indicating the length of the set. Each triple identifies the switch ID and the ports on

which the session enters and exits that switch. As described later in Sections 3.2 and 3.3, the display converts these integers to Tcl lists in C code and then to display-specific alphanumeric canvas tags in order to choose the correct canvas items to highlight on the topology display.

3.2 Implementation of the Path Module in C

The purpose of a module's C code is to handle communication with the simulator over the socket. Unlike other remote display applications, such as [9], which exchange text messages over a socket, the simulator and display exchange binary data. We use XDR (external data representation) [10] routines to convert data to and from network form. This allows the display and simulator to be run on architectures with different binary representations. The initialization code, not shown here, installs a channel handler for the socket that reads raw data from the socket into an XDR buffer. The channel handler then passes the XDR buffer to the appropriate module's handler based on the response type. (Recall that simulator responses must be self-identifying because the display is stateless.)

The path module initialization code is shown in Figure 4. The initialization code simply registers a new command with the Tcl interpreter to allow a Tcl script to request a path display.

Figure 5 illustrates the code for the request procedure. We create an XDR buffer with a call to `xdrmem_create`, and then encode each element of the request using XDR functions. In this case, the request consists of the request type, `D_PATH`, a flag indicating the path's persistence, and the session ID to be displayed. When the request has been assembled, we send the entire XDR buffer to the simulator over the Tcl socket (`PORT.socket`) between them.

When a response is received from the simulator, the channel handler servicing the socket reads the data from the network into an XDR buffer. Based on the response type, the appropriate module's handler function will be invoked. The code in Figure 6 will be invoked for all responses of type `D_PATH`. The path module expects responses to be in the format indicated in Figure 3. `HandlePath` reads all of the data from the XDR buffer and then parses it into Tcl strings by printing into temporary character arrays. Lists are assembled as shown in Figure 3 before invoking the Tcl procedure to display the path information.

3.3 Implementation of the Path Module in Tcl

As is the case with all modules in the simulator display, all of the graphical work is done in Tcl scripts. This provides for quick development, as no recompilation is necessary when making changes to a module under development. In most cases, we can even test changes in a module on the fly, without restarting the display process, simply by re-sourcing the Tcl file.

The Tcl code for the `PerformUpdatePath` procedure, which displays a route on the network topology display, is shown in Figure 7. It is called by the `HandlePath` C function described in Section 3.2. The display's path module relies on the fact that the network topology display module is implemented in a Tk canvas window and assigns tags to all of the routing arrows on the display as follows. Each line segment associated with switch x has a tag of the form "`switch`x." In addition, each segment on the path leading from input port y to output port z at a switch has a tag of the form "`in`y`out`z."

As indicated in Figure 3, the `switchInfo` argument to `PerformUpdatePath` contains a list of ⟨switch ID, input port, output port⟩ triples. To highlight a path, we find all canvas items (line segments) tagged with both the switch tag for the appropriate switch ID and the in/out tag for the given input/output ports. Note that Tcl/Tk does not provide a direct means of locating canvas items based on two or more specified tags. We could have modified the network topology module to add an additional tag of the form "`switch`x`in`y`out`z" so a search could be done for a single tag. Instead, we chose to implement the `canvasMultiMatch` procedure to provide this missing functionality so we wouldn't need to modify the network topology module each time a search on a different group of tags was required. We then duplicate each item found and copy all attributes of the original line segment to the newly created segment, which by default is now on top of the canvas stacking order. We set the new segments to a color from a predefined color list, increase the line width, and tag the new elements with a unique tag so they can be easily distinguished later. In addition, we set a mouse binding to allow the user to change the color or persistence of the path. Finally, if the path is not persistent, we create a timer callback to delete the path after a predefined interval.

Although it is not shown here, the path module also contains a few other Tcl procedures. `PathControlWindow` implements a palette-like con-

```
int PathInit(Tcl_Interp *interp) {
  Tcl_CreateCommand(interp, "RequestUpdatePath", RequestUpdatePath,
    (ClientData) NULL, (Tcl_CmdDeleteProc *)NULL);
  return TCL_OK;
}
```

Figure 4: Path module initialization procedure.

```
int RequestUpdatePath(ClientData clientData, Tcl_Interp *interp,
                      int argc, char *argv[]) {
  char buffer[BUF_SIZE];
  XDR xdrs, *xdrsend = &xdrs;
  int temp, flags = 0;
  if (argc < 2 || argc > 3) {
    /* error handling omitted */
  }
  xdrmem_create(xdrsend, (void *)buffer, BUF_SIZE, XDR_ENCODE);
  temp = (int)D_PATH;
  xdr_int(xdrsend, &temp);
  if (argc > 2 && strcmp(argv[2], "persistent") == 0)
    flags |= (int)PERSISTENT;
  xdr_int(xdrsend, &flags);
  temp = atoi(argv[1]);
  xdr_int(xdrsend, &temp);
  Tcl_Write(PORT.socket, (char *)buffer, xdr_getpos(xdrsend));
  xdr_destroy(xdrsend);
  return TCL_OK;
}
```

Figure 5: C code to request path information from simulator.

trol panel interface allowing the user to change the color or persistence properties of a displayed route. Additional procedures are used to notify other modules of the posting and removal of path displays, should they wish to annotate their displays accordingly. For example, a session route displayed in black causes the corresponding entry in the session list window to be highlighted in black, as shown in Figure 2.

4 Experiences

Tcl/Tk has saved us countless hours in the development of the graphical display for the Thunder and Lightning protocol simulator. In fact, without Tcl/Tk, it is likely that the display application would never have come into existence, because our primary research involves developing protocols for high-speed networks, not designing graphical interfaces! Only the ease and rapidity of design provided by Tcl/Tk have afforded us the opportunity to develop the display interface. Nonetheless, we have uncovered several areas in the language that we believe could use further development. We present these shortcomings in this section, along with the techniques we used to circumvent them.

4.1 Sockets between Heterogeneous Platforms

When this work began, sockets were not an official part of Tcl 7.3 and Tk 3.6. It is partly for this reason that the specification for interprocess communication between simulator and display uses binary data and not strings, which would be more natural for Tcl. Nonetheless, we retrofitted both the simulator and display to use Tcl sockets when they became available in Tk 4.1. Taking advantage of the cross-platform advances in Tcl/Tk 8.0 has allowed us to port both the simulator and the display to various platforms on which Tcl is supported. Both applications should be capable of running on any Tcl-supported platform and have been tested successfully on Solaris 2.x, MacOS 8.x, and MkLinux DR3 on a PowerPC.

Despite the uniform, cross-platform access to sockets that Tcl provides, problems arise when the simulator is run on one architecture and the display is run on another. When both applications are run on the same architecture, each safely interprets incoming binary data in its native format. However, when one application is run on a little endian architecture (*e.g.*, Windows on x86 processors) and the

```
int HandlePath(XDR *xdrrecv) {
  int numSwitches, sessionID, source, dest, *switchInfo;
  int sw, item, result, i;
  int flags;
  char buffer[10];

  /* read in path flags, session ID, # of switches, source ID */
  if (!(result = xdr_int(xdrrecv, &flags)) ||
      !(result = xdr_int(xdrrecv, &sessionID)) ||
      !(result = xdr_int(xdrrecv, &numSwitches)) ||
      !(result = xdr_int(xdrrecv, &source))) {
    /* error handling omitted */
  }

  /* read in all the switch information (⟨ID, in port, out port⟩ triples) */
  switchInfo = (int *)malloc(3 * numSwitches * sizeof(int));
  if (!switchInfo) {
    /* error handling omitted */
  }
  for (i = 0; i < 3 * numSwitches; i++)
    if (!(result = xdr_int(xdrrecv, &switchInfo[i]))) {
      /* error handling omitted */
    }
  /* read in destination ID */
  if (!(result = xdr_int(xdrrecv, &dest))) {
    /* error handling omitted */
  }
  /* now we need to parse all of this information into:
   *      session, source, list of switch info lists, destination
   */
  sprintf(buffer, "%d", sessionID);
  Tcl_SetVar(gInterp, "pathTmpSession", buffer, 0);
  sprintf(buffer, "%d", source);
  Tcl_SetVar(gInterp, "pathTmpSource", buffer, 0);
  sprintf(buffer, "%d", dest);
  Tcl_SetVar(gInterp, "pathTmpDest", buffer, 0);
  Tcl_UnsetVar(gInterp, "pathTmpSwList", 0);
  for (sw = 0; sw < numSwitches; sw++) {
    Tcl_UnsetVar(gInterp, "pathTmpList", 0);
    for (item = 0; item < 3; item++) {
      /* build up switch sublist */
      sprintf(buffer, "%d", switchInfo[(sw * 3) + item]);
      Tcl_SetVar(gInterp, "pathTmpList", buffer,
                 TCL_APPEND_VALUE | TCL_LIST_ELEMENT);
    }
    /* append sublist to switch list */
    Tcl_SetVar(gInterp, "pathTmpSwList",
               Tcl_GetVar(gInterp, "pathTmpList", 0),
               TCL_APPEND_VALUE | TCL_LIST_ELEMENT);
  }
  /* Finally, invoke the update command */
  Tcl_VarEval(gInterp, "PerformUpdatePath ", " .switch.c ",
              "$pathTmpSession $pathTmpSource $pathTmpSwList $pathTmpDest ",
              (flags & PERSISTENT) ? "persistent" : "",
              (char *)NULL);
  return 1;
}
```

Figure 6: HandlePath: C code to process a Path response from the simulator.

```
proc PerformUpdatePath {canvas session source switchInfo dest args} {
  global pathTmpNum pathColors

  foreach switch $switchInfo {
    set itemList [canvasMultiMatch $canvas [list "switch[lindex $switch 0]" \
        [format "in%dout%d" [lindex $switch 1] [lindex $switch 2]]]]
    foreach item $itemList {
      #create a copy of the original, alter it, and tag it
      set newItem [eval .switch.c create [.switch.c type $item] [.switch.c coords $item]]
      foreach attribute [.switch.c itemconfigure $item] {
        .switch.c itemconfigure $newItem [lindex [lindex $attribute 0] 0] \
            [.switch.c itemcget $item [lindex [lindex $attribute 0] 0]]
      }
      set color [lindex $pathColors [expr $pathTmpNum % \
          [llength $pathColors]]]
      .switch.c itemconfigure $newItem -fill $color -width 4 -tag pathTmp$session
    }
  }
  PathPostColor $session $color
  .switch.c bind pathTmp$session <Button-3> \
      "PathControlWindow pathTmp$session $session"
  # if not persistent, schedule delete timer and store timer ID as a tag
  # so it can be cancelled later if need be
  if {[lsearch -exact $args persistent] == -1} {
    # We use eval and escape the {} so the variables will be
    # expanded now and not when the timer expires
    set timerID [eval after 5000 "\{
      catch \{$canvas delete pathTmp$session\}
      catch \{destroy .pathTmp$session\}
      catch \{eval upvar #0 pathPersistpathTmp$session pathPersist\}
      catch \{unset pathPersist\}
      PathUnpostColor $session
      \}"]
    .switch.c addtag timer$timerID withtag pathTmp$session
  }
  incr pathTmpNum
}
```

Figure 7: Tcl code to post path trace on network topology canvas.

other is run on a big endian architecture (*e.g.*, Solaris on SPARC processors), this is no longer the case. When one interprets an integer sent by the other, the receiver will "see" the integer as a different value than the sender intended because the ordering of the bytes comprising the four-byte integer are stored (and therefore transmitted) in a different order. A similar situation occurs with the Thunder and Lightning switch processor, which uses a non-standard floating point format for efficiency.

The binary socket option combined with the binary command in Tcl 8.0 provides only a partial solution to this problem. Using the binary command, integer data can be converted to a known byte ordering, and the simulator and display could each convert from the agreed upon order to their platform's native order. However, the binary command cannot be used in this way for floating point data, which is used by other display modules. In addition, the binary command would be unavailable for use within the Thunder and Lightning switch processor, as the daemon does not have access to a Tcl interpreter.

Instead, we solved this binary data representation problem by using the XDR (external data representation) standard, which imposes a common network byte ordering (and size) for all simple data types. XDR provides functions of the form `xdr_`*type* for encoding or decoding data of type *type*. The preceding section presented code examples using the function `xdr_int` to read integers from an XDR buffer. Because all data exchanged between simulator and display are converted into network form by XDR routines, we are guaranteed that there will be no "misunderstanding" between the two.

Unfortunately, XDR is not available on every Tcl-supported platform. Most UNIX variants support

```
enum xdr_direction {XDR_DECODE, XDR_ENCODE};
typedef struct xdr_struct {
    enum xdr_direction direction;
    void *buffer;
    void *next;
    int bufLen;
} XDR;

#define easyxdr(typename) \
int \
xdr_##typename(XDR *xdr, typename *value) { \
    if (!xdr->buffer) \
        return 0; \
    if ((int)xdr->next + sizeof (typename) > (int)xdr->buffer + xdr->bufLen) \
        return 0; \
    switch (xdr->direction) { \
    case XDR_ENCODE: \
        memcpy(xdr->next, (void *)value, sizeof(typename)); \
        break; \
    case XDR_DECODE: \
        memcpy((void *)value, xdr->next, sizeof(typename)); \
        break; \
    default: \
        /* error handling: unimplemented features of XDR */ \
        break; \
    } \
    xdr->next = (void *)((int)xdr->next + sizeof(typename)); \
    return 1; \
}
```

Figure 8: General template for creating XDR functions for simple data types on big endian architectures.

XDR because it is used by RPC, but the MacOS is one platform without XDR support. To enable compilation on the Macintosh platform, we have had to implement the XDR routines we use. As most simple data types are already in network form, we can simply use the C macro shown in Figure 8 to create the missing XDR functions for most datatypes. For example, the simple macro invocation `easyxdr(long)` will define the C function `xdr_long`, which converts long integers to and from network form.

Although implementing XDR functionality on big endian architectures such as the MacOS is relatively straightforward, more work is required to implement XDR on a little endian architecture. Because the typical Tcl/Tk programmer should not have to worry about such differences in a cross-platform language such as Tcl, we strongly believe that XDR should be incorporated into the Tcl language. With the introduction of sockets, Tcl has provided an elegant, uniform interface to the differing TCP/IP stacks on the various Tcl-supported architectures. The addition of XDR (or a similar network byte-order standard for all primitive data types) would provide seamless interplatform communication and thereby greatly extend the power of the Tcl socket abstraction.

There are several ways in which XDR could be incorporated into Tcl. The most obvious is an `xdr` Tcl command similar to the binary command added in Tcl 8.0. Encoding an integer could then be as simple as executing a command such as `set dataToSend [xdr encode integer $myInteger]`. Decoding a message would require reading the message into a buffer (string) and executing a command such as `set decodedInt [xdr decode integer myBuffer]`. Note that the `xdr` command must consume data from `myBuffer` so the variable name and not its value must be passed.

Another alternative for incorporating XDR would be to use a method similar to the stream filters provided by TclDP [5]. TclDP allows the user to register a filter mechanism with a Tcl channel such that all reads and writes for the channel first pass through the filter. TclDP filters are very flexible and work well when all data passing through the channel must be transformed in the same way (*e.g.*, uuencoding). However, XDR data are processed differently depending on the type of data the user wishes

to read from the channel. Unless the filter procedure knows in advance the kind of data to be read, TclDP filters don't help with XDR. Instead, we need a mechanism that allows the user to pass a filter to each read/write call on the channel. For example, to read an XDR-encoded integer from the channel, we would call the channel read procedure, instructing it to use the integer XDR filter. This would result in exactly four bytes being read from the channel, and the returned data buffer would contain one XDR-decoded integer. Of course, Tcl would still have to provide XDR filter functions for each of the standard C types, but implementing XDR in this way would also provide the general flexibility of TclDP filters.

4.2 Canvas Improvements

Another area of Tcl that we would like to see improved is the canvas widget. Canvases are of crucial importance to our project, as they provide the most important part of our display: the network topology window. In the course of developing the topology display, we found several areas of the canvas that could be improved.

4.2.1 Relief Effects

One omission in canvas functionality is the lack of relief effects for canvas objects. This makes anything drawn on a canvas seem flat in comparison to the relief effects incorporated in the surrounding widgets.

In the case of the network topology window shown in Figure 2, we use sunken rectangles to represent the three buffers at each switch output port. To simulate this effect, we embed other canvas windows in the main canvas. This allows us to set a sunken relief effect on the embedded canvas, which gives the effect of a sunken bar graph. While this works for creating reliefed rectangles, other shapes cannot be created in this way. In addition, the complete canvas window can no longer be printed through the canvas `postscript` method because embedded windows are ignored. The generated PostScript has holes wherever an embedded window appears on the screen.

4.2.2 Selection Based on Multiple Tags

Another weakness in the canvas widget involves the handling of multiple tags. The ability to associate multiple tags with canvas items has proven invaluable in implementing the display application. As described in the previous section, tags are used to implement the path display. However, the canvas find method is only capable of dealing with one tag at a time. There are often times when items need to be selected on the basis of multiple tags. We have implemented this functionality with the `canvasMultiMatch` procedure shown in Figure 9, but it would be far more efficient if this functionality were added to the canvas find function or if a list command were added that returned the intersection of two or more lists.

4.2.3 Megaitems

The Tcl community has expressed a desire to have megawidget support added to Tk. Megawidgets [3, 11, 12] are new widgets created entirely with Tcl code from more basic widgets. No C programming is required to construct megawidgets, and they are indistinguishable from native widgets. A SuperWidget megawidget can be created with Tcl code such as `SuperWidget .myWidget` and configured with code like `.myWidget configure -relief sunken`. Because megawidgets have the same interface as a native widget, they can be replaced by native widgets, if they become available, without changes to the code that uses them. Although megawidget support is not yet an official part of Tk, the Tcl community has addressed this need.

We believe there is a similar need for "megaitems," the canvas equivalent of a megawidget. "Megaitems" are canvas item types created from more basic canvas item types. Like megawidgets, "megaitems" should be indistinguishable from native canvas items so they can be replaced by native canvas items should they become available.

"Megaitems" would have been invaluable in the implementation of the network topology display. Both the rate meters and the buffer occupancy bars are comprised of several native canvas items. We would have liked to have defined these to be new canvas types ("megaitems") and then used them in defining a new "megaitem" type: the ATM switch. While we did write procedures to abstract the construction of these items, it is clear from the code that they are not native canvas types, and significant sections of code would have to be changed to use a native rate meter item, were it to become available. Incorporating "megaitem" support into the Tk canvas would allow the Tcl/Tk programmer to take advantage of the benefits of object-oriented design and would make the Tk canvas more flexible.

```
#procedure to find items with multiple tags in a canvas
proc canvasMultiMatch {canvas tagList} {
    # find all items with the first tag
    set itemList [$canvas find withtag [lindex $tagList 0]]
    #delete first element (used to build initial list)
    set tagList [lreplace $tagList 0 0]

    set resultList [list]
    foreach item $itemList {
        # Get all tags for this item
        set itemTags [$canvas itemcget $item -tags]
        set isCandidate 1

        # Check to see if the item has all of the tags
        foreach curTag $tagList {
            if {[lsearch -exact $itemTags $curTag] == -1} {
                # missing one of the tags; so this item doesn't match
                set isCandidate 0
                break
            }
        }
        if {$isCandidate} {
            # the item matched all tags
            lappend resultList $item
        }
    }
    return $resultList
}
```

Figure 9: Procedure to select canvas items matching a set of tags.

5 Conclusion

In this paper we have presented the design and implementation of the graphical display program for the Thunder and Lightning network protocol simulator. The display application, unlike other Tcl/Tk applications of which we are aware, has the unique requirement that it must be implemented as a separate program running on a machine other than the computer hosting the simulation process. The display and simulator must, therefore, exchange data through the use of sockets. The fact that the display and simulator may be running on dissimilar architectures poses new challenges, despite the portability that Tcl provides. We have used code from the working display application to present the use of XDR in overcoming these inter-platform binary data representation issues. Finally, we have described several areas of Tcl/Tk that could use further development. In particular, we have identified several weaknesses in the canvas widget and described how XDR might be incorporated into Tcl. We believe that the modest effort required to include XDR functionality in Tcl would extend Tcl's socket abstraction beyond that of comparable scripting languages.

References

[1] ATM Forum Technical Committee. *ATM User-Network Interface Specification*. Number af-uni-0010.002. The ATM Forum, 1994.

[2] DARPA Thunder and Lightning Research Review. Technical report, University of California, Santa Barbara, March 1995.

[3] S. Jaeger. Mega-widgets in Tcl/Tk: Evaluation and analysis. In *Proceedings of the Third Annual Tcl/Tk Workshop*, pages 43–52, Toronto, Canada, July 1995. USENIX.

[4] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, USA, 1994.

[5] M. Perham, B. C. Smith, T. Janosi, and I. K. Lam. Redesigning Tcl-DP. In *Proceedings of the 5th Annual Tcl/Tk Workshop*, pages 49–53, Boston, MA, July 1997. USENIX.

[6] M. D. Santos, P. M. Melliar-Smith, and L. E. Moser. A protocol simulator for the Thunder and Lightning ATM network. In *Proceedings of the First Annual Conference on Emerging Technologies and Applications in Communications*, pages 28–31, Portland, OR, May 1996.

[7] M. D. Santos, P. M. Melliar-Smith, and L. E. Moser. Flow control in the high-speed Thunder and Lightning ATM network. In *Proceedings of the Seventh International Conference on Computer Communications and Networks*, Lafayette, LA, October 1998.

[8] M. D. Santos, P. M. Melliar-Smith, and L. E. Moser. A lossless, minimal latency protocol for gigabit ATM networks. In *Proceedings of the Sixth IEEE International Conference on Network Protocols*, Austin, TX, October 1998.

[9] V. Schubert. Using Tk as a remote GUI frontend for 4GL-database applications. In *Proceedings of the Fifth Annual Tcl/Tk Workshop*, pages 171–172, Boston, MA, July 1997. USENIX.

[10] Sun Microsystems. *XDR: External data representation standard*. Sun Microsystems, June 1987. RFC 1014.

[11] S. A. Uhler. In search of the perfect megawidget. In *Proceedings of 4th Annual Tcl/Tk Workshop*, pages 125–128, Monterey, CA, July 1996. USENIX.

[12] M. L. Ulferts. [incr Widgets] an object oriented mega-widget set. In *Proceedings of the Third Annual Tcl/Tk Workshop*, pages 61–76, Toronto, Canada, July 1995. USENIX.

[13] E. A. Varvarigos and V. Sharma. An efficient reservation connection control protocol for gigabit networks. In *Proceedings of the International Symposium on Information Theory*, pages 17–22, Whistler, British Columbia, Canada, September 1995.

[14] E. A. Varvarigos and V. Sharma. The Ready-to-Go Virtual Circuit protocol: A loss-free protocol for gigabit networks with FIFO buffers. *IEEE/ACM Transactions on Networking*, 5(5):705–718, October 1997.