Pure Tcl/Tk Train Traffic Supervision System

J.Lima
jlima@eso.org

Physics Department
Faculty of Sciences, Univ. of Lisbon
1749-016 Lisbon, Portugal

J.Rocha
jose.rocha@neumax.pt

Aerospace Laboratory
Ins. for Industrial Technology
1649-038 Lisbon, Portugal

Abstract

This paper describes an experimental subway train traffic supervision software written using pure Tcl/Tk. Train traffic supervision software interacts with traffic control PLCs (Programmable Logic Controller) to retrieve information from the railroad and interact at the exploration level, for instance, granting or deferring a train departure. This application demonstrates the effectiveness of Tcl/Tk for use in industrial control and supervision software. Emphasis will be put on coding techniques and programming paradigms used and how they were implemented into the Tcl/Tk model.

1 Introduction

A train traffic supervision application is used in train traffic control systems with two purposes: observation of the railway; operation of the railway. At the operation level one may grant or defer a train departure, define a new itinerary or manually conduct maneuvers in exceptional situations. Otherwise railway normal operation is conducted automatically by a railway certified equipment, usually a dedicated PLC (Programmable Logic Controller). Figure 1 shows a simplified but realistic architecture for a train exploration system.

The operator workstation, which runs the supervision application communicates to the train operation PLC the intentions of the operator. The PLC then verifies if and when the necessary safety conditions to perform the requested operations are met and just then will conduct the proper sequence of maneuvers. The PLC continuously informs the supervision system about any change of state in the railway elements.

This means that passenger physical integrity or equipment safety doesn't depend on the traffic supervision application. The reliability and robustness of the supervision application is rather a matter of availability and not of safety of the railway system.

A supervision application must provide a railway synoptic which gives the operator a feedback of what is going on. Typically, by means of toolbars or context menus, the operator is able to act upon the railway status as well. The software application described in this paper provides exactly this kind of functionality.

2 Motivation

The LCCS[1], which stands for *local control computer software*, is an experimental supervision application, and was initially developed as a concept demonstrator

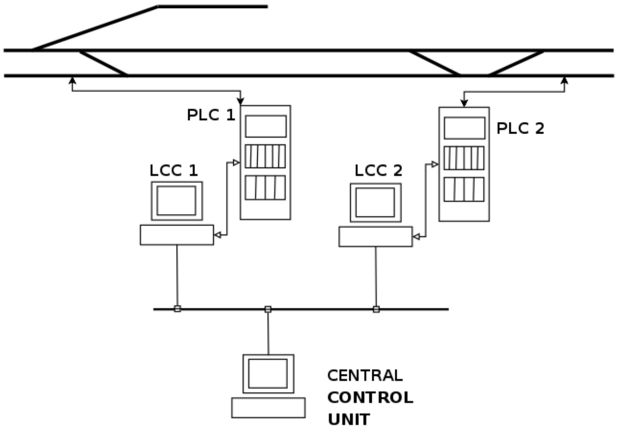

Figure 1: Train traffic control system

in the context of a development project proposed to the ML (Lisbon Subway Train Company). The proposal was not accepted. The aim of this project was to upgrade previous supervision software with in-house developed software. The ML declined the offer and the demonstrator remained property of the authors, which then decided to make it open-source.

A snapshot of the LCCS running is shown on Figure 2.

3 Application overview

The LCCS application supports one or more simultaneous windows, allowing the operator to visualize a large portion or the entire railroad in one window and some details in a magnified window. The windows allow arbitrary zooming (implemented entirely in Tcl) and panning. Many other visualization options are user configurable by means of pop-up menus. Per window toolbars are provided for easy access to common functions.

The LCCS application is a build up from several modules performing specialized tasks.

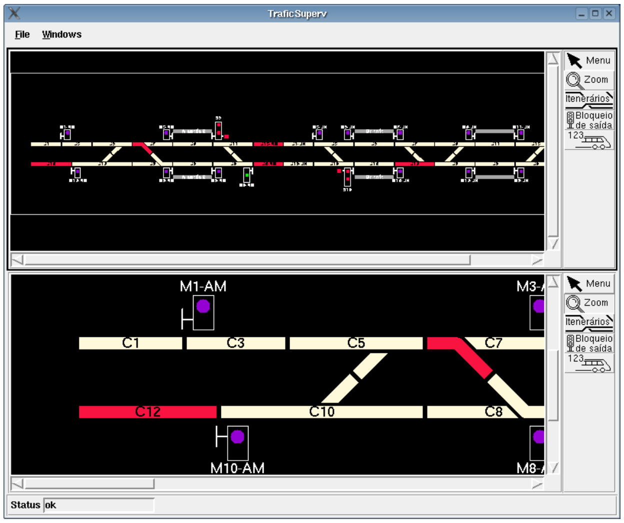

Figure 2: Snapshot of the LCCS main window

- A graphics engine built around a Tk canvas and a set of basic procedures for zooming, panning and drawing.
- A configuration client responsible for reading and parsing the railway description files and instruct the graphics engine to draw the synoptic.
- A communication tool, which will establish the communication protocol with the PLC and update the synoptic on the graphics engine.
- A user interface module, which will dispatch the operator events, converting them into commands that will be sent to other modules.

The communication between modules is performed using the Tk *send* mechanism. Figure 3 depicts how modules interact with each other and the environment (configuration files and input).

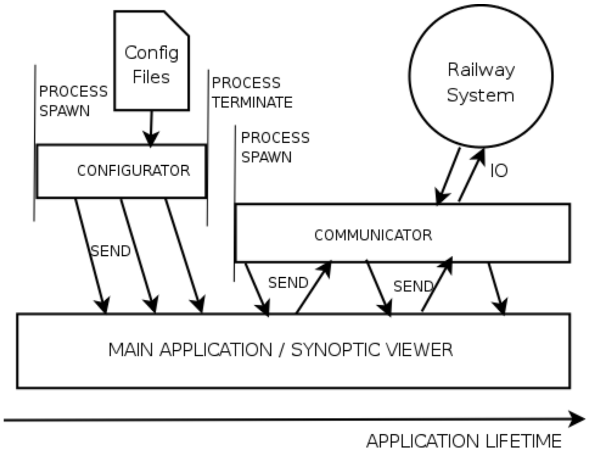

Figure 3: Application inter-process communication

4 Programming model

Tcl has a minimalist syntax which doesn't immediately reveal the powerful coding techniques which one can use. As a consequence, it happens frequently in Tcl that there is no obvious solution to a particular problem and there are several non obvious solutions. This may give origin to code which is difficult to understand. However, very concise solutions can be attained for a broad class of problems.

The implementation of the application herein described relies heavily on the use of slave interpreters and *reflective programming*. One must explain what the authors mean by *reflective programming*.

4.1 Reflection

We used a *"reflective"* coding technique to reduce the code size. The emphasis on the word *"reflective"* was deliberated since it has two meanings: on one hand, it is applicable, to some extent, in the ordinary sense of *reflective programming* as defined by [6], and on the other hand, it expresses the fact that the code so built, at run time, *"reflects"* the railway configuration. While inspecting the reflective code portions one can infer about what is being represented. That is to say, the code *"reflects"* reality.

Reflection comes into play when the *Configurator* process reads the synoptic configuration data, where each railroad element will origin a set of new procedures. These procedures will thereafter be used to change the state of the synoptic.

One should remember that the reflection in Tcl is possible because code and data share the same representation; and it was made practical because the Tcl provides a *safe* control of its interpreters. However, one should be careful because the reflective techniques tend to obscure the code.

The reflective code is only accessible through the `info proc` command because you cannot read it anywhere. That code is the result of the "Cartesian product" of a code template by the configuration data.

One possible implementation for the code interacting with the synoptic is to provide for every possible interaction procedures which take as arguments ids for the synoptic elements, perform all the necessary checks to validate the id, and execute the requested operation.

By using reflection when the element is instantiated, all the necessary procedures are created on the fly and are immediately bound (typically, by invocation of the Tk bind command) to the particular synoptic element they refer to. Also, when some synoptic element expires and is deleted, all the associated procedures are cleaned up.

This process may be much more complex, since newly created procedures can write more code.

If one browses through the LCCS code, one can't find any code to manipulate or interact with the synoptic elements. That's because that code is build when the synoptic elements are instantiated.

4.2 Everything is Tcl

The configuration files, the data stream from the PLC, the interprocess communication and obviously the code itself are all forms of Tcl. The idea is simple: if one has to exchange or process data, one should use Tcl to represent that data. Doing so, one can rely on the available Tcl interpreter to parse the data, or even better, to execute the data.

The send mechanism was used For inter-process communication, and is actually sending Tcl code to be executed in the context of the target application.

The configuration files are Tcl code executed within a tailored safe interpreter.

The data stream coming from the PLC consists of `address:value` pairs. Each pair represents the address of a certain variable followed by its current value. Since the addresses are unique, one can take them as variable names. This kind of data can be easily mapped into the Tcl syntax simply by translating it conveniently into a ascii string in the form: `''AXXX YY''`. For example,

```
A010 1
A011 0
```

The conversion may or may not be necessary at all, depending on the protocol format for the communication between PLC and LCC.

So, whenever the application needs to parse data, it sets up an interpreter (preferably, a safe interpreter) loaded with the necessary procedure set, and sources the data.

5 Implementation

The current application implementation was developed for Linux, but should run without problems in any system running X-windows. Windows family of operating systems are à-priori excluded because the Tcl/Tk port for Windows lacks the “send” mechanism [4].

The synoptic itself is a Tcl/Tk engine build around the canvas widget. The engine accepts commands through the Tk send mechanism. When the application is first launched, the Configurator is spawned as a background process. When the Configurator is done with its task, it terminates and the Communicator process is spawned.

The Communicator starts to listen a specified socket or serial port, depending how the PLC communicates with the LCC (local control computer), and using the same ”send” mechanism updates the synoptic to reflect the railway state.

The PLC used at ML was a MICROLOK® from USS. Currently, the LCCS implements the MICROLOK® protocol only partially.

Railway state. The way each particular system represents the railway state may vary. In the present case, the state is represented by `address:value` pairs. Each address maps directly into the PLC memory and its value corresponds to the state of a particular railway element. Since the address mapping is persistent, one may consider the addresses variable ids. The PLC issues an `address:value` pair whenever the value changes. A burst with all available `address:value` pairs must be sent periodically, or at least on startup.

Most variables used by the PLC to represent the railway state are boolean variables, but some assume more than two different states. Boolean variables adequately represent the state of a rail circuit, which can be either vacant or occupied. Some of the traffic lights can also be represented by boolean variables. However, some elements may only be represented by state variables with three or more states. For example, railway switches can be switched to right or left, but can also be in transit. This in transit state must be observed, since no train can be allowed to run into the switch while in transit. Many traffic lights have also more than two states.

6 Configuration

To read the configuration files, the application issues the following commands:

```
 #
 # Load The Configuration
 #
 Config::Colors [file join $ETC Colors]
 Config::Scales [file join $ETC Scales]
 Config::LanguageStrings \
              [file join $ETC Language]
 #
 # Load Synoptic
 Config::Synoptic $canvas \
       [file join $ETC global Synoptic]
 #
```

The configuration data plays an important role in LCCS. The configuration is split into two parts. The first part deals with user preferences, like colors and language settings. The second part deals with building the synoptic itself. It is in the second part that reflection comes into play.

Many new code lines will be built when loading the synoptic. Some of that code may even be used during its own configuration.

An `include` directive is provided which allows large synoptic to be loaded hierarchically, simplifying the configuration process. An example of how a synoptic description looks like is given in the following listing:

```
# Railway synoptic
include global/Segments
include global/Scales

# Rail circuits
#    id   segments     variable label
cdv C1-AM  {seg1}           A001  "C1"
cdv C3-AM  {seg3}           A003  "C3"
cdv C5-AM  {seg5a seg5b}    A005  "C5"
cdv C7-AM  {seg7a seg7b}    A007  "C7"
cdv C9-AM  {seg9}           A009  "C9"
cdv C11-AM {seg11a seg11b} A012 "C11"
cdv C13-AM {seg13}          A013 "C12"

# Trafic lights
#   id   coordinates  anchor  \
                       variable label
s2 M1-AM {3500 8700}  {top left}
                       A014     "M1"
s2 M3-AM {9000 8700}  {top right} \
                       A015     "M3"
sb SB-AM {11400 8700} {top left} \
                       B016     "SB"
s4 S5-AM {11000 8700} {top left} \
                       A016     "S5"
```

It is immediately clear that this configuration file obeys the simple Tcl grammar and punctuation rules. For example, the code line

```
cdv   C1-AM   {seg1}   A001  "C1"
```

is actually a valid Tcl command, `cdv` is a Tcl procedure which will process rail-circuit entries. This command not only will setup the canvas widget to display properly a new rail-circuit, but also will write the code necessary to deal with all issues related to the `C1-AM` rail-circuit.

The line `include global/Segments` in the configuration file refers to another file which contains essentially the coordinates off all segments that compose the synoptic. The content of this file was created with the help of a vector graphics program, and then converted to the LCCS graphics language.

7 Conclusions

These days were the computer science is dominated by the pos-Object oriented jargon, it may sound inopportune to highlight the following Tcl/Tk powerful features: code and data share the same representation; Tcl offers a great deal of control over interpreters. However, these two features made possible to implement, using the reflective coding techniques, the LCCS using a surprisingly small number of lines of code.

Using the `wc` unix tool one can easily get the line, word and character count. The count for all the modules is:

```
# cat TraficSuperv Communicator \
Addresses Configuration Graphics \
MathTools Menus Signalization1 \
Synoptic Toolbar ViaElements \
Windows | wc
   2007    7680   54388
```

This is a fairly small number of code lines for an application of this complexity.

We shall now compare with the number of lines of the configuration data. Keep in mind that the configuration data, mainly the synoptic description, describes an unrealistically small railway system with just three stations.

```
# cat Colors Language Scales \
Strings.portuguese alameda/Segments \
alameda/Synoptic global/Segments \
global/Synoptic orient/Synoptic \
orient/Segments| wc
     544    1983   13927
```

As one can see, the configuration data is over 1/4 the total code size. This is an impressive proportion if one considers that the 544 lines of configuration data will generate around 2500 lines of new code.

References

[1] LCCS home page:
http://kdataserv.fis.fc.ul.pt/ jmal

[2] *Regulamento de Sinalização*, Metropolitano de Lisboa, E.P., 1994.

[3] *MICROLOK® — Vital Application Logic Programming*. UNION SWITCH & SIGNAL, October 1991.

[4] Brent B. Welch, *Practical Programming in TclTk*. Prentice Hall PTR, 3rd Ed, 2000.

[5] N. Demers and J. Malenfant. *Reflection in logic, functional and object-oriented programming: a short comparative study*. In Proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI, pages 29-38, August 1995.

[6] Reflective programming languages:
http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/malenfant/ref96/node2.html