Tcl Fast Track: From a Novel Concept to Global Launch in Eleven Months

Clif Flynt
Noumena Corporation
Whitmore Lake, MI

Bruce Ross
The ROMaN Project, Inc
Cary, NC

September 6, 2013

Abstract

The LORACIS^{TM} Remote Monitoring software was developed for the clinical trials industry to allow the rapid verification of reported data from human studies while markedly reducing unproductive time and travel expenses. The successful completion of this unique and complex software was achieved in just eleven months due to our selection of Tcl as the programming language.

1 Introduction

1.1 Clinical Trials and the Monitoring of Study Data

Every prescription drug on the global market, and all diagnostic and therapeutic devices, must first undergo a long process of scientific evaluation and clinical testing (clinical trials) to find out if they are effective and safe for use. In the U.S. the Food and Drug Administration (FDA) controls this approval process; other nations have similar regulatory agencies.

These clinical trials are necessary and scientifically important, but are also very expensive to conduct. In 2012, the global cost for all clinical trials exceeded $42 billion. Approximately 25% of the total cost for conducting a clinical trial is spent on the verification and validation of the study data reported by the investigating physicians. This process is called monitoring and involves frequent travel to their clinics and hospitals for comparing the study medical records versus the study data.

In the end, it is the consumers of health care who must pay these costs every time our doctors write a prescription or orders a diagnostic test. In 2011, the FDA issued a draft guidance document that encouraged the clinical trials industry to develop new ways of monitoring clinical trials, including the adoption of new technologies and implementation of risk-based methods. While only a draft document, the FDA was giving study sponsors (e.g. pharmaceutical companies), consulting firms (CROs, clinical research organizations), and investigators the green light to develop new systems and processes to improve the quality and lower the overall cost for each study.

LORACIS^{TM} Remote Monitoring (LORACIS) was conceived in early 2011, prior to the FDAs publication of the draft guidance document; however, following that event our development time-line had to be compressed since it was likely that other companies would create similar products.

1.2 Regulatory Background

Every clinical trial must comply with hundreds of international, Federal, state, and local regulations or else risk being hit with fines, lawsuits, investigations, and rejection of their study results. The full regulatory background is far beyond the scope of this paper, but the two most relevant categories of regulations must be reviewed to place the rest of the story into its proper context.

Patient Privacy and Data Protections: The clinical trials industry is arguably one of the most heavily regulated industries in the world. Each country has one or more government agencies dedicated to creating regulations for the research, manufacturing, packaging, and sales of drugs, medical devices, and diagnostic tests in response to laws passed by their respective legislative or judicial branches.

These agencies include the U.S. Food and Drug Administration (FDA) the Health Canada Agency (HCA), and the European Medicines Agency (EMA). In the U.S., there are additional government agencies with regulations that must be complied with during the the conduct of clinical studies, including the Department of Health and Human Services (DHHS) and the Department Homeland Security (DHS). One of the more familiar of these non-FDA regulations is the Health Insurance Portability and Accountability Act of 1996, better known as HIPAA; an even more restrictive counterpart in Europe is the EU Directive 95/46/EC - The Data Protection Directive. Both require the protection of patient privacy and safeguarding of patient health information (PHI) regardless of whether they are participating in a study.

Computer Systems Used in Clinical Trials: In 1997, the FDA first released its regulations covering the use of electronic signatures and computer systems in clinical trials as Title 21 Code of Federal Regulations, Part 11 Section 11.1 (a), now commonly shorted to 21 CFR Part 11. As computer technology has advanced the regulation has been updated (2003, 2007) and will be updated again as needed. Virtually all software, computer hardware, and electronic systems that generate, transmit, store, or manipulate clinical trials data must comply with this regulation.

1.3 Centralized Monitoring

One time-tested method for reducing the number of locations a monitor must visit is to create a central repository of photocopied patient and study records. However, in order to be acceptable for use in verification of study data, each photocopied page should be certified (usually by initialing and dating of each page by the person doing the copying) as a complete and accurate copy of the original. Also, in order to comply with the privacy regulations all identifiable patient information (e.g. name, address, medical record number, and month/day of birth) must be obliterated (redacted, de-identified) and replaced with a study subject identification number. The resulting binder of photocopied records is called a shadow chart and may be, or not be, an accurate compilation of the study records.

Shadow charts maintained in a central location (central monitoring) are only possible for certain dispersed networks of study sites, and still require significant travel time and cost to visit the central locations. Also, the timeliness of the centralized files may range from good to abysmal. In recent years more and more medical clinics and hospitals have been moving from the use of paper to electronic medical records (EMR). EMRs have increased the potential risk of privacy breaches as more individuals are given access to the EMR computer systems for medical as well as research purposes.

As a result, the FDA's final guidance document (August 6, 2013) defined:

Required Functionality in an Effective Remote Monitoring Platform

The functionality required in a comprehensive remote monitoring platform should include the following attributes:

- Compliance with all applicable privacy protection regulations, including HIPAA and EU-DPD
- Compliance with 21 CFR Part 11
- Compatibility with commonplace computer and server systems
- Accessible from any computer with Internet access
- Ease of use for study site personnel, monitors, and other sponsor personnel
- Maintain strong document security protection at all times
- Support for multiple global languages
- Permanent audit trail
- Overall cost for use must be less than traditional monitoring systems

1.4 The Winding Road that Led to TCL

In June 2011, ROMaN began talks with several document management software development firms in search of one with the requisite experience, capability, and interest in creating our remote monitoring application. From the beginning, most were surprised by the proposed complexity of functions and security features outlined in the draft technical specifications; some declined to offer proposals based on this alone. Others were intrigued but could not fit the project into their schedules for many months to even years later, while others yet were not interested as their potential profit for making our products was simply too small.

By September 2011, a potential programming firm was located in New Zealand that initially expressed interested and had relevant experience plus a good pedigree - they have produced cutting edge document management software for more than a decade. Our CEO visited the company and within days they produced a very rough proof of concept application that showed the most critical component called the Associated Data File could, in fact, be made. Unfortunately, as negotiations progressed during the following months it became apparent that several other important features could not be made by their team while their projected budget grew beyond our ability to proceed.

In late October 2012, we contacted an expert in an older programming language known as Clarion. This language was first released in 1986 and though a bit antiquated, its most appealing feature is the use of a proprietary database engine that would allow for systemic encryption of the data as well as its use of readily available "templates" (both open source and commercially available) that can be quickly combined to produce much larger packages of functional code that would otherwise require weeks or months of more traditional programming effort. Within a month he produced a more full-featured proof of concept than his predecessors, and again it demonstrated the overall concept, by then code-named LORACIS, was at least possible to achieve. Unfortunately, over several months, it became apparent that Clarion had significant difficulty in providing the needed functionality as the third-party template library used to produce searchable PDF documents was not able to perform as was desired. It was also difficult to interface with other components using .NET and other more modern languages.

In February 2012, we approached Noumena Corporation based on referrals from the author of one component we were in-licensing. They expressed interest and following a verbal agreement, produced a proof of concept application within a month. Unlike the two previous POC applications, his showed it was possible to overcome the more worrisome challenges by using the Tcl programming language. He was able to begin working in earnest by June 2012, and the rest is history.

LORACIS^{TM} entered beta testing at actual clinical research centers during October, and the first production version was locked down by May 2013 in time for our scheduled global launch date of June 24, 2013.

This concludes the recitation of the motivation, challenges, and decision pathways that led The ROMaN Project to use Tcl for creating what is today the only comprehensive remote monitoring platform available to the clinical trials industry. The next part of this paper is devoted to a more technical explanation of how Tcl achieved this goal.

2 Solution

Some languages and development environments have ceilings that limit their usefulness.

The Clarion approach to developing the LORACIS^{TM} application ran into difficulties when there was a need for functionality that didn't exist within the Clarion framework.

In his keynote talk at the 1997 Tcl/Tk Conference Brian Kernighan reported that he found Visual Basic to be a faster tool for developing an application than Tcl/Tk, but he reached a point where the functionality he needed was not available, and the application could not be completed.

He pointed out that Tcl/Tk does not have a ceiling that an application cannot exceed.

Many languages share this facility of having no ceiling - "C" is a classic example. However, many of the limitless languages are also mid-level or low-level languages that carry long development times with them.

2.1 Choosing a development platform

The first step in determining if the application could be created and what the best development language would be was to define the base sets of functionality.

The primary requirements for this application are:

- Authenticate a local user.
- Validate per site and per study subscription licenses.
- Support multiple languages and alphabets.
- Display shrink-wrap-quality GUI.
- Interact with an encrypted database.
- Accept document from scanner.
- Accept document from print queue.
- Accept existing document.
- OCR process a scanned document.
- Redact personal information from PDF document.
- Modify PDF File metadata.
- Encrypt processed document.
- Perform remote authentication.
- Transmit document via SFTP.

The short development cycle forced the decision to a high-level scripting language as the primary language.

The need to interact with external libraries suggested a language with strong support for connecting to C and C++ libraries.

The following features are either natively supported by Tcl, are available in the `Tcllib` libraries or can be easily constructed with a few lines of code:

- Authenticate a local user.
- Validate per site and per study subscription licenses.
- Support international alphabets.
- Display shrink-wrap-quality GUI.
- Interact with an encrypted database.
- Encrypt processed document.

The other requirements could be met by interacting with existing external libraries which were available either as `.NET` components or linkable libraries.

After considering multiple languages, Tcl was the obvious choice:

- Tcl is a good platform for rapid prototyping.
- Tcl has strong support for integrating with external packages. Tcl supports .NET interactions as well as shared library, direct linking to external libraries and executing separate tasks.
- Tcl uses 16-bit Unicode for all strings, which provides strong internationalization support.
- Tk delivers high-quality GUI tools.
- Tcl tools are mature and stable. In particular the database interface with SQLite is very stable.
- Tcllib provides a wide range of industrial strength libraries.

Tcl is a high level language, thus application development is faster in Tcl than it would be in C or Java.

It's estimated that one line of Tcl replaces 10 lines of "C" code. In *The Mythical Man-Month* Fred Brooks claims that a programmer produces 10 lines of code per day, regardless of the language. Using a high-level scripting language instead of a mid-level compiled language reduces the development time by a factor of 10.

Language features are also important. A language that lacks key functionality will delay a project's completion while the programmers work around or correct the deficiencies.

Tcl has long been plagued by the lack of a native Object Oriented support. OO Programming is not the answer to all problems, but it is a good answer for some problems. The lack of native OO support hasn't stopped people from developing applications using `[incr Tcl]`, `SNIT` or other OO extensions, but having the OO support in the core where it's guaranteed to be available is a benefit.

Tcl 8.6 beta was used for the initial development of the ROMaN application. When the first 8.6 release became available that became the base platform. `TclOO` features were used effectively for complex widgets that were developed for this application.

2.2 Assembling the tools

Tcl stands for Tool Control Language, but that doesn't make it a complete toolbox. There is a rich set of development tools available for Tcl applications. Using these can cut development time even further.

Developing the ROMaN application used several off-the-shelf and semi-custom tools including:

- `Tcllib`
- `SQLite-SEE`
- `SWIG`
- `EditTable`
- `tktest`

2.2.1 Tcllib

Tcllib is not part of the standard Tcl distribution, but as `perl` would be less useful without `CPAN`, Tcl is enhanced by downloading Tcllib either from ActiveState or `http://core.tcl.tk/tcllib`.

These Tcllib packages were used in the ROMaN desktop application:

`twapi`	Invoke windows API calls to set focus, raise windows, when invoking Windows native applications
`dde`	To communicate with native windows applications.
`tcom`	Interact with .NET objects

`des md5`
`sha256`
`cryptkit`
	Encrypting and decrypting data files

`msgcat`	Internationalization

The `Tcllib` code collection is a controlled library. The contributions to `Tcllib` include validation tests and documentation.

2.2.2 SQLite

The SQLite database library was developed by D. Richard Hipp and was described at his invited talk and tutorials at the 2006 Tcl conference. SQLite is an embeddable SQL engine that uses a single file as the database rather than using a client and server. It is ideal for applications that have complex data representation requirements but do not need to control transactions with multiple writers.

SQLite has been distributed with Tcllib and other "Batteries Included" Tcl distributions for over a decade.

A downside to SQLite for this application is that an SQLite database can be opened and read with the `SQLite3` program. The SQLite database will contain a patient's personal information. Access to this information needs to be restricted to only individuals who require access to it.

Richard also provides a non-free variant of SQLite (`SQLite-SEE`) that encrypts the database file. A user needs to know the encryption key in order to access data in this file.

The SEE variant of SQLite solves the problem of restricting access to a patients personal data.

The SEE extension to SQLite supports the same Tcl interface as the usual SQLite extension. This made it easy to use standard SQLite for the early proof-of-concept and then purchase the SEE extension to be linked with the application in the last phases.

2.2.3 SWIG

Tcl does not have native tools for manipulating a PDF file: changing the metadata, performing OCR, or adding watermarks. This functionality is provided by the PDF-X library from Tracker Software:
`http://www.tracker-software.com/`.

This library is provided as linkable C and C++ dynamic link libraries. As distributed they don't interface with Tcl.

Fortunately, Tcl was designed to be interfaced to external libraries. The glue to turn an external library into a Tcl extension is generally under 100 lines of "C" code.

Getting data into and out of libraries that weren't designed around Tcl's "everything is a string" mantra requires some "C" code to translate from Tcl Objects a function's native `float`, `int` or `struct` formats.

This translation code is not difficult or tricky, but it's faster to let a code generator create it, rather they typing it by hand.

The `SWIG` application will examine an include file and generate the "C" glue code to link a library into a Tcl extension. `SWIG` stands for *Simplified Wrapper and Interface Generator*. This package was designed by David Beazley and introduced at the Tcl conference in 1996. It's available from
`http://www.swig.org/`

`SWIG` generates an `Init` function that invokes `Tcl_CreateObjCommand` to create the Tcl commands and also creates commands to translate from Tcl format to native data formats required by the library.

In practice, the library's include file usually needs to be modified, and reduced to just the parts of the API an application needs. The PDF-X library is extensive, and only a half dozen entry points were needed for this application. The need to rework the include file impacted the development time.

The `CriTcl` package is also useful for creating interfaces to external libraries. For example, `CriTcl` was used to create the `cryptkit` encryption extension.

Using `CriTcl` would have avoided the cost for reworking the include file at a cost of doing a bit more hand-coding and defining interfaces. I find `CriTcl` to be better than `SWIG` for relatively simple external APIs, but the complexity of the data structures used by the PDF-X API made `SWIG`'s parsing and code generation more attractive than creating the glue code by hand.

2.2.4 `EditTable`

A data-centric project requires data entry screens. These are simple and easy to write, but in the early phase of a project when the understanding of what's needed is changing, the time to recreate simple screens can add up.

The `EditTable` package will generate a data entry screen from an SQL schema. It was described at the 2012 Tcl Conference.

The `EditTable` package is constructed using the TclOO mega-widget design pattern described by Donal Fellows (Tcl/Tk Proceedings 2009). This pattern emulates a standard Tcl widget, allowing the new `editTable` object to be created like any other Tk widget.

Syntax: `editTable` *`widgetName mixin dbEngine args`*

`widgetName`	Name of the widget, normal Tk style
`mixin`	Name of the db engine mixin
`dbEngine`	Name of the db engine to use
`args`	Optional arguments for opening the db engine

Creating an `editTable` connected to an SQLite database named `test.db` would resemble:

```
set obj [editTable .t1 SQLITE3_support sqlite3 -dbArgs test.db]
```

Once an `EditTable` object has been created, the `getSchema` and `makeGUI` methods will create a GUI that's adequate for for testing and initial data entry.

The `getSchema` method retrieves schema information from the database or another source. The default behavior is to query the database. This is used internally to build GUIs for tables containing relations.

Syntax: `editObj` *`getSchema tableName`*

`editObj`	A widget created with `editTable` command
`getSchema`	Return the schema for a table
`tableName`	Name of the table to return Schema for

The `makeGUI` method is the workhorse that builds a GUI within the `editTable` frame. It can build a GUI for any table defined within the database. The `editTable` object can rebuild itself to display a different table as necessary. The default GUI includes buttons to perform simple searches, and add, delete or modify a record.

Syntax: `editObj` *`makeGUI schema`*

`editObj`	A widget created with `editTable` command
`makeGUI`	Construct a GUI within the editTable object frame.
`schema`	a Schema - may be return from `getSchema`

The next example shows initializing a sample database and creating a simple GUI for a table.

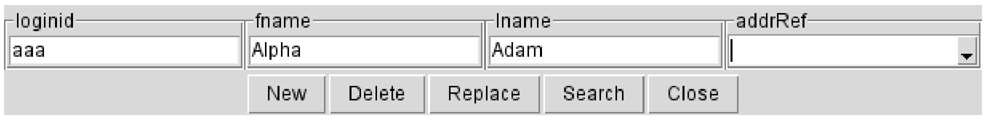
```
toplevel .tt
set obj [editTable .tt.t1 TDBC_support sqlite3 -dbArgs test2.db]
set db [$obj config db]

$db allrows {
  CREATE TABLE person (
    id integer unique primary key,
    loginid text, -- loginID
    fname text, -- First name
    lname text, -- Last name
    addrRef integer references addr
    );
}

$obj makeGUI [$obj getSchema person]
$obj config -table person
$obj populateBySearch "loginid = 'aaa'"

pack .tt.t1
```

The generated GUI resembles this:

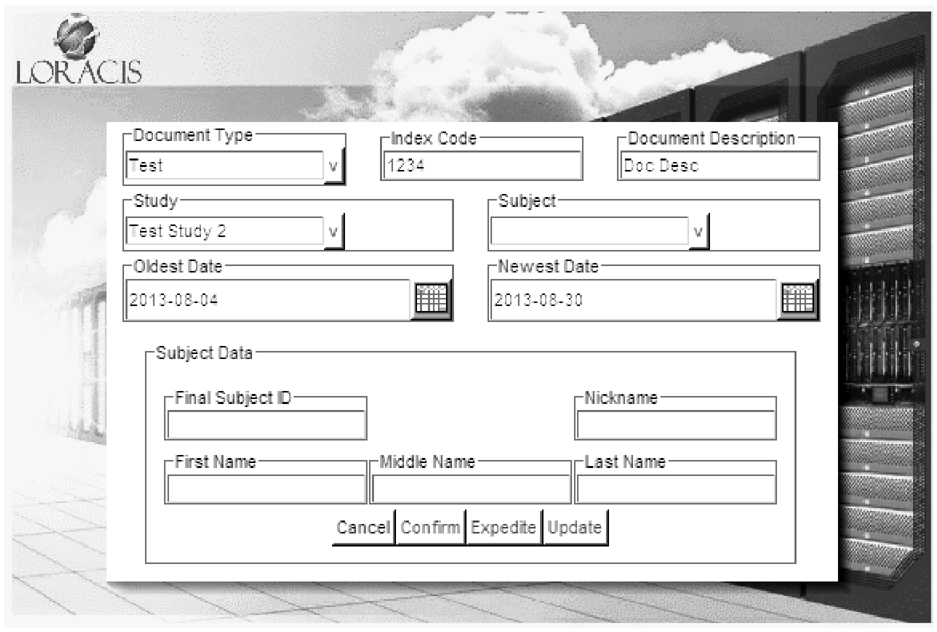


loginid	fname	lname	addrRef
aaa	Alpha	Adam	

Figure 1: Un-instrumented EditTable GUI

This was suitable for the initial development and constructing bits of functionality. The advantage from a development time perspective is that no time was wasted rebuilding better looking GUIs while the schema was still evolving.

The default GUI generated by `EditTable` was not suitable for end users, but with an instrumented schema it was quite adequate.



Document Type	Index Code	Document Description
Test	1234	Doc Desc

Study	Subject
Test Study 2	

Oldest Date	Newest Date
2013-08-04	2013-08-30

Figure 2: LORACIS^{TM}

2.2.5 tktest

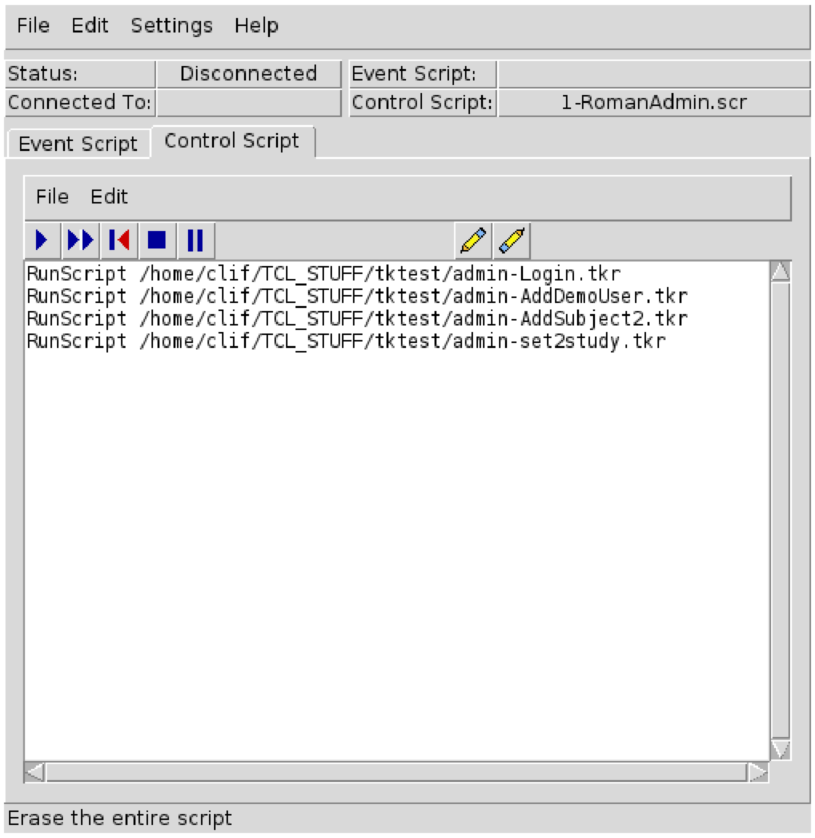
Coding projects always proceed faster at the beginning than they do at the end. It can feel like every day completes half of the remaining tasks, and final completion will take forever.

One reason that projects slow as they reach completion is that developing and testing standalone pieces is relatively fast compared to integrating and testing the final application, and then re-testing everything when a low-level bug is discovered.

The amount of time spent in re-running tests gets longer, and when the tests are run by hand the number of times they need to be restarted because of a mis-key increases.

The `TkReplay` application was developed by Charles Crowley, and reported at the 1995 Tcl Conference. Clif Flynt updated and expanded this into the `tktest` package and described in a paper at the 2004 Tcl conference.

In brief, Tk widgets can be configured with a binding to report when an event occurs. These events can then be played back to simulate a user changing focus, typing keys, and clicking buttons.



Status: Disconnected
Event Script:
Connected To:
Control Script: 1-RomanAdmin.scr

```
RunScript /home/clif/TCL_STUFF/tktest/admin-Login.tkr
RunScript /home/clif/TCL_STUFF/tktest/admin-AddDemoUser.tkr
RunScript /home/clif/TCL_STUFF/tktest/admin-AddSubject2.tkr
RunScript /home/clif/TCL_STUFF/tktest/admin-set2study.tkr
```

Erase the entire script

Figure 3: tktest

The `tktest` application was used extensively during the middle and end phases of this project.

During the mid-phase of project, when individual pieces are being integrated and the functionality is being extended it's very common to need to step through several screens and inputs to get to the section of the application where new code is being created and tested.

As that sequence of events gets longer, the probability of mis-typing and needing to restart gets larger and the fatigue factor on the developer grows. Being able to save the initial steps that lead to the new code and replay them both speeds the code/test/debug debug cycle, and enables the programmer to be productive for a longer period of time.

The LORACIS suite of applications is not quite shrink-wrap. The Loracis client and servers must be configured for initial licenses, studies and authentication tokens. The configuration is currently done in separate steps and must be validated before an installation kit is sent to the user.

The final validation of a distribution involves selecting, processing and uploading 16 documents. For obvious reasons, `tktest` is used to validate the distributions before they are sent to a client.

2.3 Conclusion

The extensible nature of Tcl/Tk meant that the complex LORACIS^{TM} application could be written using Tcl/Tk.

The high-level nature of the Tcl/Tk meant that the reduced development cycle was possible using Tcl/Tk.

The available libraries and packages meant that production grade code did not need to be written from scratch, saving even more time.

Code generation and testing packages reduced development time yet again.

In the end, a revolutionary new product was taken from conception to distribution in less than a calendar year and about 6 man months of developer time. This is a fraction of the time and cost that would have been required using other tools. The final product includes about 8,000 lines of new Tcl code and 300 lines of ”C” code.