

Collaborating applications: Tequila takes Tcl further

Jean-Claude Wippler
Equi4 Software
jcw@equi4.com

ABSTRACT

Tequila is a "middleware" kind of infrastructure written in pure Tcl, which makes it easy to write collaborating applications. A first version has been extremely effective and successful in two very different projects: a distributed telephone system testing system which has processed a quarter million daily events non-stop for over a year, and a highly interactive multi-session chat and discussion board used in high-school collaboration research. Several other projects have demonstrated the convenience of having such a generic and robust infrastructure. The next generation of Tequila will be presented, aimed at providing an even richer foundation to build on. It is heavily influenced by the first version as well as by GroupKit, a generic collaboration environment developed at the University of Calgary. Examples will be presented of how the new design simplifies building collaborative applications, such as a basic chat system in just a few dozen lines of Tcl. Tequila supports structured data and automatic persistence, which has a major impact on how clients and servers are designed. The basic components (pools, rpc endpoints, notifiers) will be presented to show how Tequila enhances Tcl's strengths, i.e. seamless integration of the Tk GUI with networking and persistence.

Introduction

It can take a considerable amount of code to implement applications that combine a user interface, network communication, and data storage. Even though this is an area that is extremely well suited to Tcl, there simply is a lot of ground to cover.

The Tequila package described here describes an approach which has been used successfully in the past, and which is now being re-implemented to apply the lessons learned so far as well as to better take advantage of the tools available today.

We will first look at the specific requirements of collaborative applications, how the first version of Tequila ("T1") dealt with these and what experience was gained. Then a new design ("T2") will be presented which cleans up the basic design, in order to make the code more modular and extensible, and extends it with "pools" as the basic concept for persistent state and data exchange and "notifiers" to handle change propagation within and between multiple collaborating application instances.

The last section will cover the details of Tequila, and can act as demonstration of the new implementation. It includes examples to illustrate the mechanisms and idioms in practical use.

Collaborating software

There are a number of aspects that set collaborating software apart from other types of software systems:

- Multiple processes, machines, platforms
- Everything happens all over the place
- Event-driven, asynchronous socket I/O

- Keeping running applications consistent
- Dealing with network failure at any time

First of all, collaborating software consists by its very nature of multiple applications, running independently, used by different people on workstations which are connected via a network. In the current design, Tequila is aimed at a configuration with one or more long-running central servers, and any number of client processes logging in and out over time.

Collaborating applications must be prepared to deal not only with changes coming via the GUI, i.e. due to local interaction, but also with state changes received via the network. Collaborating applications must be event-driven for both user interaction and network activity, to maintain a responsive user experience. They must also be very careful about consistency – so that inevitable latencies between the different clients do not cause confusion, or worse: inconsistencies across clients whereby different people end up working with different versions of the data.

Lastly, collaborating software has to deal with less-than-perfect communication channels, even if the hardware it runs on is reliable. Networks occasionally stop working, so loss of connectivity must be dealt with. In the general case, this can be a very complex requirement.

Tcl is a good fit

The Tcl scripting language, and the Tk GUI toolkit are an extremely good fit for collaborating software for a number of reasons:

- Event-driven I/O (fileevent)
- Everything is a string, easy to send around

- Traces for managing changes
- Platform independence
- Deployment, via Tclkit

Together, these features combine to create a software development environment which is extraordinarily well-suited for networked visually rich collaborating applications, in fact. It takes perhaps a dozen lines of code to implement a basic server, and about the same amount of code again to let clients communicate with it.

The total code needed to implement Tequila is under a thousand lines of pure Tcl. No C-coded extensions are needed (beyond what Tclkit includes) to create a system which handles the user interface, networking, and storage of data. The performance has turned out to be excellent, no doubt due to the fact that the level at which network activity takes place is very high.

It should also be mentioned that Starkits, the deployment technology which is now so common with Tcl/Tk 8.4 and particularly the Tclkit self-contained application runtime for Tcl/Tk, have made deployment of Tcl applications trivial. In the case of collaborating applications, this is an essential ingredient for success.

Early case study

The original idea for Tequila came from a commercial project implemented in the course of 2000 and 2001. The system is used to continuously test the properties of a telephone network by making large numbers of test calls and collecting the results. Some details:

- Long-lived “dumb” central server
- Central process driving modem banks
- A scheduler drives the main workload
- Custom requests are piggy-backed on top
- All data stored centrally
- Interactive clients on Windows and Solaris
- Simple login/password authentication
- Client “loads itself” over the net on startup
- Watchdog process restarts server if it fails

It was decided early on that the networking and storage design would be developed separately, and tested with an elaborate simulation harness. This became Tequila version 1, although the core was in fact re-coded twice from scratch, as more experience was gained.

This system was extremely successful:

- Dimensioned to handle 250 dialers on an Ultra-SPARC, the system was in fact able to handle 500 dialers in continuous simulation mode (which placed a far greater load on the system) from a measly laptop running Linux.
- As part of a very extensive pilot test, the server has

stayed “up” for 18 months without a single failure, making roughly 250,000 test calls daily.

- Software changes do not require a server restart, the scheduler and modem-bank processes can both be stopped and restarted in mid-flight.
- Client deployment consisted of a single Tclkit executable plus a Starkit (the Tclkit + Starkit deployment model were used in a production setting for the first time, although it was still called a “scripted document” at that time).
- The core system became smaller and simpler as development progressed, a clear sign that the design concepts used were working out properly.
- Development and testing, and eventually also delivery of the final product took place over three continents. ISDN-based client access turned out to be workable, even though the system was only designed for LAN use.

One interesting observation is that all of the above was achieved with a beta version of Tcl/Tk (8.2b2, I think). So much for the notion that “beta” must be buggy!

Tequila T1

The first implementation will be called “T1” from now on, to distinguish from the newer T2 design presented later on in this paper.

T1 is based on a really simple API: arrays.

The idea is that client applications “attach” one or more global arrays to Tequila, and then automatically any changes made to them anywhere will propagate to all other clients via background network communication.

This is in fact exactly what T1 does. It relies very heavily on Tcl traces to detect changes to any attached array, and then takes care of sending and receiving such updates. Traces are then also used by the application to trigger processing and user interface updates when any item in an array changes due to incoming network messages. These changes can include addition and deletion of array elements, not just modification of existing ones.

Persistent storage comes for free. This rather surprising benefit comes from the fact that all data resides on the server, so as long as a server stays up, nothing needs to be done on the client side to store information. Every time a client starts up, it receives the relevant state from the server. In a way, the server defines a permanent workspace, and clients simply get to see more or less of that state as they attach their arrays.

However, given that a server could fail, such an “all in-memory” approach as described approach is evidently a bit too risky to build serious applications on. For that reason, T1 includes server-side logic to store array contents on file: either as views in a Metakit database or as individual files in a directory. The

latter can be convenient to see what the state is while a server is running, since all state of such arrays can easily be inspected from the command line.

There is a special array called “tequila” to which clients can attach. It contains one entry for each currently connected client. This allows clients to discover which other clients are running, and to detect their demise by setting an unset trace on the tequila array.

A simple lock mechanism was added later on by Steve Landers, to allow clients to synchronize their actions. This allows a client to claim a data item so that others will not alter it while the item is displayed and being edited, for example. All locking is advisory: all parties must follow the proper convention, and refrain from altering locked items.

Experience with T1

One of the most surprising outcomes of T1, was to see just how well it fits into Tcl. Applications need to be written in a certain style, but apart from this nothing changes between stand-alone applications that do not use Tequila and applications that do. The only thing that changes is on startup: decide which arrays are shared, and attach them as part of the initialization step.

The fact that arrays “simply persist” is a benefit that is hard to over-estimate. Coding can now deal with the structure of data and the effect of changes on that data, with no attention at all to loading or saving things. This works, because all previous state changes sort of get applied automatically on startup, so all an application has to do is set up the user interface, prepare the traces that act on changes, and then attach the array. As soon as the connection is established, the network will send all state, adjusting the user interface through the traces.

Performance is surprisingly good. It turns out that Tcl, even though scripted, has absolutely no trouble keeping up with network activity and traces over a LAN. Due to the asynchronous design, many delays drop beneath the horizon. This is even the case for locally generated changes, which normally also need to make a round trip to the server to be applied consistently. The trick to achieve good performance is to design in such a way that latencies have limited impact. Bandwidth can be more or less ignored (within reason), since the level of communication is very high, i.e. the rate of messages and their size often remains quite limited.

A design decision which has paid itself back in gold, is the choice of using a generic Tequila server, and keeping all application-specific business logic isolated in clients and secondary “worker processes” – clients which run on the server to perform tasks, but with no user interface. The generic server has allowed creating a system whereby the centerpiece is rock-solid from the very start, and requires almost no debugging as the rest of the application is imple-

mented and extended.

Not all is peachy with T1, however. One unforeseen issue has been that distributed code and especially traces are a nightmare to debug, even with ample logging in the server. The timing variations make it next to impossible to reproduce problems related to order-of-events, i.e. race conditions.

Another problem with T1 is that it does not scale that far when arrays contain a lot of data. The reason for this is that all data must be sent across to each client as the array is attached, leading to potentially long startup delays. Furthermore, being arrays, all data in the client ends up being memory resident. These issues can be alleviated up to a point by carefully splitting data across multiple arrays, and only attaching a subset. In T1, an extra escape hatch was built into Tequila, allowing a procedural get/set access mode for data, i.e. foregoing attached arrays for some cases, such as activity logs.

And lastly, Tequila grew out of a single project, so its features were tuned somewhat to that. One limitation is that the transport layer was built-in as being plain sockets only, using dedicated ports. Other uses such as an HTTP wrapper via port 80 were considered but never implemented. Neither was a more secure session layer, such as SSL.

Subsequent uses in other projects have shown that although Tequila provides a very effective model for collaborating software, it does have limited uses. Sometimes, a more traditional RPC-like mechanism can make things simpler still, rather than having to shoehorn the code into changes to attached arrays.

Beyond T1

In January 2005, a new project called “Stargus” presented a chance to revisit the whole design and structure of Tequila, with the opportunity to take things a bit further.

The following aspects were investigated:

- Improved data-model support
- Far better scalability
- Support for authentication & encryption
- Client-side caching and re-connects

Many of these capabilities are found in the GroupKit package, which was found too elaborate and complex for T1 in 2000/2001, but which does offer a very rich set of concepts and implementation solutions for most of the above aspects. The one thing lacking in GroupKit is persistence, and as T1 has shown that really is a very useful and effective part of Tequila.

As a result, and drawing on the experience of Mark Roseman, GroupKit’s principal architect, it was decided to re-implement Tequila, in a design much closer to GroupKit’s, while adopting as much as possible from T1. The result is Tequila T2, described here.

Tequila T2

T2 is a complete rewrite. It was written in a very modular fashion, so that individual parts can be extended or even replaced later on, as needed.

This is one of the situations where an object system really shines. It is trivial to extend a system when its main components are object oriented, regardless of whether the OO mindset is used in the rest of the application. For practical reasons, IncrTcl was selected for T2 – with the idea that any OO system could be substituted later, if the Tcl community ever gets its act together in designating one OO system as “the one”.

IncrTcl is part of every Tclkit binary, the deployment system of choice for applications such as these. It is stable, documented, and fast.

For persistence, the Metakit database library is used. There are a number of reasons why this is a good fit:

- On-the-fly restructuring (adding properties)
- Compact both on file, and in-memory
- Easy to send serialized snapshot over a socket
- Included in Tclkit, stable

Neither IncrTcl nor Metakit are essential ingredients for Tequila, they could be replaced when other packages are available which are more suitable in some sense.

A side-effect of using components which are present in every build of Tclkit, is that Tequila can be used as is on every platform for which there is a standalone Tclkit executable, which is several dozen by now. Since ActiveTcl also contains all the necessary packages, that too is an option.

Anatomy of T2

Tequila T2 consists of the following key components:

- Pools, as the logical building blocks of data
- RPC endpoints, used for all communication
- Notifiers, a generalized version of traces

Each of these will be covered in detail in the next sections. Together these form a package called, unsurprisingly, “tequila” which is a single Tcl-only script that needs to be included in each collaborative application, both clients and servers. There is a simple generic server which can be used out of the box for simple cases, or can be used as template and starting point for a more sophisticated system. And lastly, there are a number of examples to illustrate various aspects of T2.

At this stage, T2 is fully functional, but it has not been used “in anger”: there are no large-scale production systems based on T2 so far (April 2005).

Pools

Pools are the central concept on which applications get built. A pool contains one or more named collections of data, which automatically persist and get shared with all clients connecting to the same pool (via the server – T2 only addresses centralized client/server topologies at this stage).

A “collection” in turn, is a tabular data structure: it has rows, identified by a key, and named attributes. You can think of a collection as a table. Simple tables might contain just a key and a value, in which case they are very much like Tcl arrays, or they might contain more attributes. All values in a collection are strings and can – as usual in Tcl – contain anything.

Pools represent global state. In MVC (model, view, controller) parlance, pools are the model. Pools will normally persist. The structuring of pools into collections is a way to bring different data structures together – each collection can have a different set of attributes, whereas all rows within a collection have the same set of attributes.

In the simplest case, you only need a single pool, so the basic design effort is all about deciding what collections you need, and what the key and attributes of each should be.

Here’s how to create a new standalone pool:

```
tequila::pool mypool
```

This produces a new command, named “mypool”. From there, it is easy to create a collection and add a row to it:

```
mypool set mycoll.x value y
```

This creates the collection “mycoll”, and adds a row with key “x” and an attribute named “value” set to “y”. Fetching the row again uses the “get” subcommand:

```
puts [mypool get mycoll.x]
```

You can also find out the keys of all rows in the collection:

```
puts [mypool keys mycoll]
```

Or any other property for that matter:

```
puts [mypool values value]
```

One useful subcommand for debugging is “dump”, it prints the first few rows of every collection in a pool.

There are several more subcommands. The idea is that collections are like an “array with named columns”, with “rows” as entries, and that these arrays, rows, and attributes are created on-the-fly when a value is set.

As described so far, pools are merely local. To make them shared we need to tie them to “endpoints”.

RPC endpoints

An “RPC endpoint” is perhaps best described as “this half of a connection”. It is the entrance of the tunnel leading to a remote counterpart, i.e. for a client it is the link to a server.

The following discussion is geared towards sockets, although other types of endpoints could be created. The startup sequence is bound to be different, but - once set up - endpoints are simply objects which must respond to a certain API.

Endpoints are asymmetric (just as sockets are). To establish a connection, a server somewhere must create an endpoint and specify what port number to use:

```
set s [tequila::endpoint -server 8291]
```

The above creates a server socket, listening on port 8291. Once that is available, clients can connect to that endpoint using something like:

```
set c [tequila::endpoint \  
server.domain.net 8291]
```

Once a session has been established, we can tie a pool to them. This is done when the pool is created, so that its previous state can be restored right away. Here’s an example where “mypool” is associated with a pool of the same name on the server:

```
tequila::pool mypool $c
```

Keep in mind that all network events are asynchronous and processed in the background, i.e. at idle time. The above does not instantly fill the pool, it just ties things together so that everything happens later on, when the application setup has been completed and it starts waiting for events.

Notifiers

As it stands, the above is sufficient to make pools stay in sync across multiple clients. The one missing link is that an application wouldn’t find out about any of these changes, so it’d be rather boring.

That is where notifiers come in. A notifier is an object where apps can subscribe to be notified of events. The prime user of this mechanism is the pool – every pool includes a notifier, so that changes to the pool can be associated with application scripts to execute.

The notification mechanism is based on named “events”. Applications can “bind” to an event and cause it to execute a certain script, or they can bind to a range of events by using “*” and “[...]” and “?” wildcards.

Pools define various events, such as:

- connected – the pool has been initialized
- disconnected – connection has been lost
- collection.add / .change / .delete – a row has been added/changed/deleted

- collection.attribute – a change to specified attribute

The scripts that get evaluated when an event fires can obtain additional information about the event through a mechanism that is identical to Tk: event specifiers, i.e. codes in the script starting with “%” – these get replaced before the script is actually evaluated. Event specifiers are essentially a general way of passing arguments from an event producer to all event consumers.

In the case of pools, the following event specifiers are available:

%C – name of the collection

%K – key value

%E – full event name

%R – row number

Example of use:

```
mypool bind *.add \  
{ puts “added key %K to coll %C” }
```

Not all specifiers are meaningful in all types of events (%C, %K, and %R are not set in connect/disconnect events, for example). Some events may define more specifiers – see the documentation for details.

Keep in mind that notifiers are not limited to being used by pools. The basic rule is just that event producers and event consumers must agree on consistent conventions.

Putting the pieces together

In the basic scenario, the following steps must be taken:

- A server must be started, on a well-known host and port. The server usually stays running for a long time, and runs unattended.
- Each client starts by creating a client-side endpoint connecting to that server.
- Next, clients set up one or more pools, and tie them to this endpoint. The convention is for a server to have a “system” pool, and in it a collection called “tequila”.
- For simple uses, clients can set up their end of the “system” pool and create the collections they need inside that pool.
- If multiple pools are required, then the server must either set up all pools, or be prepared to create such pools as needed when clients ask for them.
- Once pools exist, clients proceed by creating a set of bindings that determine what happens on changes to collection(s) in the pool(s).
- Lastly, each client enters the idle loop, starting the phase where connections will become active and messages get sent around between the Tequila clients and the Tequila server.

From here on, the client application becomes an event-driven mechanism, dealing with user interface events as usual (with Tk) and dealing with network events to manage the sharing of state and the propagation of changes to that shared state.

An RPC example: chat

One of the simplest networked examples is a chat – people type lines into an input area, and everyone gets to see the lines when RETURN is pressed.

The code for this example is in Appendix A. It illustrates the absolute basics of Tequila:

- Setting up the Tequila package
- Connecting to a common server
- Sending lines to the server
- Responding to incoming lines

Although the code is very simple to read and may help to get started, it also is seriously flawed: the design used in this chat application is in fact precisely the wrong approach most collaborative applications...

The reason is that the chat works in terms of actions (i.e. send this line to everyone). This breaks down when you start thinking about clients connecting and disconnecting at various times. When that happens, you often want to be brought up to date to see the same state as what others see. This brings a sense of “being” in one place, and collaborating on a common project.

With purely event-driven exchanges, you’d have to think in terms of replaying changes to clients who join later. But as we shall see, there is a much easier way to accomplish the same thing.

Model-View-Controller

From Smalltalk comes the concept of splitting the work in an application into three distinct tasks:

- Model = the state, think “documents”, “projects”, and so on.
- View = what you see on the screen (or the browser, or even reports on file or paper). There can be more than one view on the model. There can also be a model without a view – i.e. when you are not displaying it.
- Controller = the way to deal with input actions, from point the mouse, to scrolling, to entering data and pushing buttons.

With collaborating applications in general and Tequila in particular, these aspects of a design come back in full force. One reason being that there will be potentially many views, and many controllers. But in general, what you want is that everyone works with a consistent copy of the same common model.

First the bad news: there are limits to that consistency. There is no way one can automatically resolve

the situation when people perform conflict actions at exactly the same moment in time. It’s a bit like Heisenberg’s uncertainty principle: either two people are next to each other and acting in sync, or they are remote and have to accept imprecise knowledge of what the other is doing at that very same moment in time.

The good news is that this need not be a problem. The simplest solution is to serialize actions via one common server. The server will, by the fact that it processes work sequentially, serialize incoming requests in order of arrival.

This does not prevent unintended clashes. It is as with long-distance phone calls: when delays are large (as in satellite-based sessions), you occasionally end up speaking at the same time as the other party. Right thereafter, you both realize that this won’t work, stop, and retry after a short while. Normal courtesy and occasional extra retries take care of the rest.

A similar mechanism is used in Tequila. Actions are transmitted to the server, which then broadcasts it to all clients – including the originating one. The result is that all clients receive all actions in the same order, and can maintain a 100% consistent state. In the case of accidental conflicting actions, one of them will end up preceding the other, all parties will see the result in the same way, and once they realize it they can react by undoing or repeating their action. That too will be sent to the server, and all clients see the corrective actions.

By sending all requests to a central server, a single consistent model is maintained. Clients can then deal with views by themselves, and never worry about synchronization. The worst that can happen are network delays, which is unnoticeable in the case of geographically remote clients (even a few seconds would usually go undetected!).

The controller side of all clients is synchronized by the fact that they cause no local effect but merely get sent to the server to wait in line and be accepted.

There are many issues related to synchronization that are beyond the scope of this paper. Suffice to say that when the user interaction is designed with potential latency in mind, the whole server-based approach works remarkably well. No nasty race conditions, no tricky recovery mechanisms are needed.

It’s all about the model

The key to designing collaborating applications with Tequila is to focus almost exclusively on shared state, and to avoid “telling everyone what to do”.

This is why the above chat example was in fact not such a good idea. If coded naïvely, the chat would not even present a consistent log to every client. This would be the case if clients decided to add their own entries to their log and only broadcast the entries to all the other clients.

In the code in appendix A, the chat avoids this problem. It uses the server to serialize, by sending a line to the server, which then broadcasts it to all clients, including the originating one. Once that request comes back in, the originating client adds it to the log window. Just as all other clients do, and in precisely the same order.

What the chat does is use the central server to serialize all events. Each client then maintains its own copy of the model. Note that the server does not do anything with the requests, other than sending them around.

In other words, the chat relies on all clients to diligently follow all events and each manages a replica of the data model. This breaks down for clients who join later (or drop off and have to reconnect for any reason).

That's where Tequila's "pools" come in. They are the model and reside on the server. When a client connects to a pool, it first gets a copy of the exact state on the server. After that, it receives all changes to the pool, using the same broadcast mechanism as before.

Pools solve the problem of connecting at arbitrary points in time. Due to their shared design, pools also manage all further changes – using RPC, but taking care of everything transparently. With pools, you are encouraged to think in terms of state, not actions!

In a way, pools are Tequila's way of using a model. Distance and connection re-establishment are solved.

A better example: rooms

Appendix B contains the code for a more elaborate example, written by Mark Roseman. It represents a groupware system, whereby clients connect to a server with "rooms". Each room has some state, each client can view that state as well as modify it. Any client can also create rooms themselves – the system starts out with no rooms at all.

The model here is very simple and clear: each room is a pool. There is a special pool called "system", which is where clients can find a list with the names of all rooms. A simple explorer-style GUI lets you add to that list, or select an existing room from it.

When you select a room by double-clicking, the right-hand pane shows the room. Rooms have a random color assigned to them at creation time (just so it's easy to recognize different rooms). Clicking on a room adds a timestamp in the position that was clicked.

Needless to say, every client can do this, and every client sees changes in real time.

This example is illustrative of how designs with Tequila work:

- Each room is a pool – clients only deal with one room at the time. Changes to others don't concern them (and are not sent).

- The design is completely in terms of state – there are no explicit RPC calls in the rooms demo. The code works the same regardless of which and how many other clients are around.
- Connects and re-connects are easy – each time a room is selected, the client gets the current state (i.e. a copy of its pool on the server). After that, it automatically sees all changes.

The transparency of the network should become clear when you realize that an application can be developed standalone, by using local pools. All it takes to turn this into a networked system later on is to alter the way pools are initialized (i.e. associated with a server).

Current status

Tequila is currently called T2 (it's at version 2.02 at the time of this writing). That reflects the fact that it is a second-generation implementation, taking the best from the original T1 as well as from GroupKit.

T2 has not been used in a production setting. It has passed a certain amount of testing, but there are still some design changes underway – such as being able to use multiple pools on a server via single RPC channel.

The basic concepts of pools, RPC endpoints, and notifiers are stable, however. This design is expected to stay essentially as is from now on.

Performance appears to be good, judging by a few timing experiments. All the code is in pure Tcl, so there will be considerable room for improvement if bottlenecks do show up at some point. In GroupKit, "environments" (a more elaborate concept on which Tequila's pools are based) were coded in C, and so were notifiers, but so far Tcl has worked out well.

The design of collections (i.e. the tabular data structures which can be stored in pools) is relatively complete, but a few loose ends remain with respect to positional access and inserting/moving/deleting rows.

Next steps – T3

T2 is really an intermediate phase to transition away from T1's array-centric design and prepare for a number of more advanced features:

- Scalability – by using collections, the need to keep all data in memory (as with arrays) is gone. T2 introduces pools and collections, and maps them to Metakit sub-views. This is a start to maintaining data in memory-mapped files, i.e. letting the operating system do what it does so well: decide on when/how to page data in and out of RAM, transparently.
- Client-side caching – a major limitation of T1 and T2 is that all data in an array/pool is sent to the client when it connects. This does not scale well. T1 had some hooks to avoid the copying, the price

being increased application complexity. T2 was designed to eventually support a cache, so that the client side only need fetch changes – often a fraction of the complete dataset.

- Transport independence – T2’s RPC objects make it possible to insert extra layers or even replace them with something entirely different. This can be used to add compression and/or encryption to the communication channel between clients and servers
- Multiple servers – T2 makes a start with supporting multiple servers. These can be used for redundant/fail-safe scenarios, as well as for maintaining pools in separate locations (for security, or intranet vs. public).
- Database gateways – the collections in T2 support a rich table-like structure, whereas T1 only supports key-value associative arrays. This can be used to create rich gateways to relational databases via a Tequila server.
- Tools – with a generic framework such as Tequila, it is worthwhile to create generic tools that can be used by different applications, both during development and in released code. Think of auditing, debugging/tracing, web interfaces, and report-generation.

The plans for T3 have only just started. A number of the issues mentioned above will be addressed, but since Tequila is 100% open source, there is nothing to prevent anyone from taking what they like and extending it in ways they need. Preferably in such a way that others can benefit as well, of course.

Looking further

Tequila is an exciting mix of ideas and trade-offs. On the one hand, it completely bypasses conventional approaches such as the use of N-tier client/server databases and the use of HTTP and XML solutions. Instead, Tcl’s strengths are exploited in several ways:

- Using Tcl as protocol over-the-wire
- Using asynchronous file events for all I/O
- Using notifiers, as an equivalent for traces
- Using Metakit for data storage
- Using Telkit and Starkits for deployment

And of course: having Tk around the corner for building user interfaces on top of all this.

For people who know Tcl, the above are probably not very exciting, but it’s hard to overestimate the importance of having all these capabilities come together in the way Tequila uses them. Not to mention that T2 is well under 1000 lines of code so far – offering proof that all the components really are working together to form a powerful way of developing collaborating applications.

But the promise of T2 is in fact a different one: if

you ignore all the details of storage and networking and GUIs (as you can with Tequila + Tk), then what remains is the way pools and collections become the (structured!) “data backplane” of an application, regardless whether it is a single standalone script or a large multi-process / multi-site distributed system. What remains, is the application’s core itself, and the business logic that really defines what is being done.

With T3, the hope is that this potential to simplify software development with Tcl will be exposed and exploited further, much further eventually. The change in mindset needed to accomplish this, is that everything is about state and consequences of state change. As Tequila proves, location, persistence, sharing, and application launches & exits can become details that will need much less attention than before.

Conclusion

When started in 2000, Tequila immediately proved itself by dramatically simplifying the application it was initially designed for. By dealing with the general issues in a generic way, a surprisingly effective separation of (complex) application logic as well as (complex) collaboration logic was achieved. As a result, Tequila (T1) has been used in a number of projects since, with considerable productivity gains – it helped get software projects out the door faster, and it came with a solid set of features, tested and ready to build on, and with.

The current T2 re-implementation has been completed. It introduces pools, RPC endpoints, and notifiers. It also introduces collections as a more general way of managing structured data. The API is starting to look good, in that it remains relatively simple yet offers all the benefits that T1 had – as well as opening up the path to more advanced features such as client-side caching.

T3, the next phase of this project has only just started. It is likely to leave much of what T2 does alone, and simply extend on what there is to implement additional functionality.

Acknowledgments

The design of T2 was shaped profoundly by discussions with Mark Roseman, GroupKit’s principal author. As a result, it appears to be simpler (and leaner) as well as more flexible. The holy grail of collaborating software is probably the idea of tying networking, persistence, and the graphical user interface together in as automatic a framework as possible. If T2 succeeds, it will be mostly due to Mark’s involvement and experience (and if it fails, yours truly did 95% of the coding, so you’ll know who to blame...).

I would also like to thank Steve Landers, for participating in many of T2’s discussions, and for always coming up with good ideas and questions.

The original work on Tequila would not have been

possible without Steve Landers and Cameron Laird, who worked on the project where Tequila was originally born. Between us, we came up with the whole idea and together we carried it through to make it a pretty effective product. I'd like to thank Larry Blasingame for making some of the toughest go-ahead decisions ever by placing so much trust in a bunch of excited software engineers, halfway across the globe.

And last but not least, I'd like to thank Mike Doyle and Eolas Technologies Inc, for funding my work done on T2 and for providing an exciting opportunity to take Tequila further for everyone, by supporting its public release as open source software.

References

Tequila home – <http://www.equi4.com/tequila.html>

The Tcl'ers Wiki - a collaborative web site for the Tcl community, <http://wiki.tcl.tk/>

GroupKit by Mark Roseman, this was part of the GroupLab project at the University of Calgary – now released independently, <http://www.groupkit.org/>

IncrTcl – Object oriented extension for Tcl, <http://incrtcl.sourceforge.net/>

Metakit – embedded database extension for Tcl (Mk4tcl), <http://www.equi4.com/metakit.html>

Tclkit – a standalone runtime for Tcl/Tk, includes IncrTcl and Metakit, <http://www.equi4.com/tclkit.html>

Appendix A – chat demo

```
# Tiny chat system: assumes generic tequila server is running (port 18396)
# Clients broadcast their msgs to everyone currently connected to the server.

package require Tk
package require tequila 2.02

set rpc [tequila::rpc localhost 18396 -command chatrecv]

grid [text .t -width 40 -height 10 -yscrollcommand ".s set"] \
     -column 0 -row 0 -sticky news
grid [scrollbar .s -command ".t yview"] -column 1 -row 0 -sticky ns
grid [entry .e] -column 0 -row 1 -columnspan 2 -sticky we
bind .e <Return> sendChat

proc sendChat {} {
    $::rpc send to all chatmsg [.e get]
    .e delete 0 end
}

proc chatrecv {conn data} {
    if {[lindex $data 0] eq "chatmsg"} {
        eval $data
    }
}

proc chatmsg {msg} {
    .t insert end $msg\n
    .t see end
}
```

Appendix B – rooms demo

client.tcl:

```
package require Tk
package require tequila

set rpc [tequila::rpc localhost 18396]
tequila::pool system $rpc

frame .rooms
button .rooms.new -text New -command newRoom
listbox .rooms.l -yscrollcommand ".rooms.sb set"
scrollbar .rooms.sb -command ".rooms.l yview"
frame .room
canvas .room.c -width 400 -height 400 -background #ccc
grid .rooms -column 0 -row 0 -sticky ns
grid .rooms.new -column 0 -row 0
grid .rooms.l -column 0 -row 1 -sticky news
grid .rooms.sb -column 1 -row 1 -sticky ns
bind .rooms.l <Double-1> enterRoom
grid .room -column 1 -row 0 -sticky news
grid .room.c -column 0 -row 0 -sticky news
grid columnconfigure . 1 -weight 1
grid rowconfigure . 0 -weight 1
grid rowconfigure .rooms 1 -weight 1
grid columnconfigure .room 0 -weight 1
grid rowconfigure .room 0 -weight 1
wm title . "<No Room>"

proc nextid {} {
    if ![info exists ::nextid] { set ::nextid 0 }
    expr {[system cget -clientid]+1}*1000000 + [incr ::nextid]}
}

system bind rooms.* roomsChanged
system bind connected roomsChanged

proc newRoom {} {
    set roomid [nextid]
    $::rpc send poolmanager create room$roomid
    system set rooms.$roomid name "Room $roomid" pool ::room$roomid \
        backgroundcolor [format "%02x%02x%02x" [expr int(rand()*256.0)] \
            [expr int(rand()*256.0)] [expr int(rand()*256.0)]]
}

proc roomsChanged {} {
    .rooms.l delete 0 end
    foreach i [system keys rooms] {
        .rooms.l insert end [system get rooms.$i name]
    }
}

proc enterRoom {} {
    set idx [.rooms.l curselection]
    set roomid [system get rooms!$idx key]
    if {[info exists ::currentroomid] && $roomid == $::currentroomid} return
    if {[info exists ::room]} {
        .room.c delete all
        .room.c configure -background #ccc
        bind .room.c <Double-1> {}
        $::room destroy
    }
    set ::currentroomid $roomid
    set ::room [tequila::pool room$roomid $::rpc \
        -servername [system get rooms.$roomid pool]]
}
```

```

    $::room bind connected roomEntered
}

proc roomEntered {} {
    wm title . [system get rooms.$::currentroomid name]
    set color [system get rooms.$::currentroomid backgroundcolor]
    .room.c configure -background $color
    bind .room.c <Double-1> { clickWorkArea %x %y }
    $::room bind notes.* { noteChanged %K }
    foreach i [$::room keys notes] { noteChanged $i }
}

proc clickWorkArea {x y} {
    $::room set notes.[nextid] x $x y $y text [clock format [clock seconds]]
}

proc noteChanged {key} {
    if {$key=""} return
    foreach {x y text} [$::room get notes.$key x y text] break
    if {[.room.c find withtag note$key] eq ""} {
        .room.c create text $x $y -text $text -tags note$key
    } else {
        .room.c coords note$key $x $y
        .room.c itemconfigure note$key -text $text
    }
}
}

```

server.tcl:

```

package require tequila

class poolMgr {
    variable rpc
    method constructor {rpc_} {
        set rpc $rpc_
    }
    method create {name} {
        uplevel #0 ::tequila::pool -server $name $rpc
    }
    method connect {name clientname} {
        # later use this instead of connectPool
    }
}

# mainline
set rpc [tequila::rpc -server 18396]
poolMgr poolmanager $rpc
[$rpc cget -receiver] allow poolmanager
tequila::pool -server ::tequila::system $rpc -file try.db
[$rpc cget -receiver] allow tequila::connectpool
catch { vwait forever }

```