An overview of two key Stargus technologies - secure authentication and shared storage

Steve Huntley, Steve Landers, Mike Doyle, Cyndy Lilagan

The Buonacorsi Foundation
Wheaton, Illinois

ABSTRACT

Stargus is an ongoing project to develop a compact, portable and scaleable virtual computing environment, based on the Tcl/Tk platform. This paper introduces two key Stargus technologies aimed at supporting Cloud Computing: secure authentication via BetterID and shared storage via Mockingbird. These technologies are aimed at achieving scaleability and security by leveraging Cloud Computing while retaining the flexibility and productivity of the Tcl language and associated technologies.

1 Introduction

The Stargus project has been running since 2005, and was first introduced publicly at the Tcl2005 conference in Portland. The name Stargus is a combination of Starkits (**ST**and **A**lone **R**untimes) [1] and Argus (a mythological being with many eyes, some of which were always open, known as a zealous watchman; a very watchful person; a vigilant guardian).

The goal of Stargus is to make it easier for domain specialists to develop secure, reliable and scaleable applications without being tied to individual computing platforms or needing specialist software development skills. This will be achieved by abstraction of various aspects of the application development cycle, including data storage, user interfaces, security and communications.

Several Stargus sub-projects have been sponsored, including Cryptkit[2], Tequila T2 [3], Vlerq[4], Tkhtml3 [5], Politcl[6] and Critcl 2 [7].

This paper describes the design of two key Stargus technologies aimed at leveraging Cloud Computing: secure authentication via BetterID and shared storage via Mockingbird.

2 Background

Cloud Computing has gained much attention within the IT world over the past few years. It has been described as "a style of computing in which dynamically scalable and often virtualized resources are provided as a service over the Internet." [8] Usually Cloud Computing

infrastructure is accessed via web services interfaces (hence the "cloud" so often used in Internet diagrams).

A typical characteristic of Cloud Computing is shared access to enterprise-level computing infrastructure on a "pay for use" basis, meaning there are no up-front expenses nor long-term commitments. This makes it particularly attractive during the start-up phase of a project, when resource requirements have yet to be identified. This approach is also attractive for projects that require infrequent use of significant resources (so called "burst" applications).

Not all the attention on Cloud Computing has been positive (e.g. Cory Doctorow's anti-cloud rant in the Guardian [9]) since it has, like many new technologies, attracted the attention of marketeers and profiteers. But as Jon Stokes noted in an Ars Technica article[10], Cloud Computing does offer new opportunities -- in particular, allowing smaller organizations to access big organization IT infrastructure and its capabilities.

There are numerous Cloud Computing vendors, both for general services (e.g. Google App Engine[11] or Microsoft Azure [12]) and for specific service providers (e.g. Salesforce.com [13]). But perhaps one of the most visible examples is Amazon's AWS (Amazon Web Services) [14] which provides access to Amazon's global computing infrastructure, and which is the focus of the initial Stargus Cloud Computing support.

Amazon's AWS comprises several products, the key ones being:

- EC2 (Elastic Compute Cloud) -- resizable compute capacity
- SDB (SimpleDB) -- indexed storage and querying of multiple data sets
- S3 (Simple Storage Service) -- storage and retrieval of data "buckets" or "blobs" (from 1 byte to 5Gb in size)
- SQS (Simple Queue Service) -- a reliable, highly scalable, hosted queuing service for storing messages as they travel between computers.

What benefits does AWS provide for Stargus? There is no need to install and administer a database, it provides reliability, durability and availability, and it scales without tuning.

However, AWS is accessed via web services APIs and therefore isn't very accessible to Tcl developers familiar with abstracted higher-level interfaces provided by packages like TDBC [15], SQLite [16] and Tequila[17].

In addition, there are security and authentication issues that are exacerbated in a Cloud Computing environment, where there should be no single point of compromise or failure. Also the ability to rapidly add and remove servers makes centralized management of users a potentially onerous task.

For these reasons, the BetterID and Mockingbird projects were created to address the Security/Authentication and Storage needs of Stargus applications within a Cloud Computing environment.

3 BetterID

Virtually all multi-user computing environments, from dumb login terminals to globally distributed supercluster nets, face the questions of how to permit access to resources, and how to apportion privileges. Regrettably for the evolution of computer architecture, these questions are most often not addressed thoroughly, if at all, until quite late in the design and deployment cycle. Persistent issues of abuse of access and privileges that have been enabled by neglect of these questions has retarded realization and scaling of the full collaborative potential of the Internet.

In order to avoid yet another cul-de-sac in the design of a distributed computing system, the Stargus project is incorporating a secure authentication and privileging scheme into its architecture from the outset. Our new authentication protocol, called BetterID, was partially inspired by the deficiencies of existing popular protocols (e.g. OpenID). It makes use of PoliTcl, the Stargus policy-based privilege-granting protocol which was introduced at the 15th Annual Tcl/Tk Conference in 2008. PoliTcl was based upon the Tcl Plugin security system, adapting it for use outside of a plugin context, and adding cryptographic security capabilities.

BetterID is an implementation-agnostic authentication protocol designed to be adaptable to custom clients, or to existing popular clients such as Javascript-enabled web browsers, as required. Thus it can conceivably be used for authentication in contexts outside the Stargus architecture.

The BetterID protocol uses the well-tested technology of RSA cryptographic key-pairs for digital signing and encryption of authentication requests. It aims to provide an innovative solution to the persistent problem of key management by storing all the public and private keys of account-holders in a simple highly-available static file web server, such as that provided by Amazon's Simple Storage Service.

Similar to OpenID, an account-holder's identity is mapped to a URL. But in the BetterID protocol, something meaningful is stored at the individual's assigned URL: an RSA public key. This public key is globally readable, so a digital signature or encrypted message signed with the corresponding private key can be authenticated/decrypted by anyone. Thus the individual's identity as the holder of the private key corresponding to the specified public key can be verified.

In order to make key management as reliable as possible, the individual's private key is stored on the same static file web server, in strongly-encrypted form. Thus one or several clients can be easily “initialized” (via download, decryption and installation of the private key) by a mobile user as desired, needing only an internet connection and a generic unregistered client program.

Specific additional safeguards are used in the secure authentication process to defend against malicious attacks such as phishing and DNS poisoning.

The motivating aim of BetterID is to make it easy for a participant in the Stargus net to sit at any computer, install and configure a generic client program, and, from that point, log in to the Stargus environment and start requesting services from it. Since the services available from Stargus are anticipated to be many and finely-defined, a BetterID client will be required to

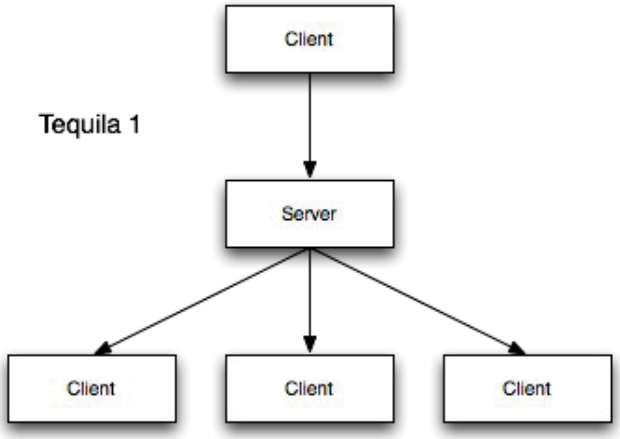
access Stargus via a sophisticated programmable grantor of privileges. The policy-driven server, based on the PoliTcl system, will be the privilege-granting gatekeeper to the Stargus environment.

4 Shared Storage

4.1 Tequila 1

In the early stages of the Stargus project, the shared storage was provided by the Tequila package (a.k.a. Tequila 1 or T1). Tequila (written by Jean-Claude Wippler) implements persistent shared arrays, using Tcl traces and socket connections to give the illusion of data sharing between multiple clients.

The Tequila client code operates by adding traces on attached arrays. When an array element is changed, the associated trace command sends an update notification to the Tequila server via a socket connection. The Tequila server (in turn) notifies other clients via their socket connections.



Development using Tequila is very natural to the Tcl programmer, who no longer thinks in terms of databases and communication protocols, but rather attaches arrays to a Tequila server and proceeds to read, write and unset values in the array.

```
tequila::open $server $port
...
tequila::attach name
...
set name(one) 1
puts $name(one)   ;# returns 1
```

Using this approach, it has proven possible to quickly take existing Tcl/Tk applications and make them distributed without the need for major changes.

The persistence in Tequila 1 is obtained (almost for free) by the use of the Metakit database[18] in the back-end server. Metakit is characterized by column-wise storage accessed via memory mapped files. This provides simplicity, performance and flexibility (for example, Metakit storage can be restructured on the fly).

Tequila 1 has proven to be extremely effective and has been used in a number of open source and commercial projects, including a distributed telephone network testing system (which 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.

Tequila's development model of shared persistent arrays has proven to be a comfortable fit for many Tcl developers. Yet, in spite of this, there are a number of issues with Tequila 1 that limit its applicability to larger applications. In his EuroTcl 2005 paper[19] Jean-Claude Wippler identified the main ones as:

security	Tequila uses sockets and has no built-in authentication
scaleability	large arrays take time to copy from the server on startup
debugging	trace-driven code can lead to non-deterministic results
data-model	the array model can be limiting, with each array corresponding to a single row In a database, rather than to a collection of rows

To address these issues (and others such as improved data structuring) a new Stargus project was started in 2005: Tequila T2.

4.2 Tequila 2

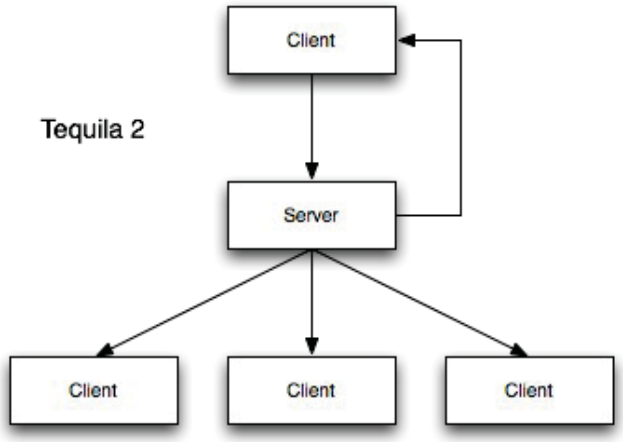
Tequila 2 (T2) is a complete, modular rewrite of Tequila, drawing on the experience gained in Tequila 1 and also on the expertise of Mark Roseman, the principle architect of GroupKit, a Tcl-based framework for developing collaboration/groupware applications [20].

The details of the T2 architecture are described in Wippler's EuroTcl 2005 paper, but in summary it provides the following components:

pools	shareable and persistent collections of data
collections	a tabular data structure with rows identified by a key, and named attributes
rpc endpoints	communication objects - one half of a connection to a remote counterpart (usually a server)
notifiers	a generalized trace facility that supports a call-back mechanism to trigger actions when an event occurs

T2 notifiers are analogous to traces, and provide the key differential between Tequila 1 and Tequila 2: the T1 API is synchronous, whereas the T2 API is asynchronous.

In the T2 model, the data change isn't reflected in the local client immediately, but rather propagates to the server, then out to other clients as well as back to the originating client, at which point the change is visible and the notifier facility can trigger actions in the client code.



T2 provides an Object-oriented style API. A value is assigned using a "set" method on a pool object, and retrieved using a "get" method. Client code might resemble:

```
set rpc [tequila::rpc $server $port]
...
tequila::pool mypool $rpc
...
mypool set name.one 1
puts [mypool get name.one]
```

Where 'name' is the name of the collection and 'one' is the attribute being set.

But in reality it is more complex. All the T2 operations are asynchronous, and setting a value involves a round trip to the server, so it is likely that the 'mypool get name.one' command won't return the expected value.

With T2, the programmer has to "think async," and use notifiers to wait until the value has been set in the pool:

```
set rpc [tequila::rpc $server $port]
...
tequila::pool mypool $rpc
mypool bind connected {set ::connected 1}
vwait ::connected

...
mypool set name.one 1
bind mypool.name.one {set ::changed 1}
vwait ::changed
puts [mypool get name.one]
```

Tequila 2 achieved its goals and was even used successfully in a commercial product, a distributed document management system. But it is an intermediate step to Tequila 3, which was intended to introduce more advanced features such as:

- scaleability (no need for all data to be held in memory)
- performance (through client side caching)
- transport independence (abstracted behind the RPC objects)
- reliability (through multiple servers)

Development of Tequila 3 has not yet reached beyond the planning stage, but its goals have been revisited here, in light of Cloud Computing developments over the past few years.

4.3 Tequila Redux

Tequila 1 provides the benefits of simplicity and familiarity, but at the expense of reliability, scaleability and security.

Tequila 2 laid the foundation for addressing these issues, but requires a new and different mindset unfamiliar to Tcl programmers. Adapting to this mindset has proven difficult, even for experienced Tcl programmers.

When designing the Stargus Shared Storage support for a Cloud Computing environment, emphasis was given to retaining the simplicity of the Tequila 1 API approach (shared, persistent arrays) for simple applications, while addressing the key issues of scaleability, reliability and security.

This isn't to preclude the development of Tequila 3, or adoption of the Tequila 2 programming model, but rather it extends the Tequila philosophy from the desktop to a Cloud Computing environment.

So, in the first instance, Mockingbird (a.k.a. Tequila Mockingbird) implements the shared persistent array model. But subsequently, an API nearer to the Tequila 2 API will be needed, for a number of reasons:

- to accommodate applications that require more scaleability
- to provide finer-grained control of when values are set and propagated
- to provide API commands for searching the data on the server rather than in the client
- to provide additional API commands such as callbacks for specific events
- to allow access from other languages that don't support traces (via a Mockingbird StarDLL [21])

That being said, even the first version will go well beyond the limitations of Tequila 1, given that Cloud Computing offers the potential to provide reliability and scaleability on the server side.

4.4 The Mockingbird API and Data Models

The Mockingbird API needs to be considered from two viewpoints:

1. the Tcl programming model, and
2. the database features being accessed in the cloud

Arguably, there is also the communications model to be considered, but, since that role is handled by Amazon AWS, it won't be considered,at least, initially.

We have already noted that the shared, persistent array approach will be used. But before addressing the Tcl programming model alternatives, it will be beneficial to consider the initial target back-end: Amazon's Simple DB[22].

Amazon's SimpleDB is (conceptually) a giant distributed hash table supporting key-value storage. In other words, it is a TupleSpace [23]. Like arrays, TupleSpaces are associative memory, so there is a natural fit,at least conceptually, between Tcl arrays and SDB. Keys and values are always strings (both UTF-8), and the primary operations are set, get and delete.

SDB also has a query operation that supports equality, inequality, prefix and boolean operators. So, while Mockingbird will use arrays as the abstraction for set, get and delete operations, it will use API methods for querying, and perhaps for asynchronous operations such as non-blocking reads. The SimpleDB data model is (as you would expect) simple -- domains are collections of items that are described by attribute-value pairs.

As well as Tequila 1 (circa 1999), there have been a number of other Tcl projects that implement persistent shared storage, including:

- Tuplespace[24] by Todd Coram (c 2002)
- Tgdbm[25] by Stefan Vogel (c 2003)
- Tie[26] by Andreas Kupries (c 2004)
- arraysync[27] by Steve Redler IV (c 2006)
- Tupleserver[28] by Mark Janssen (c 2008)
- Redis[29] by Salvatore Sanfilippo (c 2009)

The two TupleSpace implementations are of interest, but both operate behind an object-method interface exclusively, rather than behind arrays. This does have the advantage that other functionality beyond set/get/delete (e.g. non-blocking reads) can be accessed via specific methods, but it doesn't meet our goal of working with arrays in the simple case.

Arraysync is designed for performance. Like Tequila it doesn't have built-in security or authentication, but it allows any client to become a server and is well suited to high-performance applications in peer-level private networks.

Tie is a framework for the creation and management of persistent Tcl arrays. It is targeted at flexibility, allowing arbitrary back-ends (data sources) to be used. Tie provides a number of data sources (arrays, remote arrays, files) and there are others, such as Colin McCormack's Metakit

back-end for Tie[30]. McCormack notes that the Metakit back-end can be combined with Tie remote arrays to provide a Tequila-like facility.

Redis is a key-value database, with a built-in network interface, that supports strings, lists and sets with atomic operations to push/pop elements.

Tgdbm is a wrapper for the Gnu GDBM, and also provides both an object/method interface and can be tied to arrays. It also supports an 'attach' mode that allows for sparse arrays (that is, not all of the database need be present in memory). Tgdbm shows how a hybrid array/object API might look.

And then there is the Tequila1 code itself, just 140 lines of client-side Tcl code, with three main API calls: open (a connection to the server), attach (to an array) and close (the connection).

At its simplest, the Mockingbird API will have 5 commands:

connect	connect to an application's cloud
authenticate	authenticate the user
attach	attach to an array
detach	detach from an array
disconnect	disconnect from the Cloud

The client login process involves two steps:

- the stargus::connect command takes a single argument -- an ID for the application's Stargus server (actually a URL). If successful, Tcl code is returned from the server to define the stargus::authenticate command
- the stargus::authenticate command is then called to authenticate the user

When called with no arguments, the stargus::authenticate command will pop up a Tk login form to prompt for the user and password. It can also be passed the user or the user and password as arguments, and optionally a Tk frame in which to display the form. If the user and password are both supplied, the form isn't displayed, allowing a custom form to be implemented should the developer need to.

The rationale for the connect command defining the stargus::authenticate command is threefold:

- it saves each application developer from having to do so
- it facilitates consistency across applications
- it allows for improvements to be made centrally and automatically become available to Stargus applications.

A typical connection will look like the following:

```
if [stargus::connect mysite.net/myapp] stargus::authenticate
```

The authentication process will also provide the client with a one-time private key to use when encrypting and signing communication to the server, similar to the cookie provided to an authenticated web client by BetterID.

Once authenticated, the client can attach arrays and (like Tequila 1) any changes will be synchronous with respect to the client -- that is, changes are reflected in the array immediately and traces propagate the changes to the Cloud.

Unlike Tequila 1, Mockingbird supports access permissions using PoliTcl. In the first version, the permissions will be restricted to read or read/write on attached arrays, but there is no reason this couldn't be extended to permissions on individual elements within the array.

As well as confirming the user's identity, the stargus::authenticate command returns the user's PoliTcl rules, so that the Mockingbird client can perform some permissions checking. When a user attempts to write without permission, the Mockingbird client will silently ignore the change, although there will be an option on the attach command to specify that an error should be generated, or a call-back procedure invoked instead. Of course, changes will never be propagated by the server unless permitted by the user's PoliTcl rules, but this approach allows more graceful handling in the client application.

All communication with the Stargus Cloud can be encrypted using the BetterID security system. Encryption can be specified either globally, in the stargus::connect command, or in individual stargus::attach commands.

The stargus::detach command will unset the array in the client but not affect the persistent storage nor other clients. The stargus::disconnect command will disassociate the authenticated client from the Cloud, and requires another stargus::connect / stargus::authenticate before the client can use Stargus features. There hasn't been a need for a "deauthenticate" command identified as yet.

Conceptually, the Tequila-style array interface corresponds to a single row in a database table, with each element corresponding to a column value. This is ideal for simple storage and allows traces to identify the individual elements that have changed in the array. But it doesn't allow for an array to represent a table in a database, or a spreadsheet – for that we need a Tequila 2 style object/method API.

4.5 Beyond Tequila 1 - Local Cache, Sparse Arrays, Tables and Locks

Before considering the design of the Tequila 2 style object/method API, the Tequila 1 issues were revisited to see if they could be mitigated (if not resolved) prior to the Tequila 2 approach being addressed.

There are two scaleability related issues:

- the startup time to retrieve and load the array on connection
- the storage requirements when arrays are large

The first can be addressed by caching the array locally, the second by allowing for sparse arrays (where values are only retrieved as needed).

A local cache will be implemented using a database library such as Metakit or SQLite, allowing the cache to be stored in a single file with no need for a database server. As the array is being filled on connection a copy will be written to the database, and subsequent updates either applied as they arrive or under program control at an application specific commit point.

On reconnection, the cache will be checked for consistency (via a checksum generated when it was committed) and the array initially filled from the cache. The cache will also contain the ID of the last contiguous update to be received and applied (a contiguous update is one that has arrived "in order"). Once the array has been filled from the cache, the subsequent updates (potentially including repeats of "out of order" updates) will be retrieved from the server and applied to the local array.

Sparse arrays can be implemented by using Tcl variable read traces. This will allow "lazy" attachment of arrays to the back-end storage, with values filled when the array element is read.

The following script demonstrates how variable read traces work:

```
proc ReadCmd {v1 v2 op} {
    upvar 1 $v1 v
    if {![info exists v($v2)]} {
        after 5000 set ::$v2 1
        vwait ::$v2
        set v($v2) $v2
    }
}

proc WriteCmd {v1 v2 op} {
    upvar 1 $v1 v
    puts "${v1}($v2) set to $v($v2)"
}

array set foo {}
trace add variable foo read ReadCmd
trace add variable foo write WriteCmd

puts "initial values = [array get foo]"
after 2000 set foo(new) 10
puts "foo(bar) = $foo(bar)"
puts "foo(bing) = $foo(bing)"
puts "final values = [array get foo]"
```

The output is as follows (text in brackets are comments not part of output):

```
initial values =
foo(new) set to 10      (after 2 seconds)
foo(bar) = bar          (after a further 3 seconds)
foo(bing) = bing        (after 5 seconds)
final values = bing bing bar bar new 10
```

Note the following:

- the "`array set foo {}`" command is important because the process breaks down if the variable isn't an array to start with
- the '`after 2000 set foo(new) 10`' command arranges for a value to be set after 2 seconds, to confirm that write traces still work
- the ReadCmd procedure is called when the read trace fires -- if the array element isn't defined then it waits for 5 seconds, sets the value and returns -- simulating the delay in retrieving the value from a server
- the '`puts "foo(bar) = $foo(bar)"`' command will block until the read trace returns (after 5 seconds)
- during this time, the write trace will fire when `foo(new)` is set to 10
- after the first puts returns, the second will take a further 5 seconds
- since all the values are now set, the read trace won't wait for 5 seconds, and the final `[array get foo]` will return immediately

With such "lazy" attachment it will be possible to implement a form of "memory management" to keep the size of attached arrays in the client within bounds -- useful in memory constrained clients or for large arrays. A periodically scheduled "reaper" command can unset array elements based on criteria such as size or frequency of use. When next accessed, these elements will be obtained from the Cloud. Alternatively, the Mockingbird client can prune the array before setting a new value when a threshold has been reached.

An alternative to be explored will be the use of an array of dicts to represent the SDB domain -- one element per item (keyed by the item name), the value being a dict representing the attribute/value pairs from SDB.

This approach will allow traces to identify which item has changed, but not the attribute(s) that have changed. For that, a "dict diff" command will need to be implemented, initially in Tcl but, if successful, a TIP[31] may be raised to specify a C version for performance. Alternatively (or perhaps, in addition) the Mockingbird client code would maintain an additional array identifying the attributes that have been changed in the most recent update.

One upside benefit to using an array of dicts is that the "dict with" [32] command can be used when processing items, leading to much cleaner application code:

```
foreach item [lsort [array names myarray]] {
    dict with $myarray($item) {
       puts "Name:    $name"
       puts "Address: $street, $city"
    }
}
```

A distributed locking facility has already been layered on top of Tequila in the Kitview project[33]. This facility will be updated and provided as part of Mockingbird. Locks can be general (i.e. an entire array) or specific (a range of elements down to elements). When accessed via the object/method API, individual item attributes can be locked.

A general lock can be made more specific (e.g. a lock on an entire array can be changed to a lock on an individual element). Specific locks can only be made more general if this won't conflict with an existing lock.

If other clients hold a lock preventing a lock request from being granted, a "wait lock" can be granted and the client notified by a call-back when the lock has been granted. A client can hold several locks or wait-locks concurrently on different arrays.

4.6 Further steps - thinking async

An asynchronous T2-style API will also need to be supported, although simplified. There is no need for RPC Endpoints in the Cloud environment. There will be one pool corresponding to the application ID, and the collections will correspond to the SDB domains.

Taking the lead shown by Tgdbm, it will be possible to open a collection and optionally specify an attached array. Any operations via the array will exhibit the Tequila 1 semantics (synchronous in the client) while operations via the object/method API will be asynchronous:

```
stargus::collection mycollection -array name
```

The data structuring in collections will be a simplified version of that in Tequila 2 -- no unnamed storage and simple key/item and attribute/value mapping onto SDB.

Notifiers will be similar to Tequila 2, again, simplified given the Cloud environment and storage based on SDB. The events supported will include attachment of arrays and add/change/deletion of elements.

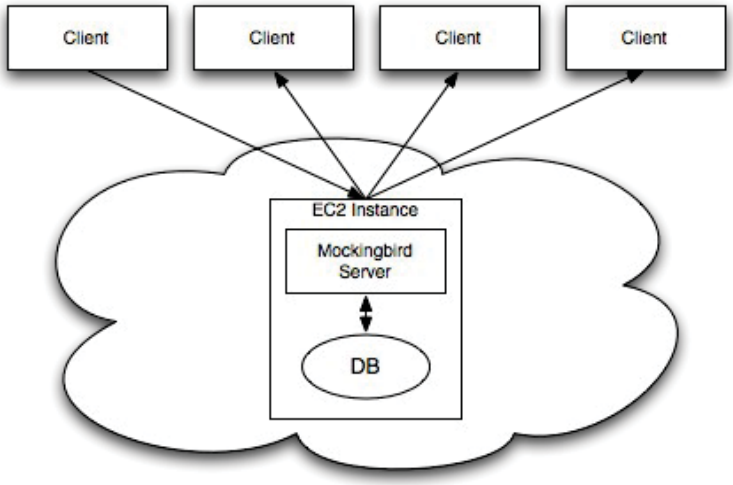
Such an asynchronous API will require a significant change in mindset for developers. As Wippler notes in his EuroTcl 2005 paper, in this environment "everything is about state and consequences of state change" [34].

But the results are worth it: "As Tequila proves, location, persistence, sharing, and application launches & exits can become details that will need much less attention than before" [ibid]. The reader is referred to Wippler's paper for a succinct handling of the issues and consequences of this.

4.7 Mockingbird Architecture

The initial Mockingbird design direction was to take the existing Tequila server code and run it on an EC2 instance:

In this model an update made in one client is sent to the server, stored in a local database then propagated to the other clients.



This approach has a number of issues, including:

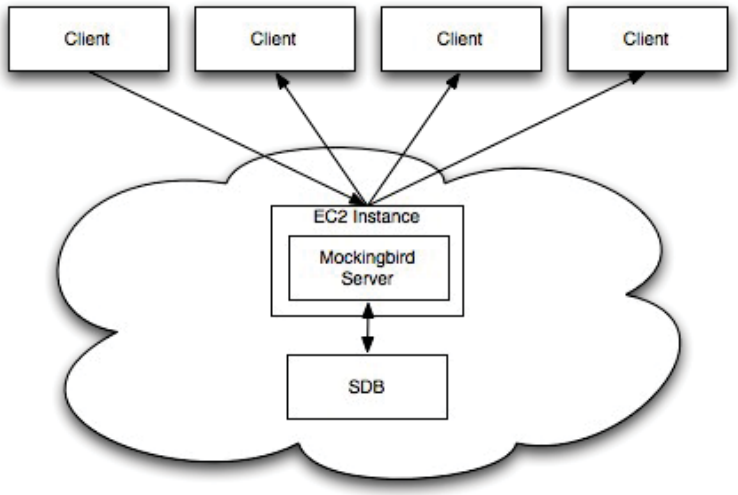
- it won't leverage the benefits of Cloud Computing -- in particular, having storage in the virtual machine will mean multiple EC2 instances can't easily be instantiated for scaleability
- communication would be via socket connections, and therefore may not get through firewalls nor would it be secure by default

The first design concept to leverage the Cloud was based on using the Amazon AWS storage facilities behind a re-implemented version of the Tequila server that used http for communication.

Amazon provides two storage services:

- SimpleDB (SDB) is a web service providing the core database functions of data indexing and querying
- Simple Storage Service (S3) is a web services for storing and retrieving data buckets of any type (text or binary) or size (from 1 byte to 5Gb), retrieved by key.

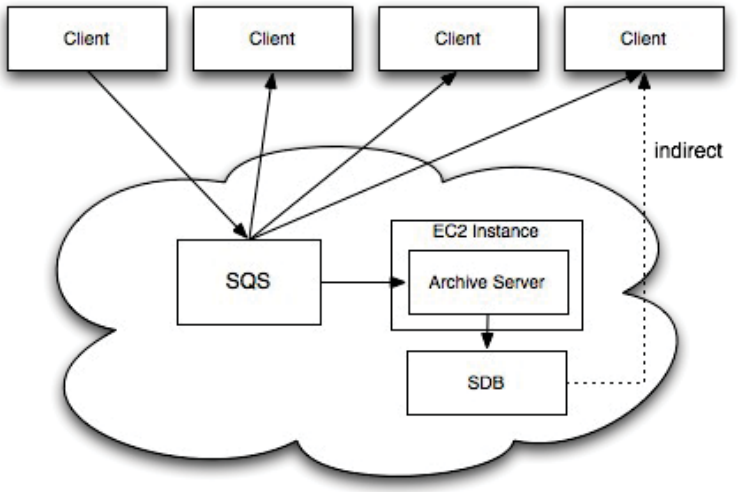
A Tequila server running on an EC2 instance can comfortably use SDB for storage, with one main limitation -- SDB data elements (a.k.a. fields within row within a table) are limited to 1Kb in size. However, this can be worked around by storing larger values in multiple elements or adding a level of indirection and storing the value in an S3 bucket.



Changing to use http requests instead of sockets would alleviate the firewall concerns, and using SDB for storage provides scaleability, performance and resilience. But there would still be a single server that could become the bottleneck.

Ideally Mockingbird could either run server-less or with multiple servers for scaleability. Either way, the use of http for communications would preclude this without adding complicated failover schemes and/or connection routing.

As an alternative, the Amazon SQS (Simple Queueing Service) was considered for client/server (and possibly inter-server) communication. SQS is a reliable, highly scalable, hosted message queueing system.



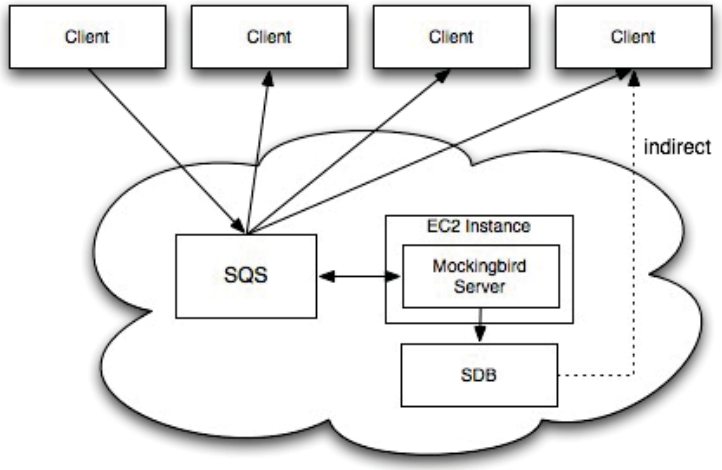
The first SQS-based model considered was server-less, except for an archive server used to log transactions to disk, so new clients can be brought up-to-date on connection.
Each array being shared has an SQS queue associated with it -- users have read and/or write permission to the queue depending on their access permissions for the array.

In this design, when an array element is changed in a client, a message would be written to the message queue for the array, all active clients listening for messages on that queue would be notified and would apply the update to their copies of the array. This could either be done directly (if the message contains the new value) or indirectly (the message indicates which array element has changed, and the client pulls the new value from SDB).

When a new client connects it would directly load the initial values for each array from SDB. The server's only function is to archive updates into SDB.

While fine in theory, this approach isn't feasible since SQS doesn't work as a "broadcast message" service where multiple readers each get a copy of the message. Rather, SQS guarantees that at least one reader will see the message, and provides a feature (the visibility timeout) that guarantees only one reader sees the message.

The design eventually decided upon is a variation on the following:



There is a global "write" queue used by all clients, onto which changes made by a client are multiplexed. Each client also has an individual "read" queue on which change notifications from other clients are written by the Mockingbird server.

When a client changes a value, a message is written to the global "write" queue. The Mockingbird server will receive the message and (after authentication and checking the user's permissions) write the change to SDB and propagate the message to the "read" queue of all other clients attached to the array.

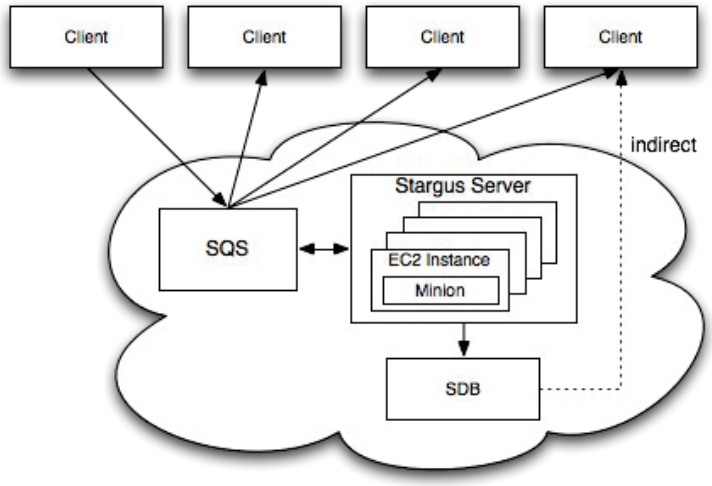
The attraction of this approach is that the EC2 instance running the Mockingbird server doesn't have any state -- changes are read, stored in SDB and propagated to other clients via each client's own SQS "read" queue. SQS ensures that a message is locked when a message is being processed, so that other readers can't try to handle it simultaneously. This means multiple servers can easily be instantiated depending on the load.

The other advantage of this approach is that much finer access permissions than simple array read/write permissions can be implemented since the Mockingbird server will only send updates to clients entitled to see the information they contain.

Each update will be assigned a unique identifier by the Mockingbird server. This is necessary because SQS doesn't guarantee the order of arrival of messages. This won't be a problem for most applications since the client can simply discard earlier updates if a later one has already been applied. There will however be a class of applications that do care (e.g. logging applications) and so there will be an API call to specify a callback procedure to be run when an update arrives. This can be used to handle out-of-order updates in an application-specific manner.

The server is conceptually the combination of all EC2 instances, so the individual server instances have been named "minions" to distinguish them from the overall server, which has been called the Stargus server, since it will handle a number of other functions besides storage:

- authentication upon connection (using BetterID)
- access policies (using PoliTcl)
- transactions and logging



Minions will use SDB, SQS and possibly sockets to communicate amongst each other.

Minions themselves can be multi-threaded, to maximize the throughput. Although it may well be that the event-driven single-threaded approach will suffice, and this step will only be taken after

gaining experience in the practical issues of performance management in a cloud computing environment.

In summary, Mockingbird will provide shared, persistent storage across multiple clients using the features of the Amazon Web Services (AWS) Cloud Computing environment to achieve scaleability, performance and reliability:

- SimpleDB (SDB) to provide indexed, structured storage
- Simple Queue Service (SQS) to provide inter-process and inter-computer communication
- Elastic Compute Cloud (EC2) to provide on-demand and scaleable computing capacity

4.8 Mockingbird Security

The security model in Stargus multiplexes multiple users onto a single Amazon AWS account. While it would be simpler for each user to have a separate account, that isn't practical from a deployment or infrastructure viewpoint. For example, most applications will likely want a single point of billing for Cloud services, and a single view of the application usage.

AWS uses private and public keys to authenticate a client when transmitting a message via a SQS message queue. This presents a problem, because the client must have a copy of the private key to sign a request, which would introduce unacceptable security issues for a shared account.

If only part of a request could be signed, the Stargus server could provide a client-specific pre-signed prefix that could be prepended to any communication. This would allow a request to be accepted by SQS and delivered to the Stargus server, where PoliTcl rules could determine if the user was allowed to perform the operations requested.

But since this isn't possible, the solution is to use anonymous access to queues and handle extra security and authentication in Stargus itself. So, anyone can write to the global "write" queue, and anyone can read from the per-client "read" queues. Although the name of each queue will be obfuscated and provided to valid clients upon authentication, this potentially leaves the Mockingbird system open to abuse. Such potential abuse would not be limited to denial of service. The account owner is responsible for all costs related to the queue, so there will need to be other measures taken to restrict access and detect abuse.

Fortunately, SQS includes an access policy language that can restrict connections by IP address (and also time). When the client application is authenticated using `stargus::authenticate` (which uses http/https) the client's IP address will be authorized to write to the global "write" queue. Likewise client's "read" queue will be created and restricted to that IP address, and can be used as a "back-channel" to instruct the client to change to another queue periodically or if invalid connection attempts are detected.

Even if the client is behind a NAT, this will greatly reduce the opportunity for abuse. But it doesn't stop abuse from a valid user, so the rate of queue write attempts can be monitored and an IP address blocked if necessary. This would introduce a potential denial of service vulnerability, but would nevertheless prevent unbounded charges to the AWS account owner.

A downside of this approach is that the client can no longer access the SDB copy of the storage directly when initially connecting, unless the data is made publicly readable. The Stargus server will return the data to the client or, alternatively, it could create a temporary encrypted copy of the data and supply the client with the URL to load it directly).

4.9 Data Dictionary

Mockingbird will include a data dictionary based on that used in the KitView project. The KitView dictionary contains a description of each data collection, sufficient to generate a user interface at run-time. This information can include such meta-data as:

- data description (table, column, format)
- validation, access and transformation rules (written in Tcl)
- an index of application specific scripts (that are themselves stored in the Cloud)

Information in the data dictionary is presented as collections, like any other Mockingbird storage. In addition, Mockingbird will maintain several collections that contain:

- a log of application usage
- a transaction log of changes
- active clients
- active locks

Access to the dictionary will be controlled by PoliTcl.

5 Local Area Clouds

Not all Stargus applications will need to use the AWS (or other) commercial cloud services.

For this reason Stargus introduces the concept of "Local Area Clouds" -- that is, an implementation of the Stargus server that will work over local networks, using a database for storage and Zeroconf [35] for service discovery.

The actual database to be used isn't critical, so, although a Metakit back-end will be used in the first instance (to leverage the existing work in Tequila 1 and 2), there is no reason why TDBC can't be used to provide access to arbitrary database systems.

Zeroconf (aka Bonjour, previously called Rendezvous) is a technology that provides (amongst other things) automatic location of network services. Bonjour is included in Apple's MacOS X, and there are implementations available for both Linux (and other Unices) and Windows. There is a Tcl client implementation[36] available, and it is proposed to use this as the basis for the Stargus Zeroconf client support.

A local Stargus server can be deployed as a self-contained Starkit, and clients will not need to know the IP address of the server running it.

If an organization wants to deploy an in-house Private Cloud – i.e. larger than a Local Area Cloud but without using an external commercial service such as AWS – then there are vendors of Cloud solutions that support the Amazon AWS APIs. One example is Eucalyptus Systems [37], which provides both Enterprise (commercially supported) and Open Source software for implementing private and hybrid clouds, using the same Amazon AWS API.

The Local Area Cloud will complement the hosted Cloud deployment option -- allowing Stargus applications to seamlessly scale from individual computers (perhaps with the client and server in the same process) to local networks, and even to global networks.

References

[1] Starkit deployment technology - http://www.equi4.com/starkit
[2] Cryptkit - a Tcl binding to the Cryptlib Security Toolkit - http://wiki.tcl.tk/cryptkit
[3] Tequila T2 - http://www.equi4.com/metakit/tequila.html
[4] Vlerq - a research project about data structures and persistence http://www.equi4.com/vlerq.org/
[5] Tkhtml3 - http://wiki.tcl.tk/15586
[6] Politcl - In dotNyet - Proceedings of the 15th Annual Tcl/Tk Conference, Manassas, VA, 2008 http://www.lulu.com/content/3830746
[7] Critcl 2 - http://wiki.tcl.tk/14655
[8] Cloud Computing - http://en.wikipedia.org/wiki/Cloud_computing
[9] Cory Doctrow - Not every cloud has a silver lining http://www.guardian.co.uk/technology/2009/sep/02/cory-doctorow-cloud-computing
[10] Jon Stokes - "The Cloud": that term does not mean what you think it means - Ars Technica http://arstechnica.com/staff/carthage/2009/09/the-cloud-that-term-does-not-mean-what-you-think-it-means.ars
[11] Google App Engine - http://code.google.com/appengine
[12] Microsoft Azure - http://www.microsoft.com/azure
[13] Salesforce.com - http://www.salesforce.com/cloudcomputing
[14] Amazon Web Services - http://aws.amazon.com
[15] Tcl Database Connectivity - http://wiki.tcl.tk/TDBC
[16] SQLite - http://wiki.tcl.tk/SQLite
[17] Tequila - http://www.equi4.com/metakit/tequila.html
[18] Metakit - http://www.equi4.com/metakit
[19] Collaborating applications: Tequila takes Tcl further http://www.equi4.com/pub/docs/tcl2005e/tequila.pdf
[20] GroupKit - http://wiki.tcl.tk/groupkit
[21] StarDLL - http://wiki.tcl.tk/StarDLL
[22] Amazon SimpleDB - http://aws.amazon.com/simpledb/
[23] Tuplespace - http://c2.com/cgi/wiki?TupleSpace and http://en.wikipedia.org/wiki/Tuplespace
[24] Tcl Tuplespace implementation - Todd Coram http://wiki.tcl.tk/tuplespace
[25] Tgdbm - http://wiki.tcl.tk/tgdbm

[26] Tie - http://wiki.tcl.tk/tie
[27] Arraysync - unpublished - Steve Redler IV <steve@sr-tech.com>
[28] Tupleserver - ttp://wiki.tcl.tk/tupleserver
[29] Redis - http://code.google.com/p/redis/
[30] Metakit backend for tie - http://wiki.tcl.tk/13716
[31] TIP - Tcl Improvement Proposal - http://wiki.tcl.tk/TIP
[32] dict with - http://wiki.tcl.tk/13733
[33] Kitview - ftp://ftp.oreilly.com/pub/conference/os2001/tcl_papers/landers_v1_1.pdf
[34] Collaborating applications: Tequila takes Tcl further
http://www.equi4.com/pub/docs/tcl2005e/tequila.pdf - page 8
[35] Zeroconf - http://www.zeroconf.org/
[36] Tcl_Bonjour - http://github.com/dongola7/tcl_bonjour/tree/master
[37] Eucalyptus Systems - http://www.eucalyptus.com/