# A Multi-threaded Server for Shared Hash Table Access

Andrej Vckovski and Jason Brazile
*Netcetera AG*
{andrej.vckovski,jason.brazile}@netcetera.ch

## Abstract

This paper presents a multi-threaded socket server allowing access to shared hash tables. It is implemented using Tcl 8.1 multi-threading capabilities and runs multiple Tcl interpreters to service client requests. The application is designed as a pre-threaded server which allows a single working thread to handle many requests. The central shared data object is a hash table with structured values which allows access by all threads. Synchronization is based on a reader/writer lock implementation using the synchronization primitives available in Tcl, i.e., mutexes and condition variables. The application achieves insert rates that are significantly higher than what current commercial database management systems achieve. The usage of third-level language programming in C and application-specific scripting in Tcl allows a design based on a light-weight, robust kernel on the one hand and easily modifiable application-domain code on the other. The experiences with thread-safety and other threading features in Tcl 8.1 have been largely positive in this real-world application.

## 1 Introduction

There are two motivating factors for presenting this system. First, it provides an example of the usage of some of the multi-threading capabilities introduced with Tcl 8.1. Second, it provides a general solution to the frequent desire to have shared, direct access, tabular data in the context of transaction-oriented applications such as programs with an HTML graphical user interface.

The original motivation for this application, however, differs slightly. The initial purpose was to support a data feed handler to provide a cache for pseudo-real-time financial market information. We describe the initial requirements in a first section, which is followed by a discussion of the system architecture. Then, we give an overview of two central implementation aspects: a pre-threaded socket server and synchronized access to Tcl hash tables. In the final section, we discuss our experiences with the Tcl threading API and some performance results.

## 2 Application background

Several large data vendors such as Reuters, Bloomberg, Bridge and others provide numerous kinds of financial market information, such as stock quotes, using various formats and mechanisms. Also, many stock exchanges worldwide provide direct access to their data using various formats and mechanisms. A common characteristic of all these data providing mechanisms is that the data is delivered as a sequence of incremental updates to some base information. Such incremental updates usually contain a unique ID identifying the item being updated and then a set of key/value pairs providing the new attributes of that item. Such attributes might be for example the last paid price and its associated time stamp for a given financial instrument.

There are basically two typical access patterns to that sort of information. A user either requests all or selected attributes of a financial instrument (request/response) or the user subscribes to a financial instrument with the intention to receive all further updates to that item (subscribe/notify). Since the update rates on data feeds can be quite high (several hundred update messages per second) it is common to store all information in main memory, especially as memory cost currently allows main memory sizes of several gigabytes. Thus, such main memory caches need to support insertion of incremental updates on the one hand while being able to answer user queries for current data on the other.

The application presented here needs to handle various data feeds as described above. Therefore, it was necessary to provide an environment that allows quick change cycles. If 6 different data formats have to be supported and every data format definition changes once a year, the overall change rate might be one change every 2 months. Also, the feed handler, being a server or daemon process, needs to be very robust because it will have very long run times and be accessed by many clients. For the implementation, there were basically three implementation alternatives:

- Use a commercial (disk-based) database management system (DBMS) that can be tuned to cache tables entirely in main memory (for performance reasons)
- Use a commercial main-memory database (MMDB)
- Develop a specific cache manager

The first two options had been rejected because of price/performance issues (e.g., an RDBMS that has to provide these update rates is very expensive to build and maintain) as well as robustness and complexity (MMDB) issues.

Also, the typical data structure can not be efficiently described as relations. There are many possible attributes for every item (instrument), but in most cases, only a few of them actually have values, i.e., most 'columns' are 'null'. In addition, frequent changes to the set of attributes are made, usually consisting of the addition of new attributes. Therefore, a relational approach would be either computationally expensive if the attributes would need to be normalized in a second table, or hard to maintain if the database structure had to be permanently modified. Based on these arguments, we decided to build a custom feed handler and cache management system using as much existing technology as possible and focusing on simplicity, robustness and maintainability.

## 3 System architecture

Based on the requirements discussed above, we chose an implementation strategy consisting of:

- A multi-threaded socket server
- Using Tcl's hash tables for data storage
- Using one Tcl interpreter per servicing thread to provide easy customization of data manipulation

The idea was that the server would run a thread for every connected client and maintain a set of shared hash tables. Every thread would have its own Tcl interpreter with which the clients would communicate, and the interpreter would have special commands that allow clients to access the shared hash tables, i.e., insert, retrieve and delete keys and values. The server should only provide very general functionality and let users or client applications implement most higher level functionality using Tcl code rather than more error prone C or C++ programming.

This application needs to run on Unix platforms and so we chose C, Tcl 8.1 and POSIX threads as the implementation basis. C++ was considered somewhat fragile because in multi-threaded applications it is always very important that one knows exactly what is going on to prevent race conditions and deadlocks. C++ compilers and runtime libraries tend to include hidden overhead into the application code that cannot be easily tracked. Also, we wanted to reduce overall complexity as much as possible.

The choice of a multi-threaded server allows for an easy usage of shared data structures, e.g. as in the hash tables mentioned above. However, unlike a forking server that spawns off separate processes to service requests, a multi-threaded server is less robust in the sense that there is no address space isolation preventing problems in servicing a request that can impair overall stability. Still, we believe that by making a server as generic as possible, we can reduce the complexity to a level where it is possible to design, implement and test a fairly stable multi-threaded server.

The communication with clients is based on TCP using Berkeley sockets with a simple message format. A request message consists of Tcl code that is evaluated in the thread's Tcl interpreter. The message result is either the result of the evaluated Tcl code or the corresponding error message. That is, the model is similar to Tk's `send` command.

The application's main thread also has a Tcl interpreter. This interpreter is "connected to" standard input and output (i.e., takes commands from standard input) and has a few additional commands that support creation of the server socket, maintaining the number of current threads and doing signal handling. Signal handling is an important requirement for long-running processes and is discussed in a separate section below.

In addition to threading and hash table extensions, every Tcl interpreter (i.e., the per-client thread interpreters and main thread interpreter) have additional commands for logging. Similar to signal handling, useful logging with several levels of verbosity is also very important in long running processes because such programs usually have no user interface and therefore rely on log files for their output. Signals and log files can be seen as a kind of primitive user interface to daemon processes.

The architecture described above is shown in figure 1.

The next section covers a few aspects of the implementation.

## 4 Shared hash tables and locking

Most objects being used by multiple clients at the same time have a need for some kind of synchronization. Sometimes that need is even the key motivation for the object to be used by multiple clients. In our case, the synchronized objects are hash tables that can be accessed by all threads. The synchronization has to guarantee consistency of the hash tables with parallel inserts, updates, and queries on the data. The selection of a synchronization mechanism usually depends strongly on the expected access patterns which can significantly influence system performance if, for example, an implementation variant is chosen that leads to high lock contention. The main synchronization questions we faced here were similar to typical locking issues in database management systems:

- Locking granularity
- Lock sharing

Locking granularity describes the trade-off between fine-grained locking, which minimizes lock contention but needs a large number of locks to be acquired when doing operations involving many data items, and course-grained locking, which uses fewer locks at the expense of performance. For example, most commercial database management systems offer different levels of locking granularity. On full table scans single table locks are used, while cursor operations rely on a logical row-level locking or a physical page-level locking. However, such adaptive locking granularities also introduce a lot of additional complexity which - especially in the case of locking - increases the risk of errors and dead lock situations.

Lock sharing describes the situations where there is an access pattern which allows readers and writers to be distinguished. Readers can be synchronized using a shared lock (many readers can simultaneously access a resource) while writers need exclusive access to the resource. Again, there is usually a trade-off between using exclusive locks and shared locks. Shared locks are typically more expensive to acquire but reduce lock contention.

In our application, a few estimates showed us that as a first approximation, it would be sensible to use shared (reader/writer) locking on the entire hash table. That is, a reader always locks the entire table for shared access, and a writer locks the entire table for exclusive access. This approach turned out to be very efficient in our case where there is typically only one or a few writers and many readers.

The (preliminary) Tcl 8.1 threading API offers (similar to POSIX threads) two synchronization primitives: Mutexes (exclusive locks or semaphores) and condition variables. It was therefore necessary to provide our own implementation of reader/writer locks based on mutexes. Considering the frequent need for reader/writer locks and their simple implementation, it would be worthwhile to include an implementation in the Tcl API.

Our shared hash table implementation was therefore protected by a single reader/writer lock that is acquired depending on the type of operation. The hash table is based on Tcl's hash table with the extension that there is additional structure imposed on the values entered in the hash table: all values ('rows') are entered as attribute/value pairs allowing selective 'columns' or attributes to be queried and updated.

The access to shared hash tables is implemented using two command procedures. The first command called `sharedhash` is used to create, delete and use hash tables in an interpreter. After having either created a new hash table or acquired a handle to an existing hash table, a second command is created in that interpreter with the same name as the hash table. The hash table is then accessed using this command much in the same way as for example Tk widgets or [incr Tcl] instances are associated with a command, as the following example illustrates:

```
# create a shared hash table
# with the name 'foo'
sharedhash create foo 20
# use the hash table named
# 'foo' in this interpreter
```

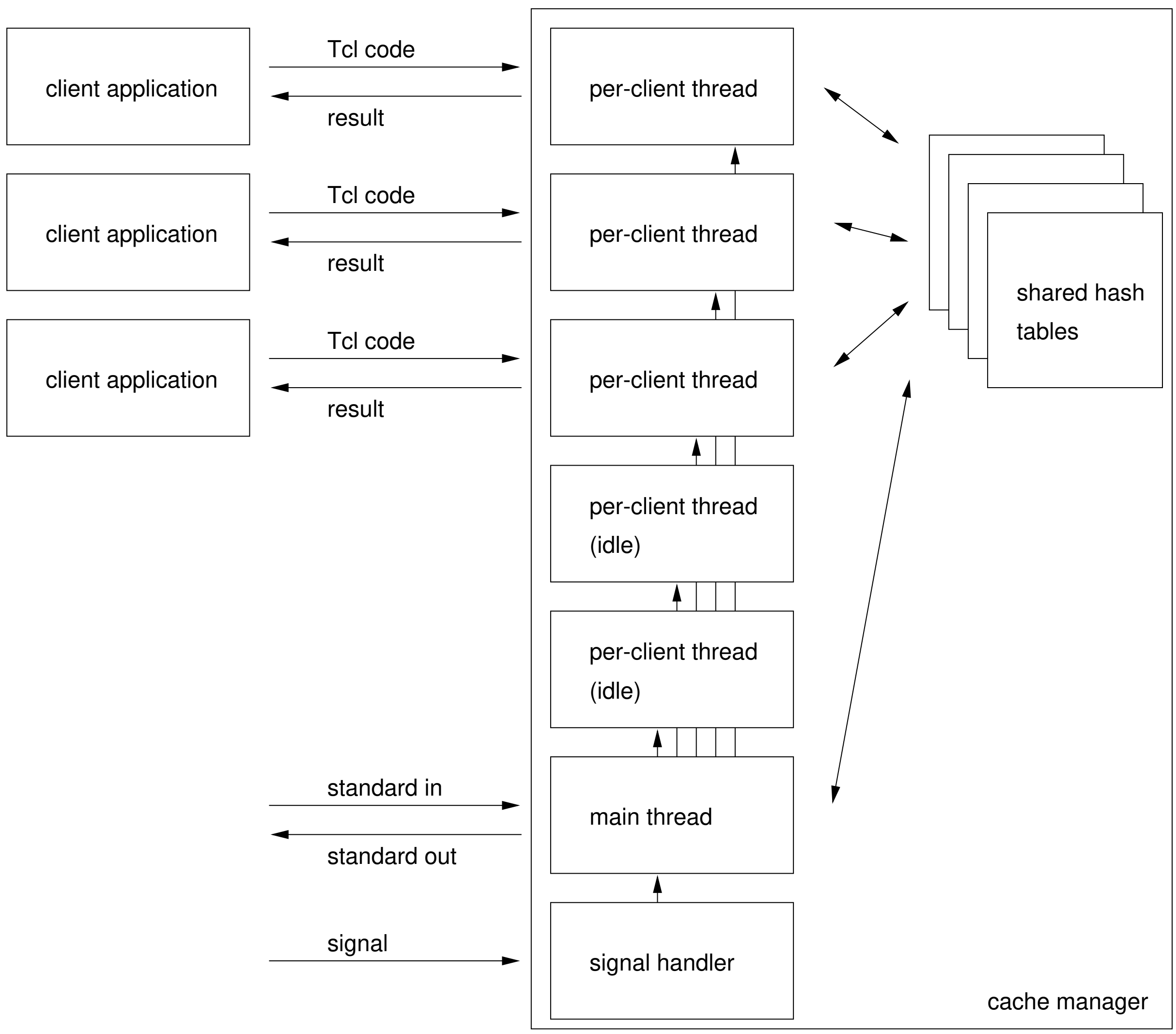


Figure 1: Architecture overview

```
shared hash use foo
# access the shared
# hash table
foo set 12345 {
  attrA valueA
  attrB valueB
}
# set an array with the
# attributes of the hash value
array set res [foo get 12345]
```

The following subcommands for the `sharedhash` command are supported:

**sharedhash create *name ?maxattrs?***
: Creates a new shared hash table with the given name which can hold at most *maxattrs* attributes per row.

**sharedhash use *name***
: Makes the shared hash table with the given name accessible in the current interpreter as a command with the same name.

**sharedhash names**
: Returns a list of all defined shared hash tables

**sharedhash forget *name***
: Removes a reference to the shared hash table from this interpreter, i.e., deletes the associated command. If that was the last reference, deletes the hash table.

The hash table is then accessed using the following subcommands:

***name* set *key attribute-value-list***
: Sets the *attribute-value-list* to the current content of the hash tables value associated with *key*, deleting all previously defined attributes.

***name* get *key ?attribute-list?***
: Gets either the named or all attributes for the given *key*.

***name* update *key attribute-value-list***
: Updates the current values associated with *key* with the *attribute-value-list*.

***name* names *?pattern?***
: Returns a list of keys matching the pattern or all keys

***name* attributes**
: Returns a list of all the currently defined attributes

***name* delete *key***
: Removes the entry from the hash table

***name* foreach *pattern varnames attributes code***
: Loops over all entries with keys matching the pattern and assigns the variable names in *varnames* the values of the attributes *attributes*.

***name* updateforeach *pattern variablename code***
: Loops over all matching entries with a write lock and executes code for every entry. The code can access the current element using a special key `#current`.

***name* stats**
: Returns hash table statistics as returned by `Tcl_HashStats`.

## 5 Pre-threaded socket server

As mentioned above, performance was a critical issue. Therefore, we decided to base the server on a pre-threaded design [1]. This means that the servicing threads are not created for every request but pooled in advance. This approach has a few advantages:

- Threads can be easily re-used. Initializations such as the creation of a Tcl interpreter needs only to be done once. A thread can service many requests, i.e., not every new request needs the creation of a new thread.
- Overall server design is more symmetrical because there is not a special thread that accepts connections and spawns off new threads.

However, there are also some drawbacks with a pre-threaded solution. As the load changes over time, it might be necessary to asynchronously create and destroy threads. While the creation is simple, the destruction of running threads is not trivial, especially if the destruction should not be deferred.

A servicing thread's main function is to perform the following steps:

1. increment the thread counter

2. create an interpreter
3. register all additional commands
4. evaluate thread constructor Tcl code
5. while (termination not needed)
    (a) acquire exclusive lock to accept a connection
    (b) wait for a incoming request (`accept`)
    (c) release lock
    (d) increment working thread counter
    (e) while (not end-of-file of socket)
        i. receive a message
        ii. evaluate the Tcl code
        iii. send the message with the result back
    (f) decrement working thread counter
    (g) check if there are threads scheduled to terminate
6. evaluate thread destructor Tcl code
7. delete interpreter
8. decrement thread counter
9. terminate thread

## 6 Main thread

In addition to the request handler threads, the application also runs a main thread that is different from the request handlers. The main thread executes the `Tcl_Main` procedure and handles standard input. The interpreter in the main thread contains additional commands to create a (listening) server socket and to control the number of working and free (not bound to a request) threads.

Long running applications in an UNIX environment usually need to handle signals for various reasons. For example, it might be necessary to perform cleanup after the process has been notified to terminate (SIGTERM signal), the application might want to re-read configuration information, dump internal information to log files or change log levels. In multi-threaded applications, signal handling needs special consideration as the so-called *async-safety* (meaning that a system call is safe to be interrupted by an asynchronous signal delivery) and *thread-safety* (meaning multiple threads may simultaneously call the function) are orthogonal in the sense that thread-safe calls are not necessarily async safe [2]. For that reason, the best way to handle signals is to block all signals in all threads and create a special signal handler thread that only waits for signals and uses thread synchronization methods (e.g., condition variables) to notify other threads if necessary.

Therefore, the main thread in this application creates a dedicated signal handler thread and also defines a new Tcl command that allows the main thread's Tcl interpreter to be notified by new signals. To keep the entire application simple, we decided not to use Tcl's event loop and notification mechanism for this purpose.

In addition to the shared hash table functionality, the following commands are available in the main thread's Tcl interpreter:

**server** ***port ?nthreads ?constructor-code? ?destructor-code?***

This command creates a server socket and optionally *nthreads* servicing threads waiting for connections. It also allows constructor and destructor code blocks to be specified. These code blocks are executed in the servicing thread's interpreter after creation of the thread or before termination of the thread, respectively. This can be used, for example, to define variables and procedures, or load libraries or source code modules. This code does not use Tcl's `socket` function, in order to have more control over socket options such as address reuse.

**servercontrol** ***?minidlethreads? ?maxidlethreads?***

If called without arguments, the current number of available and running threads is returned. If called with *minidlethreads* and *maxidlethreads*, it ensures that there are at least *minthreads* and at most *maxidlethreads* idle, i.e., non-servicing threads available.

**waitforsignal** ***?sec? ?usec?***

Waits the given time (or forever) for any signal to be delivered to this application and returns the signal number.

Using this call, a typical code example running in the main thread might look like this:

```
#!/usr/local/bin/ncm

# create a shared hash
sharedhash create foo
```

```
# create a server socket
# and 10 servicing threads
server 6000 10 {
  sharedhash use foo
} {
  sharedhash forget foo
}

set terminate 0
while {!$terminate} {
  # wait 2 sec for a signal
  set s [waitforsignal 2]
  if {$s==15} {
    set terminate 1
  }
  # adapt number of idle threads
  servercontrol 5 20
}
```

## 7 Tcl and multi-threading

Multi-threading support has been on the wish list of the Tcl community for a long time. This was motivated mainly by application areas where Tcl was used as an embedded scripting language for some large software system, and where that software system was a multi-threaded application. The problem with Tcl was not so much that it did not support threading but that the Tcl library was not thread-safe itself (i.e., using non-reentrant functions, unsafe system calls etc.). Two extensions have emerged in the community to add thread-safety and eventually also thread support to Tcl. Steve Jankowski's MTTcl [3] used Solaris threads. Another extension was PtTcl or Pthreads-Tcl by Richard Hipp and Mike Cruse using POSIX threads [4]. Finally, thread-safety made its way into the Tcl core with Tcl 8.1. This is insofar important as most of the threading issues do indeed affect core components of Tcl and therefore, with Tcl 8.1, patching of the Tcl core isn't necessary anymore.

The multi-threading related issues in Tcl 8.1 are:

- Tcl core is thread safe, i.e., the tcl library can be used in multi-threaded applications
- Some internal data structures are stored as *thread specific data*. This allows, for example, every thread to have its own event queue.
- Some new API calls were introduced to create mutexes and condition variables. However, it seems at the time of this writing that this API isn't yet final.

The most notable missing features are official API calls for thread creation, scheduling and cancellation. Even though there are thread creation abstractions (e.g., to create a notifier thread), there are no externally usable (that is, declared in the external header `tcl.h`) functions for thread creation, cancelling and scheduling. For the time being, this is a wise choice because providing a platform independent abstraction would probably disarm many of the needed features of the platform-specific thread implementation. For example, POSIX threads allow a rich set of attributes for every thread that are not directly matched by Windows NT's different thread model. The decision to stick with a least common denominator is not a problem if the there are well-defined interfaces and transparent data structures to the underlying OS specific threading system. For example, it was easy to provide reader/writer-locks (based on Sun's SPLIT package [5]) using mutexes and condition variables once it was clear that the rest of Tcl did not use thread cancellation anywhere.

Our experiences with Tcl 8.1 and its multi-threading support showed that it is a safe choice to use Tcl 8.1 in multi-threaded applications. Even if none of Tcl's threading API is used (i.e., not even mutexes or condition variables), the thread-safety and usage of thread specific data for certain internal data structures is very useful if not mandatory within multi-threaded applications. However, it will be necessary in the future to provide detailed and precise documentation of those features. When writing multi-threaded applications, engineers are very much interested in knowing relevant "side effects" such as the creation of background threads and the like. Debugging multi-threaded applications is not trivial and therefore, one expects precise information on what should/will happen in API calls.

## 8 Conclusion and future work

The work presented here shows a successful example of using Tcl's new multi-threading features in a real-world application. Moreover, it proves again that a design approach using a small, reliable kernel written in a third-generation language and delegating most of the overall application complexity to a scripting language is a promising engineering approach. And, especially when developing multi-threaded applications, fighting complexity is the main objective of a system design in order to avoid dead locks, race conditions and the like. The entire cache manager and shared hash table implementation as described in here is less than 3000 lines of

C code (not including the Tcl library, of course).

The lightweight implementation is also a positive influence on performance. On a Sun 270 MHz UltraSPARC II processor, up to 1000 inserts per second can be achieved with simultaneous queries. The application went into production in late Summer 1999 in continuous (7x24 hour) operation and has not posed any major problem since. The application handles more than 1 million updates per day with peak rates of several hundred updates per second. The built-in Tcl hash function in Tcl's hash table implementation proved to be very effective, maintaining a short search distance and overall balance even in tables with more than 100,000 entries.

Our future plans with the cache manager are to provide better checkpointing (currently, the hash tables are periodically written to disk by a special client process though no consistency is enforced) and more synchronized elements among the servicing threads, e.g., named message queues. Eventually, we plan to release the code into the public domain.

## 9 References


1. Stevens, W. Richard. *Unix Network Programming (Vol 1)*. Prentice Hall, 2nd ed., 1997.

2. Lewis, Bill, and Berg, Daniel J. *Threads Primer*. Prentice Hall, 1995.

3. *MTtcl - Multi-threading for Tcl* `<http://www.activesw.com/people/steve/mttcl.html>`

4. *An Introduction To Pthreads-Tcl* `<http://www.hwaci.com/sw/pttcl/pttcl.html>`

5. *Solaris to POSIX Interface Layer for Threads (thread.c, thread.h and synch.h)* `<http://www.sun.com/workshop/threads/apps.html>`