September 01, 1999

Event-Based Servers in Tcl

Multithreading without threads

Stephen Uhler

Event-based programming is a powerful alternative to threads when building high-speed network servers. Stephen implements a web server in Tcl that is based on events and callbacks.

Stephen is a senior staff engineer at Sun Microsystems laboratories. He can be contacted at stephen.uhler@sun.com.

The primary goal of a high-performance server is to spend all of its time servicing client requests. It shouldn't spend any time waiting for client input, waiting for a client to accept output, or waiting for client-specific resources to become available. If a client or resource isn't ready, the server should deal with a different client instead.

Many people think that multithreading is a must for high-performance servers. In this article, I present an alternative to a threaded architecture that yields as good as or better performance than thread-based systems. To illustrate, I'll implement a web server that demonstrates the simplicity of my approach, which is based on events and callbacks. Finally, I'll suggest when it is appropriate to use event-based systems and when a multithreading approach is a better choice.

Thread-Based Servers

In an ideal world, writing a server is straightforward, as the following server loop indicates:

1. Get a request from the client.

2. Compute the response.

3. Transmit the response to the client.

As long as the client cooperates and is always ready, this server is ideal. There is no overhead switching among clients -- they are serviced as the requests arrive. The server operates in a strict first-come/first-served manner, and since the clients are always ready, the server is always busy servicing requests, thus achieving maximum throughput.

In the real world, where servers are fast and clients are slow, this simple scheme falls apart. The server can go no faster than its slowest client. While getting a request from the client, the server ends up waiting for the request to arrive. While the client is reading the response, the server waits some more. The solution to this waiting is to have the server work on another client's request while waiting for the first client to be ready.

A common approach to this situation is the use of threads -- a general-purpose mechanism for handling concurrency by providing multiple streams of execution with shared programs and data. Modern preemptive thread systems manage the switching of threads automatically and provide primitives for managing synchronized access to avoid data corruption.

On the surface, threads seem ideal. The simple request-at-a-time server magically becomes concurrent with threads. Where the server would have waited for a slow client, the underlying thread system realizes the server needs to wait, automatically switches to a new execution context, and begins servicing a different client. The code appears sequential on paper, but behind the scenes, the thread system magically switches

between threads, eliminating the time spent waiting for a client.

Unfortunately, threads aren't really so simple. Because the thread system can't know the intent of a particular program, it switches threads at an arbitrary point. If a thread is interrupted in the middle of modifying a shared data structure, the next thread gets corrupted data.

You need locks to prevent data corruption. A lock prevents thread A from running while thread B is modifying shared data. Forget a lock, and the program breaks. If A is waiting for B, and B is waiting for A, then you have deadlock, and the program breaks. If the locks are placed around large sections of code, so-called "coarse-grain locking," then threads are waiting on locks much of the time. Concurrency is reduced and performance is back to the level of the aforementioned simple-server scenario. If fine-grain locking is used, with locks tightly bracketing critical code sections, then concurrency is improved -- but it is easy to forget a lock, and overhead increases as more time is spent checking and managing locks.

Event-Based Servers

Where threads preserve the sequential look of the simple server by using concurrency, event-based systems require a different program structure -- an event loop that:

1. Waits for something to do.

2. Figures out what to do, and does it until I/O is needed.

Step 2 involves doing whatever the server can do without waiting, such as getting part of a client request, computing part of a result, or sending part of a response. The program flow no longer appears to be sequential. However, with this approach there is no concurrency, no locks, no data corruption, and almost no overhead.

All blocking I/O is replaced by event notification and callbacks. Instead of waiting for data to become available, the server is notified when it is available, processes the available data, then waits for the next notification. Notifications on behalf of many different clients can be intermingled, based solely on the order in which the clients are ready.

Tcl supports event-based programming via the *fileevent* command, which handles I/O events, and the *after* command, which manages temporal events. The *fcopy* command packages a common *fileevent*-*event* idiom in a single command for even higher performance.

When first faced with event-based programming, one source of confusion for threads programmers is how to deal with the program context. In a threaded system, the execution context for each thread lives on a stack. The threads package manages the saving and restoring of the execution stack as threads are switched in and out. With event-based systems, the execution context needs to be handled explicitly by you. When an event happens, you need to know for which client the event happens, and what the state of the client request was. Fortunately, managing the execution state associated with the events is easy to do in Tcl.

An Event-Based Web Server in Tcl

A web server implements the HTTP protocol, which defines how clients make requests and how servers respond to those requests. An HTTP request consists of a request line containing a URL, some headers, a blank line, and an optional body, which is typically used for form data in POST requests. The server reads the request, gathers the relevant information, finds the resource represented by the request, and replies to the client in a format that is similar to the request -- a response line, some headers, and the body of the response (the data displayed by the browser).

Available electronically (see "Resource Center," page 5) you'll find a small but complete web server that supports HTTP 1.0 and is written in about 200 lines of Tcl. It uses Tcl 8.0, the most recent production release of Tcl, and runs on Windows, UNIX, and the Macintosh. Rather than use Tcl namespaces, which earlier versions of Tcl do not support, I used the prefix "DDJ" on all my functions to prevent namespace clashes. This lets the code run on earlier versions of Tcl with only minor modifications. (Tcl is available at ftp://ftp.scriptics .com/pub/tcl/. A full-featured web server that uses the techniques presented and written entirely in Tcl by Stephen Uhler and Brent Welch is available at http://www .scriptics.com/tclhttpd/.)

The first thing the event-driven server does is call the socket command; see Listing One. The parameters to the socket command cause the server to call the function *DDJaccept* (a callback, see Listing Two) anytime the server accepts a socket connection from a client. The line *vwait forever* starts the Tcl event loop until the value of the variable *forever* changes, which, in this case, is never. From this point on, everything the server does is in response to an event.

When a client connects to the server, *DDJaccept* is automatically called. The first two

parameters of *DDJaccept*, *root*, and *timelimit*, were specified when the callback was defined in *socket*. Tcl automatically adds some additional parameters. The first additional parameter is the name of the socket that the program uses to communicate with the client.

The next line, *upvar #0 $socket request*, is a standard Tcl idiom that is used to manage the execution state for the various clients. In an event-based system, each client request needs to be uniquely identified so that its state can be associated with it. The socket name *$socket* serves this purpose: It is a unique value for any active client connection. Next, *upvar* creates a persistent (global) reference to this client's state, and gives it a local name -- *request*. Whenever a callback is registered for an event, *$socket* is included as a parameter so that the function can associate the right execution state with the socket.

The request array stores client-specific data and *array set request* initializes the state for this client request. The next line, *fconfigure*, sets the socket into nonblocking mode to ensure the server will never be stuck waiting on I/O.

Finally, two more callbacks are set up. The *fileevent* callback causes the function *DDJread* (Listing Three) to be called any time data is available to be read from a client. In general, not all of the data will be ready at once. The last callback sets a *watchdog* timer to fire after 60 seconds. This helps prevent denial-of-service attacks by preventing malicious clients from hogging socket connections. It also provides an opportunity to use time-based events. Both callbacks get passed *$socket* to make sure we get the proper execution state when the event happens.

Most of the server is fairly generic: It sets up the event handlers and creates the machinery for managing the execution state. The *DDJread* procedure contains all of the HTTP-specific handling code. It reads the request line, processes all of the HTTP headers, and reads the request body (the POST data, for instance). As this information is acquired, it is stuffed into the execution state of the client, using the request array.

In a nonevent-based server, the various parts of the request would be processed sequentially -- first the request line, then the headers, and finally the request body. The current part of the request being processed is implicit in the sequential nature of the code. In the event model, the execution state must be explicit. The *switch* block in *DDJread* uses the execution state to keep track of which part of a request is being dealt with. In actual operation, a header line might arrive from one client, followed by the initial request line from another, and a body from a third, seemingly out of order when viewed from the perspective of a sequential system. By representing the code as a state machine, the proper section of code is found to deal with the current request. Although the order of the arms of the *switch* statement is the same as in the sequential case, they could be ordered arbitrarily without affecting the operation.

Once the entire request has been processed, *DDJrespond* (Listing Four) is called to deliver the results to the client. *DDJrespond*, in a more general case, would be used to dispatch the request to the appropriate handler, based on the requested URL and request headers. In this simple example, I'll just dispatch files, although all of the information about the client request has been neatly packaged into the request array. When this procedure is called, the entire request has been processed, so the *filevent* and *watchdog* times for this request are canceled. The URL is converted to a filename, and if the file exists, it is sent to the client, along with the proper HTTP headers. Otherwise, a standard error message is returned.

The *fcopy* command allows the contents of the file to be transmitted to the client in the background, eliminating the need for another set of event handlers that would dole out pieces of the file to the client at the speed the client could accept them. When *fcopy* completes the data transfer, *DDJcopydone* (Listing Five) is called, and the connection to the client and the execution state are removed.

If, for some reason, the client is unable to complete the request in the allotted time, the timer event, set up when the client connection was first opened, expires, and *DDJtimeout* (Listing Six) is called. *DDJtimeout* closes the connection to the client, and removes the execution state associated with it.

The other functions in the server (see Listing Seven) consist of HTTP-specific utility routines that are needed to create a fully functional server, but have nothing to do, per se, with event-based programming.

Event-Based Limitations

Assuming the underlying support for nonblocking I/O, the chief disadvantage of an event-based system arises when the computation required to handle a single event takes too long. For example, if the web server had a built-in search capability that took 60 seconds of elapsed CPU time to complete, then no other clients would be serviced until the search was complete. The throughput would still be high, and the responses would still be returned in first-come/first-serve order, but in this case, that's not good enough. The easiest solution is to break the computation into smaller pieces, say 60 one-second

computations. After the first second's worth of computation is complete, the current state of the calculation is saved, and a timer event is scheduled using *after* to cause the next portion of the computation to commence after other client requests have had a chance to be serviced.

If the computation is not easily subdivided, or it is impossible to predict in advance how long it would take, then using threads for this portion of the system may be a good idea.

Events or Threads?

The decision to use events or threads for a particular application depends on several factors: throughput, overhead, and fairness. You should use events when:

- There is underlying support for nonblocking I/O.
- The time to process an event is small compared to the overall latency of the system, or the need for fairness isn't important.

Examples of applications that benefit from events include: Internet servers (HTTP for instance), where the user's expected response time is small compared to the time required to calculate the response; and GUIs, where the human response time for an input event, such as a button press, is long in comparison to the time needed to process that button.

You should consider using threads when:

- There is underlying support for threads in the OS and the required libraries.
- There are few data dependencies among the various threads, minimizing the need for synchronization locks.
- Multiple CPUs are needed for a particular application, and true concurrency is required.

Examples of systems where threads are appropriate include high-performance database servers, when true concurrency on multiple CPUs is required, and certain numerical calculations, where a single computation takes a long time, and there are few data dependencies.

There are many real systems with requirements that don't fit neatly into either category. In such cases, both techniques can be used in the same application. A GUI, written using events, could be interfaced to a thread-based simulation system.

Conclusion

Event-based programming presents a powerful alternative to threads in the construction of high-speed network servers. It is possible to obtain maximum throughput with the minimum overhead without the need for locks or the fear of data corruption. The built-in support for event-based I/O in Tcl, coupled with simple techniques to manage execution state, make it ideal for this type of application. In the server I present here, there is exactly one thread of execution -- no two event handlers are ever running at the same time, yet the performance and throughput is comparable to commercial servers.

References

Ousterhout, John. "Why Threads are a Bad Idea (For Most Purposes)." *1996 USENIX Conference Proceedings*; http://www.scriptics .com/people/john.ousterhout/threads.ps/.

Welch, Brent. *Practical Programming in Tcl and Tk*, Second Edition. Prentice Hall, 1997, ISBN 0-13-616830-2. http://www .beedub.com/book/.

Welch, Brent and Steve Uhler. "Web Enabling Applications." *1996 Tcl/Tk Conference* (abstract); http://www.scriptics.com/ tclhttpd/usenixAbstract.html.

DDJ

Listing One

```
  set port 8080     ;# the port to listen on
  set timeout 60000 ;# max seconds to wait for client request
  set root [pwd]    ;# our document root

  socket -server [list DDJaccept $root $timeout] $port
  vwait forever
```

Back to Article

Listing Two

```
proc DDJaccept {root timelimit socket ip args} {
    upvar #0 $socket request
    array set request [list State start Root $root Ip $ip]
    fconfigure $socket -block 0 -translation {auto crlf}
```

```
    fileevent $socket readable [list DDJread $socket]
    set request(Cancel) [after $timelimit [list DDJtimeout $socket]]
}
```

Back to Article

Listing Three

```
proc DDJread {socket} {
    upvar #0 $socket request
    # unexpected EOF - abort
    if {[eof $socket]} {
        puts stderr "$socket: Eof ([array get request])"
        close $socket
        after cancel $request(Cancel)
        unset request
    }
    switch $request(State) {
        start {         # Get HTTP request line
            gets $socket line
            if [regexp {(POST|GET|HEAD) ([^?]+)\??([^ ]*) HTTP/1.0} \
                    $line {} request(Proto) request(Url) request(Query)] {
                set request(State) headers
            } else {
                DDJerror $socket "400 Bad Request" "Invalid request:$line"
            }
        }
        headers {
            set count [gets $socket line]
            if {$count == 0} {  # end of headers
                catch {incr count $request(content-length)}
                if {$count > 0} {
                    fconfigure $socket -translation {binary crlf}
                    array set request [list data {} State body Count $count]
                } else {
                        DDJrespond $socket
                }
            } elseif {[regexp {([^:]+):[    ]*(.*)}  $line {} key value]} {
               set key [string tolower $key]
               if {[info exists request($key)]} {
               append request($key) ", " $value
               } else {
               set request($key) $value
               }
               set request(Key) $key
            } elseif {[regexp {^[   ]+(.*)} $line {} value]} {
               append request($request(Key)) " " $value
            } else {
               DDJerror $socket "400 Bad Request" "Invalid header:$line"
            }
        }
        body {
            append request(Body) [read $socket $request(Count)]
            set request(Count) [expr {$request(content-length) - \
                    [string length $request(Body)]}]
            if {$request(Count) == 0} {
                DDJrespond $socket
            }
        }
    }
}
```

Back to Article

Listing Four

```
proc DDJrespond {socket} {
    upvar #0 $socket request
    fileevent $socket readable {}
    after cancel $request(Cancel)
    set fileName [DDJurlToFile $request(Root) $request(Url)]
    if {[file isfile $fileName]} {
        append response [DDJheaders "200 data Follows" \
                [DDJcontentType $fileName] \
                [file size $fileName]] \
                "Last-Modified: [DDJdate [file mtime $fileName]]\n"
        puts $socket $response
        if {$request(Proto) != "HEAD"} {
            set in [open $fileName]
            fconfigure $socket -translation binary
            fconfigure $in -translation binary
            fcopy $in $socket -command [list DDJcopyDone $socket $in]
        } else {
            DDJcopyDone $sock ""
        }
    } else {
        DDJerror $socket "404 Not Found" "Can't find $request(Url)"
    }
}
```

Back to Article

Listing Five

```
proc DDJcopyDone {socket fd size} {
    upvar #0 $socket request
    puts stderr "[incr ::Ok] $request(Ip) $request(Url) ($size bytes)"
    catch {close $fd}
    close $socket
    # parray request
    unset request
}
```

Back to Article

Listing Six

```
proc DDJtimeout {socket} {
    upvar #0 $socket request
    puts stderr "$socket: timeout ([array get request])"
    close $socket
    unset request
}
```

Back to Article

Listing Seven

```
proc DDJcontentType {fileName}
  Looks at the Url suffix, and determines the HTTP document type
proc DDJheaders {code type length}
  Formats the standard set of HTTP headers used for replies.
proc DDJerror {socket code reason}
  Generates a standard error response
proc DDJdate {seconds}
  Formats an HTTP date string
proc DDJurlToFile {root Url}
  Converts a URL into a file name
proc DDJdecode {data}
  Converts any %xx codings embedded in a Url into the equivalent character
  representation
```

Back to Article

RELATED ARTICLES

CUDA, Supercomputing for the Masses: Part 9

Dr. Dobb's Agile Update 10/08

Understanding Parallel Performance

Complex Requirements On an Agile Project

My Platform

TOP 5 ARTICLES

No Top Articles.

RSS | All Feeds