

## Re: Socket Programming

**Source:** <http://coding.derkeiler.com/Archive/Lisp/comp.lang.lisp/2004-05/2583.html>

---

**From:** David Steuber ([david\\_at\\_david-steuber.com](mailto:david_at_david-steuber.com))

**Date:** 05/31/04

Date: 31 May 2004 10:43:26 -0400

Peter Seibel <[peter@javamonkey.com](mailto:peter@javamonkey.com)> writes:

> *David Steuber <david@david-steuber.com> writes:*

>

> > *Peter Seibel <peter@javamonkey.com> writes:*

> >

> > > *As I understand it the point of the C10K page is that at the OS  
> > > level--at least in mainstream OS's--there's no good solution to the  
> > > problem of keeping track of which of 10,000 (let alone 100,000)  
> > > connections may have data to deal with. Unless I'm misunderstanding  
> > > them there's little that better languages or ways of thinking about  
> > > writing concurrent software can do to fix *that* particular problem.*

> >

> > *I don't think you need to worry about the 100,000 simultaneous  
> > connection problem. IPv4 supports only 2<sup>16</sup> ports. Even if a process  
> > could get 100,000 file descriptors, there would be no way for one  
> > machine to have that many open IP connections. This is actually a  
> > major reason for the HTTP protocol vs FTP where a typical session is  
> > much longer. IP connections are a very limited resource.*

>

> *That doesn't matter to the server. Every connection to a server is to  
> the same port anyway (typically). It's the client IP-address/port that  
> makes the connection unique. So while it's true that I can only have  
> (expt 2 16) connections from a single client IP address, I can have  
> many, many connections from many, many clients.*

Another poster also corrected my misconceptions about sockets. The reason I thought there was a limit on server connections was the way I thought accept worked. I don't recall how I got this idea, but I thought that when a connection to port 80 was accepted, a separate port was used for the transaction so that the server could continue to listen on port 80.

I'm perfectly happy to be wrong on that account. It means I can settle for cheaper hardware :-)

> > *What I got from the C10K page was that the primary problem with one  
> > thread per connection was running out of stack space. If incoming*

## comp.lang.lisp: Re: Socket Programming

> > *data causes ALL the threads to wake up and then the ones that should  
> > be idle to go back to sleep (thundering herd), then that would be a  
> > real problem as well. If you can say ahead of time what the maximum  
> > stack space you will need is, you should have a way to tell the OS not  
> > to give you more than that.*  
> >  
> > *However, either way you cut it, time slicing has to be done  
> > somewhere. Polling is expensive because you are querying descriptors  
> > that are not ready. The other two methods that were mentioned in the  
> > article seem like reasonable fixes, but you would still have to do  
> > some sort of processing on idle connections.*  
>  
> *Why? The kernel knows when packets arrive on the wire as it is  
> responsible for decoding them.*

I'm talking about polling buy the user code. It's kinda like the kids in the back seat asking, "are we there yet?"

Something driven by an event from the kernel will obviously work better because it just sits passively until kicked.

> > *Proper 1:1 threading should work nicely. The ideal case (see sig) is  
> > that data comes in on the ether and the kernel knows exactly which  
> > thread to wake up to read it. If the kernel can handle that detail,  
> > then things are much easier for developers in userland.*  
>  
> *Yes. That's called asynchronous i/o. When you issue the i/o call, you  
> pass along a callback that should be run when the i/o completes. So if  
> I do a read on a socket, I pass along the code that I want to run when  
> there is actually data to read. Nothing happens until some data shows  
> up for that connection.*  
>  
> *Which the kernel \*does\* know since somewhere down in the TCP stack in  
> decoded the incoming packet and figured out which connection it  
> belongs to, etc. Of course I'd also like to keep a thread-like notion  
> of dynamic context so my special variables, condition handlers, and  
> restarts are still in place.*

Ok, asynchronous i/o sounds good from an event driven perspective. Using a call back sounds very hairy.

When I did programming for Win16, it was all event driven. Each event was fetched in a message loop. The Win16 API call getMessage (or something like that, it's been a while and I suck with remembering names) would block until a message came in. (There was also a peekMessage call that would return immediatly, so you could do background processing without setting up a timer).

The only problem with that model was there was no concurrency. If you took ten seconds to process a message, that was ten seconds that the entire system was frozen. Win32 improved things somewhat to making it

so that only your application was frozen unless you kicked off a thread to handle the event so that you could go back to your event loop as quickly as possible.

- > *Actually the question of what the Ultimate Right Thing is is*
- > *interesting to me. I see it as perhaps analogous to an argument I*
- > *used to have on a regular basis with one of the other developers at my*
- > *last gig: whether a general purpose garbage collector had to \*in*
- > *principle\* less efficient (in computer resources) than manually*
- > *managing memory. I had two arguments in favor of GC. The "weak"*
- > *argument was that while it might be possible to actually manage memory*
- > *more efficiently without GC (by taking advantage of*
- > *application-specific knowledge of memory usage patterns) that in*
- > *practice it would be so much harder to outperform a good GC*
- > *implementation that the programmer using GC would have way more time*
- > *to improve the overall efficiency of their program through better*
- > *design that it would always outperform a program written using manual*
- > *memory management. My "strong" argument was that memory management is*
- > *actually complex enough (e.g. you have to consider locality issues,*
- > *etc.) that ultimately a good GC implementation would do better on any*
- > *reasonably complex application than a human could. (I.e. the same*
- > *reason I want my compiler to allocate registers for me.)*

In the general case I agree with you here. There are some things the compiler should be able to figure out (or the runtime) so that consing and gc are as cheap as possible. Lisp and the various functional languages should do very well with this because information on memory usage is available to the system. Perhaps more than there is available to the programmer. I'm a little less convinced with Java, but then it could just be that I've been using an immature product.

One thing in Lisp that interests me is dynamic-extent. When you rebind a special variable in a let, you know at compile time that you have dynamic-extent and may be able to use the stack for allocation just like C automatic variables. That means no allocation at all. You don't get much faster than that. I'm not sure what happens when you try to close over a special variable that has been dynamically-bound. I would expect the specialness to mean that you refer to the previous value of the variable when you leave the lexical scope of the binding. I'll have to reread that bit in the CLHS.

Other lexicals established by let should also effectively have dynamic-extent. If they are closed over or returned from the let, then you know they have indefinite extent. But if that does not happen, and this should be known at compile time, then stack allocation can be used again.

I may be wrong on the details and would be happy for corrections. Bottom line is that the compiler should be able to do a very good job of managing memory for you. It should even be able to tell when you have a large chunk of static memory that you treat as read only or

just do in place modifications to.

*> So back to threads, there's part of me that would like to believe that  
> ultimately general purpose threading systems will be so good that the  
> path to the most efficient program simply will be to spin up as many  
> threads as you want and let the threading system take care of it.  
> Analogues to my weak and strong arguments for GC can be made. But I  
> don't believe that current threading systems have achieved that goal  
> yet. Not being an OS or virtual machine researcher, I'm not sure what  
> the open questions are or how near a solution we are. Thus I'm hoping  
> Luke--who claimed writing super-scalable software is a solved  
> problem--will answer my other post.*

The only problems I see have to do with allocating a stack for the thread and dealing with shared memory. The stacks can eat up your virtual address space very quickly. The only way I can think to deal with that is to not use overly large stacks. If you have good memory management, you can put the big objects (even if they are short lived) on the heap so that stack space doesn't need to grow in some unbounded way. I don't know all the gotchas that are there. Running out of stack space is not a good thing though.

Shared memory is much easier to deal with. Fast mutexes seem to be the answer there. I'm sure there are other tricks as well. Rebinding a special in a let should give you a thread local storage of sorts. A functional programming style will reduce the need for shared memory. In anycase, serialization is something to avoid where possible because that defeats the whole point of having threads.

What I like about threads is that I can, for the most part, treat each thread like a batch process. I find cooperative multi-tasking to be much more painful than having a preemptive system in place. I've been down the cooperative road in the past with C and C++. I hate that road.

Perhaps if everyone said to hell with runtime efficiency, it's easier to program with 1:1 threads, the OS support will develop and mature to support that style so that the hardware can be properly utilized.

It sounds a lot like threading and automatic memory management go hand in hand.

--

An ideal world is left as an exercise to the reader.  
--- Paul Graham, On Lisp 8.1