Developing High Performance Socket Applications

Internet and Intranet Applications and Protocols

March 28, 2006

Joe Conron
What is “high performance”?

• High availability
• More messages per second
• Shorter response time
• High bandwidth efficiency
• Low resource usage
When is a service “unavailable?”

• When it fails
  – Uh Oh, another bug!

• When it needs maintenance
  – Another Microsoft patch needs to go in!

• When service cannot keep pace with rate of requests for service
  – Too many users
  – Message arrival rate exceeds service rate
How to Achieve High Availability?

• Minimize system outage due to bugs by doing Excellent Testing.

• **Good design** - partition and distribute function
  – Isolates systems that need high levels of maintenance
  – Makes patching easier
  – Improves fail-over strategy.

• Use **stateless** approach whenever possible

• Provide **redundancy**

• Provide high levels of **concurrency**
Design and Programming of Socket Applications

• We need a **model** – hard to discuss any problem without a model.

• Two kinds of socket application models
  – **Request/Response**: client server applications like web server and web browser
  – **Data Streaming**: data collection or data distribution apps like news, financial data, instrumentation data, multimedia, etc.
Request Response Model

Client Request

T1

Server Processes Request

T2

Server Sends Response

T3

T1 ::= Time for request to travel from Client Application code to Server application code
T2 ::= Time for Server application code to process request and generate response
T3 ::= Time for response to travel from Server Application code to Client application code

Our goal: Minimize T1, T2, and T3
Data Streaming Model

Our goal: minimize $T_1$ and $T_2$ and handle small values of $\lambda$
Socket “Internals”

• Whether using **TCP** or **UDP**, the transport layer driver/handlers have Receive and Send buffers.

• Recall **TCP flow control** – what happens if not enough room in receive buffer?

• Not so obvious – what happens if not enough room in the **send buffer**?

• Even less obvious, what happens if **UDP** receive buffer is full?
Answers: Nothing Good!

• TCP:
  – if receive buffer is full, the other side must stop sending.
  – If send buffer is full, application is blocked on a write call (unless non-blocking I/O)

• UDP
  – If receive buffer is full it silently drops all arriving packets
  – What if send buffer is full?
Setting Socket Buffer Sizes

• You can increase the size of the TCP and UDP Socket send and receive buffers
  • setSendBufferSize(int)
  • setReceiveBufferSize(int)
• The default size depends on the OS
  – Windows  16kB
  – Solaris  48kB
• How big can you make it?
TCP Delayed Sending

• TCP by default will **delay** transmission of a partially full buffer until an ACK for previous transmission is received.

  – WHY is this a good idea??

• So, if you want to improve response time (decrease latency), disable TCP delay feature:

  setTCPNoDelay(boolean)

• What is the effect on number of transmitted TCP segments if you **setTCPNoDelay(true)**?
Improving Application Performance

• Avoid unproductive work
• Avoid live-lock
• Avoid deadlock
  – But also avoid race conditions!
• Control debilitating effects of Garbage Collection via pre-allocation of Objects
• Use in-line code rather than loops
• Use UDP in rather than TCP when possible
Avoid Unproductive Work

• Unproductive work is any processing that does not result in progress.

• Example:
  – accept a message, allocate space for it, parse it, then find out there are insufficient resources to further process the message, so you discard the message.

• Use short-circuit processing whenever possible.
Short-Circuit Processing

- Short circuit logic is any rule or rules that you can apply to cease processing of any request because:
  - You determine that you cannot possibly compute an answer.
  - You know a priori that you have insufficient resources to produce an answer.
  - In streaming applications, you can determine that the received message is not of interest.
Examples

• You receive a request type that is not supported.
  – Normally the “application” layer makes that decision. Lower layers – socket read thread for example – simply read messages and pass up to higher layers.
  – What if lower layer “knew” how to quickly locate request type and had a list of valid request types? What good things can we do?
Examples

• A queue with flow control is an excellent way for a higher layer to indicate to lower layer that it is too busy (out of resources) to process new requests.

• Higher level sets “Q Full” condition

• Lower layer will only process a new message if ~(Q Full)

• When upper layer “congestion” eases, it resets Q Full indicator
Avoid Live-lock

• In communications applications, live-lock is caused by arrival of events at one Thread at such a high rate that the system spends all of it’s time handling only these events, leaving no time for any other processing.

• Typically caused by naïve design
How to avoid live-lock

• Uncontrolled “lively” Threads should periodically yield().

• Control “lively” Threads via resource based “rate control”
  – Lively Thread can only run when it has resources, else it blocks waiting for resource
  – Other Threads – typically higher layer functions, pass resources to lively Thread as they make progress.
Example

• Lower layer needs a buffer to receive a new message from Socket.
• Upper layer provides buffers to lower layer as if processes a message and hence “frees up” the buffer.
• Requires use of a pre-allocated buffer pool.
Example

- Typically, socket receive Threads sit in a “tight” loop:

```
while(true)
{
    buf = new whatever()
    read(buf)
    upperLayerQueue.add(buf)
}
```

**LiveLock?**

```
while(true)
{
    for (int i=0; i < yield; i++)
    {
        buf = new whatever()
        read(buf)
        upperLayerQueue.add(buf)
    }
    Thread.yield();
}
```

**Better?**
Avoid Deadlock

• Deadlock occurs when a system makes no progress because Thread A wants a resource held by B and B wants a resource held by A.

• Easy to avoid
  – Rule: if any process needs resources r1, r2, … then always acquire in order r1, r2, r3
  – If holding r1, r2, r3 always release in order r3, r2, r1
  – Same holds for subsets of a resource sequence (e.g., r1, r2)
  – Can you prove that it works for any number of contending Threads?
Race Conditions

• Race conditions occur when two or more Threads perform some sequence of read or write operations on the same data set.

• Example:
  – Iterate over items in a Hashtable
  – Another Thread removes items
  – (even though Hashtable is synchronized, you still have a problem!)
Avoiding Race Conditions

• Use synchronize blocks
• DO NOT overuse!
  – Using sync blocks when there is no contention or no undesirable side-effects of contention is a terrible waste of time!
• Better Approach
  – Build synchronization into your objects rather than depending on good will!
Garbage Collection
“GC is a thief in the night”

• One of the consequences of a very busy dynamic system is that it creates many new objects whose life is short.

• A heavily loaded system ( 20K – 100K messages/sec) can spend as much as 30 seconds with all application Threads suspended while a full GC runs.

• This is poison to high performance system
Garbage Collection

• Recall the “tight loop” socket reader model

```java
while(true)
{
    buf = new whatever()
    read(buf)
    upperLayerQueue.add(buf)
}
```

• What happens to the “whatever” Objects?
Garbage Collection

• The whatever Objects hang around until there are no more references to it, but that could be long enough for the Object to be moved from the “Eden” into a “survivor” space.

• Once that happens, it will take more than a “minor” GC cycle to detect a free Object

• So, we should try to minimize the frequency with which we allocate new Objects.

• How can we do that and still meet our processing demands?
Use Object Pools

• Using Queuing Theory, we should be able to predict how many buffers we will need to handle a given $\lambda$ and $\mu$
• Pre-allocate the required number of Objects into a “pool” class (or Factory class)
• Get objects from the pool to perform new service (for example, to read another message from the socket)
• Return Objects to the pool when finished (upper layer has processed message)
• Memory is cheap: failure to perform is expensive
Object Pools: Threads

• A typical TCP Server application has a “listener” Thread that accepts new connections on the ServerSocket and allocates a new client Socket and Thread to process a request (HTTP server for example)

• Same problem as before: as request frequency increases, so does number of Threads. While not so bad for GC, very bad for scheduler!
  – Create and Destroy of Threads is expensive

• Better approach: use Thread Pools or combination of Thread pool and asynchronous I/O (NIO)
  – Topic for another lecture!
Loops

Which code runs faster? (search an array of size 3 for a match on String x)

```java
for (int i=1; i < array.length; i++)
{
    if (array[i].equals(x))
    {
        return true;
    }
}
return false;
```

```java
if (array[0].equals(x))
    return true;
if (array[1].equals(x))
    return true;
if (array[2].equals(x))
    return true;
return false;
```
Use UDP When Possible

• Clearly UDP is “faster” transport protocol than TCP
  – No flow control
  – No congestion control

• But UDP is unreliable transport!
  – What does that mean?
  – What network element drops IP packets?
  – Why?
Useful Datagram Protocol

• Local networks (LAN) typically do not have routers between nodes on the same segment.
• So, who will drop IP packets?
• If IP packets aren’t dropped, then we have a reliable “network”
• Is that enough?
  – Do we need congestion control?
  – Flow control?
UDP

• Suppose we want to put an HTTP server on the LAN to serve some application specific content.
• If we use UDP rather than TCP, how many sockets would the server have to allocate for 100 concurrent requests?
• 1,000 concurrent requests?
• Is this a good thing?
• How many Threads would we need to read new requests?
• How many Threads to write responses? (assume unlimited bandwidth)
UDP

• UDP has another advantage over TCP for many applications:
  – Preserves message boundaries

• Catch 22
  – I said earlier in the presentation that TCP delays sending to improve efficiency
  – If I use UDP, won’t it increase the number of IP packets transmitted?
  – The higher the message rate, the more likely it is that the number of packets will increase if we use UDP.
You can implement message batching
Collect messages in a buffer.
When first message is placed in buffer, start a timer.
If timer goes off or buffer cannot hold any new messages, write the buffer to the Socket.
UDP Needs Fragmentation Handler

• Maximum UDP Datagram is about 64K Bytes.
• Up to you to build in a message structure within the Datagram similar to IP Fragmentation mechanism.
• Not hard – worth doing to use UDP
UDP for Data Streaming

• UDP is best for high speed data streaming
• We are seeing a dramatic increase in the use of UDP (IP Multicast) to deliver real-time financial information.
• Partition data stream by “data type” (OTC equities, FX, commodities, etc).
• Allocate an IP Multicast Channel for each data stream.
• Allocate a redundant secondary channel for each data stream and transmit parallel streams from two different end systems.
• Provide TCP based repair service
  – If receiver sees gap in datagram sequence, connect to repair service and request missing datagrams.
Summary

• Achieve high performance by
  – Good design (another course!)
  – Use pools (factories) of pre-allocated Object wherever possible
  – Use Short Circuit logic at every opportunity
  – Use UDP whenever possible
  – Be aware of GC overhead
    • Read about jconsole in JAVA 5