

Minimizing Messaging Latency

Copyright © 2005 - 2006 29West, Inc.
March 15, 2006

Table of Contents

1. Introduction.....	1
2. Message Batching.....	1
3. Congestion Control	2
4. Flow Control.....	3
5. Kernel/User Transitions.....	3
6. Message Routing	4
7. Ordered Delivery.....	5
8. Receiver Process/Thread Switching	6
9. Further Reading.....	6

This white paper is a practical guide to various design and implementation techniques to minimize messaging latency in distributed applications, and how these techniques can be used with the 29West (<http://www.29West.Com/>) Latency Busters® Messaging (**LBM**) product.

1. Introduction

Everybody wants their distributed applications to use a fast, efficient form of messaging. After all, if your computer had a "fast/slow" switch, would you ever set it to "slow"? (Don't laugh! (<http://29west.com/links/turbo.html>)) But there are some applications that *need* high performance messaging to be successful. These applications generally emphasize one of two different measures of performance: throughput (total bits per second that can be sustained) and latency (elapsed time between the sending and subsequent reception of a message). In the latter case, even an extra millisecond of latency can be a serious blow for these *latency-sensitive* applications.

This white paper is intended to help designers and implementers of latency-sensitive systems minimize their messaging latency.

There are many techniques that can be used to reduce message latency. However, as you will see, some techniques can have negative side-effects on other characteristics of a system. For example, sending smaller packets can reduce latency but can also result in greater CPU and network load. These sorts of tradeoffs are inevitable and need to be taken into account when designing a system, choosing hardware, and allocating resources. Ideally, a messaging system allows not only the selection of many of these latency-reducing techniques, but also allows fine-grained adjustment of their operating parameters so that the tradeoffs can be optimally balanced for a given system.

2. Message Batching

Many messaging implementations are written to minimize network overhead and maximize overall throughput by batching together short application messages and sending several of them in a single network packet. Network bandwidth is best conserved when packets are consistently at or near the MTU (Maximum Transmission Unit). The TCP protocol uses Nagel's algorithm (http://en.wikipedia.org/wiki/Nagle%27s_algorithm) to do this batching, and many other messaging systems use similar methods.

However, this batching introduces delays in the transmission of application messages, which can be undesirable for latency-sensitive applications. Some system designers want to eliminate this batching completely. For TCP-oriented messaging, it can be done with socket option `TCP_NODELAY`, with the result that even very short messages are sent immediately, each in its own packet. Alternatively, UDP-oriented messaging can be used (UDP does not include any batching in the protocol stack). These methods eliminate the Nagel-like delays, but also reduce efficiency, increasing both the CPU load and network bandwidth usage. The result is a reduction in latency with a potential increase hardware costs (both computer and network) to achieve a desired overall throughput.

There is a case where batching can lead to a *improvement* in average message latency. If the application has several messages that it wants to send, combining them into a single network packet can save a significant amount of overhead, resulting in a decreased average latency for the group of messages. However, since most batching algorithms use a timer to trigger the transmission of the last bit of data that doesn't fill a packet, those algorithms tend to erase the latency savings made through the reduction in overhead.

The 29West **LBM** product allows batch transmission to be based on both time and packet size, or batching can be turned off. Moreover, LBM allows batching to be configured on a transport session basis, so that the same application can reduce or eliminate batching for latency-sensitive data while allowing less-sensitive data to be batched, increasing both network and computer efficiency. Finally, LBM supports explicit batching, allowing the application to delimit groups of messages that can be combined while forcing the last message to be sent immediately (instead of by a timer).

3. Congestion Control

Network congestion can happen when multiple applications have to compete for limited network bandwidth. As network utilization approaches 100%, data is delayed in queues and can be lost in switches and routers, thus requiring re-transmission. In its most severe form, a network can suffer *congestion collapse*, a self-reinforcing condition where more and more bandwidth is consumed by applications trying desperately to recover delayed or lost data. Reliable multicast protocols (like PGM) can sometimes contribute to congestion collapse by generating NAK storms (a.k.a. NAK implosion). We've seen cases where network throughput drops to near zero, with virtually all of the bandwidth consumed by NAKs and retransmitted data.

To avoid congestion collapse, protocols need to be designed to react to congestion in ways that don't make the problem worse. For example, the TCP protocol uses a variety of algorithms which basically slow down the sending application when congestion is detected. TCP is designed to spread the slowdown fairly evenly across all competing connections, thus giving each connection an equal share of the available network bandwidth.

Designers of latency-sensitive applications often want to avoid this "equal sharing". They want better control over how network bandwidth is allocated to applications so that highly time-critical data can get through quickly, albeit somewhat at the expense of less time-sensitive data. Many designers choose a UDP-based messaging system (unicast or multicast) to deliver their latency-sensitive data because UDP does not automatically slow down in the face of congestion. Unfortunately, as many designers have discovered, UDP-based messaging can cause instability in a

congested network, sometimes leading to congestion collapse. Since network bandwidth is a finite resource, some measures need to be taken to maintain stability when there are more time-sensitive messages to send than can fit.

The 29West **LBM** product addresses this requirement through the use of rate limits. LBM's reliable multicast is immune to NAK storms due to its internal NAK-generation algorithms plus the ability to limit the rate of re-transmitted data independently from new data. These rate limits are critical to maintaining network stability in the face of congestion, while still giving the designer the ability to allocate bandwidth according to the relative time-sensitivity of the data being sent.

4. Flow Control

In a smoothly-running system, receivers of latency-sensitive data are designed to be able to keep up with the expected average data rate. Short-term bursts of traffic may exceed the receiver's ability to keep up, but the data is buffered so that the receiver can catch up when the burst is over. These buffers allow the sender to transmit data as it becomes available without worrying about the receiver's ability to keep up. (Note that for latency-sensitive applications, it is usually undesirable for these buffers to be too large. These applications generally prefer that old "stale" data is discarded rather than saved for long periods of time.)

However, even a very well-designed system can get into conditions where one or more receivers get further and further behind. If the condition is temporary and the application not latency-sensitive, large buffers can be used to maintain data continuity for slow receivers. But no matter how big buffers are made, a design decision needs to be made to determine what to do if and when the buffers finally do fill up.

Many messaging systems, including those based on TCP, use sender flow control when buffers fill up, causing the sending application to be blocked, preventing it from sending at its natural rate. This can lead to overall system problems when only a few receivers are having trouble keeping up (the *crybaby receiver* (<http://www.29west.com/docs/THPM/thpm.html#CRYBABY-RECEIVER>) problem). In a system that flow-controls the sender, *all* receivers suffer from the slowdown of a few. This is desirable behavior for systems which require that *every* receiver must get *all* messages. However, designers of latency-sensitive systems usually prefer that an overly-slow receiver experience data loss rather than slow down the whole group.

The 29West **LBM** product supports a TCP-based transport for those applications that require sender flow-control, and two UDP-based transports for applications that do not. Buffering is done on both the sender and the receiver to provide full data integrity under normal conditions. If a receiver slows down for too long and the buffers overflow, the receiver is informed that unrecoverable loss has happened. The designer can control the amount of buffering done to strike a good balance between burst recovery and stale data delivery.

5. Kernel/User Transitions

In modern operating systems, network packets are transferred to and from the hardware interface by the operating system kernel. Since applications typically run in user space, a transition between user and kernel space is necessary to get message data into and out of the machine. The overhead involved in making this transition can be significant, introducing both message latency and CPU load. Designers of high-performance applications generally try to minimize the number of user/kernel transitions needed to get a message from a sending application to a receiving application.

One fairly easy way to minimize user/kernel transitions is to avoid making multiple system calls per message. For example, an application message may contain an application header followed by a data payload. It may be

convenient for the programmer to make two system calls to send the message - first to send the header and then to send the payload. This is especially true if different parts of the code are responsible for header and data generation. Similarly, the receiving application may find it useful to first read the header, perhaps to get the length of the rest of the message, followed by a second read of the proper number of bytes. However, for high-performance applications, the header and payload should be combined by the application and given to the kernel with a single system call. The receiving application should create a receive buffer large enough for the largest possible message and do a single read. These steps improve both CPU and network efficiency, leading to greater overall throughput.

When senders and receivers are on different machines, a given message must cross the user/kernel boundary a minimum of two times. Many high-performance applications seek to further improve efficiency using batching to combine multiple application messages and send them to the kernel as a single buffer. This can reduce the *average* number of kernel/user transitions per message to *less* than one. However, for latency-sensitive applications, message batching is often undesirable since it can introduce more latency than the user/kernel transitions it eliminates (see Section 2 for exceptions).

Server/daemon-based messaging systems introduce additional user/kernel transitions. For example, TIBCO® **RV**TM requires that a sending application send a message first to a daemon process on the same machine, then to a corresponding daemon on the receiving machine, before it is sent to the receiving application. If batching is suppressed, this leads to a minimum of six user/kernel transitions per message. TIBCO **SmartSockets**TM, as well as most JMS-based systems, don't use daemon processes, but do use a central server, leading to a minimum of four user/kernel transitions per message.

When using messaging systems that rely on daemons and/or servers, it may be useful to utilize some form of batching and experimentally determine what level of batching provides the smallest average latency.

The 29West **LBM** product buffers entire application messages to minimize the number of operating system write calls. In addition, it does not use daemons or central servers; messages are sent directly from sending application to receiving application. These measures lead to the best-case of two user/kernel transitions per message when batching is turned off. (For designers who need to emphasize throughput and efficiency over latency, batching can be enabled to further reduce the user/kernel transitions, often to less than one per message.)

6. Message Routing

Many application developers want the power and flexibility of a generalized topic-based publish/subscribe messaging paradigm ("pub/sub"). Senders of data do not have to know how many or where the receivers are, and receivers do not have to know how many or where the senders are. The messaging system is responsible for figuring out where messages should go.

One common method of accomplishing this is to have a centralized server with which senders and receivers are required to register. In most server-based messaging systems, that centralized server actually performs the routing of each data message (e.g. TIBCO **SmartSockets**, Sonic **JMS**, etc.). This application-level message routing can be simple to implement, but for latency-sensitive applications it often introduces unacceptable delay since each message needs to be received, processed, and forwarded to the final destination.

One common method for avoiding these delays is to use UDP multicast as the message transport. This tends to fit well with the pub/sub paradigm since a message need only be sent once onto a given network and all interested machines will receive the message. However, it has only been relatively recently that multicast-enabled routers have been widely deployed. Before then, messaging systems needed to use application-level gateways to get multicast messages from one network to another (e.g. TIBCO **RV**). Once again, undesirable latency is introduced by these gateways.

These days it is generally better to make use of multicast-enabled routers to get messages where they belong. Modern routers can do this routing at wire speed, eliminating the user/kernel transitions of application-level servers or gateways. Some designers run into resistance from network administrators to enable multicast routing because it requires greater care and learning to design, configure, and operate the networks. For example, since multicast traffic does not have automatic congestion control built into the kernel, a lack of careful design can lead to severe congestion on the network. When multicast routing is enabled, that congestion can spill over onto all the networks of an organization. However, for latency-sensitive applications, the added cost and effort of careful design and operation can pay large benefits.

The 29West **LBM** product normally does not make use of application-level servers or gateways, relying instead on modern network hardware to route messages. For organizations that are not willing or able to enable multicast routing, **LBM** can make use of either TCP or unicast UDP for message transport. However, **LBM**'s rate limits provide very good mechanisms to prevent severe congestion, removing some of the resistance to enabling multicast routing. In addition, **LBM** allows the designer flexibility in segmenting topic spaces across different multicast addresses, allowing only that traffic that needs to be communication across networks to be routed. Finally, for those cases when multicast routing is inappropriate, a gateway function is currently under development to interconnect multiple multicast networks over lower-speed WAN links.

7. Ordered Delivery

Modern networks are very reliable, and dropped packets happen far less often than they used to. However, most messaging systems still need to deal with dropped packets and arrange for their retransmission. That obviously introduces latency in the reception of that particular packet (waiting for its retransmission), but in most systems the delivery of packets subsequent to that lost packet are likewise held up. Buffering of successfully-received packets after a loss, and delay of delivery until successful re-transmission of the lost packet is a characteristic of virtually all modern messaging systems. It is needed to make the message delivery order the same as the message transmission order (a requirement for many applications).

To minimize the impact of ordered delivery delay after packet loss, a designer needs to minimize the frequency of lost packets. The most effective way of minimizing packet loss is to prevent network congestion. One way to do this is to use TCP as the message transport. TCP uses very effective algorithms to detect and control network congestion. However, Latency-sensitive applications are often better served by other techniques, such as the 29West **LBM** product's multicast rate limits (see Section 3 for more on congestion).

Another technique for reducing average latency is to segment application data across several topics and use a separate transport session for each topic (where a session might correspond to a TCP connection, a UDP port, or a multicast address). A packet loss in one transport session will only add latency to that transport session. However, this can become impractical if the number of desired topics becomes very large (e.g. a separate topic per stock symbol).

The 29West **LBM** product introduces a new design for UDP-based transports where multiple topics can be mapped to a particular transport session, but order is not maintained across the whole transport session. Instead, each topic is ordered independently from other topics. Thus, a packet lost on a transport session will only add latency to the affected topic(s), while other topics on the same transport session will be unaffected.

For those applications that can tolerate out-of-order messaging, **LBM** also makes available arrival-order delivery. A lost packet will not delay the delivery of subsequently-received messages of the same topic. Instead, the application will continue to receive the latest data while **LBM** requests retransmission in the background. The application will receive the retransmitted data when it becomes available. (Note - as will be discussed below, there are other potential

benefits to designing applications to be able to handle out-of-order data. We often encourage designers to at least give the option some thought.)

8. Receiver Process/Thread Switching

Most messaging systems make use of an independent process or thread that is "owned" by the messaging system. It is used to do housekeeping (like timer management), is usually used to receive each incoming network packet, and often is used to send each outgoing network packet. The actual application typically runs in a separate set of cooperating threads and/or processes. The switch between the messaging thread/process and the application thread/process is another source of message latency.

To minimize latency, it can be useful to design the messaging system to call an application function directly from the messaging layer when a message is received, without requiring a thread/process switch. If the application function is able to fully process the message quickly, it can do so and immediately return to the messaging layer, resulting in the message being received and consumed without any thread switching at all. However, in systems with that design, care must be taken to prevent the application function delaying its return for too long. For example, if the application function needs to block, then the entire messaging system will be brought to a halt, introducing latency across all topics and transport sessions. If lengthy processing is required, it is usually better for that processing to be done in an application thread.

The 29West **LBM** product typically requires no thread switching at all to send messages; the application thread calls the send function, which performs the LBM processing and eventually calls the socket write directly. LBM also allows application callback for received messages to be processed from the **LBM** thread. For applications that require non-trivial processing of received messages, **LBM** also provides thread-safe event queues so that the application callback on received messages is done from an application thread. Note that the programming paradigm is the same - in both cases an application callback is the method for message delivery. This makes it easy to try both methods and choose the one that performs more appropriately for your specific application.

Also note that **LBM**'s event queues are fully thread-safe. One method for reducing average message latency (especially during traffic bursts) is to have multiple application "worker" threads dispatching a single event queue. For multi-CPU machines, this can allow received messages to be processed with true parallelism, reducing the time required to process a group of messages, thus reducing the average latency. Especially when a non-trivial amount of processing needs to be done on received messages, this average latency reduction can far outweigh the cost of the thread switch from **LBM** to application. However, as with arrival-order delivery discussed above, parallel processing of incoming messages can introduce complexity in the design of the application, especially as it relates to message processing order. Messages will be *dispatched* in the order they are received, but the operating system may schedule threads to run on available processors in an unexpected order, with the result that messages are finished being processed in a different order. For applications that can tolerate this kind of ordering, the benefits of reduced average latency and increased throughput can be considerable (on multi-CPU hardware).

9. Further Reading

Our Topics in High-Performance Messaging (<http://www.29west.com/docs/THPM>) white paper has more detailed information on TCP latency (<http://www.29west.com/docs/THPM/thpm.html#TCP-LATENCY>), latency budgeting (<http://www.29west.com/docs/THPM/thpm.html#MESSAGING-LATENCY-BUDGET>), and sources of latency (<http://www.29west.com/docs/THPM/thpm.html#LATENCY-SOURCES>).