Evolution and Practice: Low-latency Distributed Applications in Finance

The finance industry has unique demands for low-latency distributed systems

Andrew Brook

Virtually all systems have some requirements for latency, defined here as the time required for a system to respond to input. (Non-halting computations exist, but they have few practical applications.) Latency requirements appear in problem domains as diverse as aircraft flight controls (http://copter.ardupilot.com/), voice communications (http://queue.acm.org/detail.cfm?id=1028895), multiplayer gaming (http://queue.acm.org/detail.cfm?id=971591), online advertising (http://acuityads.com/real-time-bidding/), and scientific experiments (http://home.web.cern.ch/about/accelerators/cern-neutrinos-gran-sasso).

Distributed systems—in which computation occurs on multiple networked computers that communicate and coordinate their actions by passing messages—present special latency considerations. In recent years the automation of financial trading has driven requirements for distributed systems with challenging latency requirements (often measured in microseconds or even nanoseconds; see table 1) and global geographic distribution. Automated trading provides a window into the engineering challenges of ever-shrinking latency requirements, which may be useful to software engineers in other fields.

This article focuses on applications where latency (as opposed to throughput, efficiency, or some other metric) is one of the primary design considerations. Phrased differently, "low-latency systems" are those for which latency is the main measure of success and is usually the toughest constraint to design around. The article presents examples of low-latency systems that illustrate the external factors that drive latency and then discusses some practical engineering approaches to building systems that operate at low latency.

WHY IS EVERYONE IN SUCH A HURRY?

To understand the impact of latency on an application, it's important first to understand the external, real-world factors that drive the requirement. The following examples from the finance industry illustrate the impact of some real-world factors.

REQUEST FOR QUOTE TRADING

In 2003 I worked at a large bank that had just deployed a new Web-based institutional foreign-currency trading system. The quote and trade engine, a J2EE (Java 2 Platform, Enterprise Edition)

TABLE 1 **Units of time**

Millisecond = 10^-3 seconds	1,000 milliseconds = 1 second
Microsecond = 10^-6 seconds	1,000 microseconds = 1 millisecond
Nanosecond = 10^-9 seconds	1,000 nanoseconds = 1 microsecond

application running in a WebLogic server on top of an Oracle database, had response times that were reliably under two seconds—fast enough to ensure good user experience.

Around the same time that the bank's Web site went live, a multibank online trading platform was launched. On this new platform, a client would submit an RFQ (request for quote) that would be forwarded to multiple participating banks. Each bank would respond with a quote, and the client would choose which one to accept.

My bank initiated a project to connect to the new multibank platform. The reasoning was that since a two-second response time was good enough for a user on the Web site, it should be good enough for the new platform, and so the same quote and trade engine could be reused. Within weeks of going live, however, the bank was winning a surprisingly small percentage of RFQs. The root cause was latency. When two banks responded with the same price (which happened quite often), the first response was displayed at the top of the list. Most clients waited to see a few different quotes and then clicked on the one at the top of the list. The result was that the fastest bank often won the client's business—and my bank wasn't the fastest.

The slowest part of the quote-generation process occurred in the database queries loading customer pricing parameters. Adding a cache to the quote engine and optimizing a few other "hot spots" in the code brought quote latency down to the range of roughly 100 milliseconds. With a faster engine, the bank was able to capture significant market share on the competitive quotation platform—but the market continued to evolve.

STREAMING QUOTES

By 2006 a new style of currency trading was becoming popular. Instead of a customer sending a specific request and the bank responding with a quote, customers wanted the banks to send a continuous *stream* of quotes. This streaming-quotes style of trading was especially popular with certain hedge funds that were developing automated trading strategies—applications that would receive streams of quotes from multiple banks and automatically decide when to trade. In many cases, humans were now out of the loop on both sides of the trade.

To understand this new competitive dynamic, it's important to know how banks compute the rates they charge their clients for foreign-exchange transactions. The largest banks trade currencies with each other in the so-called interbank market. The exchange rates set in that market are the most competitive and form the basis for the rates (plus some markup) that are offered to clients. Every time the interbank rate changes, each bank recomputes and republishes the corresponding client rate quotes. If a client accepts a quote (i.e., requests to trade against a quoted exchange rate), the bank can immediately execute an offsetting trade with the interbank market, minimizing risk and locking in a small profit. There are, however, risks to banks that are slow to update their quotes. A simple example can illustrate:

Imagine that the interbank spot market for EUR/USD has rates of 1.3558 / 1.3560. (The term *spot* means that the agreed-upon currencies are to be exchanged within two business days. Currencies can be traded for delivery at any mutually agreed-upon date in the future, but the spot market is the most active in terms of number of trades.) Two rates are quoted: one for buying (the *bid* rate), and one for selling (the *offered* or *ask* rate). In this case, a participant in the interbank market could sell one euro and receive 1.3558 US dollars in return. Conversely, one could buy one euro for a price of 1.3560 US dollars.

TABLE 2 **Sequence of events**

Time (ms)	Event
100	Banks A and B receive messages indicating that the interbank rate increased to 1.3563 / 1.3566. Both start the process of computing updated quotes.
150	Bank A publishes an updated quote of 1.3562 / 1.3567.
200	Client C receives the updated quote from bank A but still has the old quote from bank B of 1.3557 / 1.3561.
210	Client C recognizes an arbitrage opportunity and sends a request to sell 1 million euros to bank A (receiving $1.3562 million) and at the same time sends a request to buy 1 million euros from bank B (paying $1.3561 million).
260	Bank A receives the request from C in which A receives 1 million euros from C and pays $1.3562 million in return.
260	Bank B receives the request from C in which B pays 1 million euros to C and receives $1.3561 million.
270	Since the quote is valid, A sends back an acknowledgment to C and at the same time sends a request to the interbank market to sell 1 million euros.
270	B likewise finds the request to be for a valid quote (it hasn't yet sent out the update that was started at 100 ms), so acknowledges to C and sends a request to buy 1 million euros from the interbank market.
320	C receives acknowledgments from A and B. Its net profit is $100 (it received $1.3562 million from A and paid $1.3561 to B).
350	Bank B publishes its updated quote of 1.3562 / 1.3567. This does not change the processing of the order that was already received from client C at the old rate.
380	Bank A receives acknowledgment from the interbank market. Bank A makes a net profit of $100 (it paid $1.3562 million to C and received $1.3563 million from the interbank market).
380	Bank B receives acknowledgment from the interbank market. Unfortunately, the current interbank rate is 1.3563 / 1.3566, so it its order is filled at 1.3566, which means it pays $1.3566 million to buy the 1 million euros needed to offset the amount sold to C. Bank B thus suffers a net loss of $500.

Say that two banks, A and B, are participants in the interbank market and are publishing quotes to the same hedge fund client, C. Both banks add a margin of 0.0001 to the exchange rates they quote to their clients—so both publish quotes of 1.3557 / 1.3561 to client C. Bank A, however, is faster at updating its quotes than bank B, taking about 50 milliseconds while bank B takes about 250 milliseconds. There are approximately 50 milliseconds of network latency between banks A and B and their mutual client C. Both banks A and B take about 10 milliseconds to acknowledge an order, while the hedge fund C takes about 10 milliseconds to evaluate new quotes and submit orders. Table 2 breaks down the sequence of events.

The net effect of this new streaming-quote style of trading was that any bank that was significantly slower than its rivals was likely to suffer losses when market prices changed and its quotes weren't updated quickly enough. At the same time, those banks that could update their quotes fastest made significant profits. Latency was no longer just a factor in operational efficiency or market share—it directly impacted the profit and loss of the trading desk. As the volume and speed of trading increased throughout the mid-2000s, these profits and losses grew to be quite large. (How low can you go? Table 3 shows some examples of approximate latencies of systems and applications across nine orders of magnitude.)

To improve its latency, my bank split its quote and trading engine into distinct applications and rewrote the quote engine in C++. The small delays added by each hop in the network from the interbank market to the bank and onward to its clients were now significant, so the bank upgraded

TABLE 3 **Examples of approximate latencies**

Time scale (exponent seconds)	Examples
0 (1 second)	Acceptable response time to Web page loads or other query-/response-style interactions.
-1 (100 milliseconds)	Human perception of delay in interactive media (e.g., voice communications). Realtime advertising auctions. Network latency across continents or oceans.
-2 (10 milliseconds)	Hard-disk seek times and related I/O operations on spinning disks when the data is not cached.
-3 (1 millisecond)	Realtime analytics in competitive environments (e.g., natural language processing, topic classification).
-4 (100 microseconds)	Limits of metropolitan-area communications: microwaves travel about 30 km in air; light travels about 20 km in fiber.
-5 (10 microseconds)	Trade execution logic for relatively simple strategies running on CPU.
-6 (1 microsecond)	Sending a packet from one server to another over a single-hop local-area network.
-7 (100 nanoseconds)	Main memory access on modern Intel microprocessors. Thread context switch (best case; it can be much worse). Switching latency in 10-Gbps cut-through switches.
-8 (10 nanoseconds)	Accuracy of good commercial GPS-based time sources.
-9 (1 nanosecond)	Time to execute a single instruction on a modern CPU, assuming that operands are in registers or L1 cache.

firewalls and procured dedicated telecom circuits. Network upgrades combined with the faster quote engine brought end-to-end quote latency down below 10 milliseconds for clients who were physically located close to our facilities in New York, London, or Hong Kong. Trading performance and profits rose accordingly—but, of course, the market kept evolving.

ENGINEERING SYSTEMS FOR LOW LATENCY

The latency requirements of a given application can be addressed in many ways, and each problem requires a different solution. There are some common themes, though. First, it is usually necessary to measure latency before it can be improved. Second, optimization often requires looking below abstraction layers and adapting to the reality of the physical infrastructure. Finally, it is sometimes possible to restructure the algorithms (or even the problem definition itself) to achieve low latency.

LIES, DAMN LIES, AND STATISTICS

The first step to solving most optimization problems (not just those that involve software) is to measure the current system's performance. Start from the highest level and measure the end-to-end latency. Then measure the latency of each component or processing stage. If any stage is taking an unusually large portion of the latency, then break it down further and measure the latency of its substages. The goal is to find the parts of the system that contribute the most to the total latency and focus optimization efforts there. This is not always straightforward in practice, however.

For example, imagine an application that responds to customer quote requests received over a network. The client sends 100 quote requests in quick succession (the next request is sent as soon as the prior response is received) and reports total elapsed time of 360 milliseconds—or 3.6 milliseconds on average to service a request. The internals of the application are broken down and measured using the same 100-quote test set:

- Read input message from network and parse – 5 microseconds
- Look up client profile – 3.2 milliseconds (3,200 microseconds)
- Compute client quote – 15 microseconds
- Log quote – 20 microseconds
- Serialize quote to a response message – 5 microseconds
- Write to network – 5 microseconds

As clearly shown in this example, significantly reducing latency means addressing the time it takes to look up the client's profile. A quick inspection shows that the client profile is loaded from a database and cached locally. Further testing shows that when the profile is in the local cache (a simple hash table), response time is usually under a microsecond, but when the cache is missed it takes several *hundred* milliseconds to load the profile. The average of 3.2 milliseconds was almost entirely the result of one very slow response (of about 320 milliseconds) caused by a cache miss. Likewise, the client's reported 3.6-millisecond average response time turns out to be a single very slow response (350 milliseconds) and 99 fast responses that took around 100 microseconds each.

Means and outliers

Most systems exhibit some variance in latency from one event to the next. In some cases the variance (and especially the highest-latency outliers) drives the design, much more so than the average case. It is important to understand which statistical measure of latency is appropriate to the specific problem. For example, if you are building a trading system that earns small profits when the latency is below some threshold but incurs massive losses when latency exceeds that threshold, then you should be measuring the peak latency (or, alternatively, the percentage of requests that exceed the threshold) rather than the mean. On the other hand, if the value of the system is more or less inversely proportional to the latency, then measuring (and optimizing) the average latency makes more sense even if it means there are some large outliers.

What are you measuring?

Astute readers may have noticed that the latency measured inside the quote server application doesn't quite add up to the latency reported by the client application. That is most likely because they aren't actually measuring the same thing. Consider the following simplified pseudocode:

```
(In the client application)

for (int i = 0; i < 100; i++){
  RequestMessage requestMessage = new RequestMessage(quoteRequest);
  long sentTime = getSystemTime();
  sendMessage(requestMessage);
  ResponseMessage responseMessage = receiveMessage();
  long quoteLatency = getSystemTime() - sentTime;
  logStats(quoteLatency);
}
```

```
(In the quote server application)

while (true){
  RequestMessage requestMessage = receive();
  long receivedTime = getSystemTime();
  QuoteRequest quoteRequest = parseRequest(requestMessage);
  long parseTime = getSystemTime();
  long parseLatency = parseTime - receivedTime;
  ClientProfile profile = lookupClientProfile(quoteRequest.client);
  long profileTime = getSystemTime();
  long profileLatency = profileTime - parseTime;
  Quote quote = computeQuote(profile);
  long computeTime = getSystemTime();
  long computeLatency = computeTime - profileTime;
  logQuote(quote);
  long logTime = getSystemTime();
  long logLatency = logTime - computeTime;
  QuoteMessage quoteMessage = new QuoteMessage(quote);
  long serializeTime = getSystemTime();
  long serializationLatency = serializeTime - logTime;
  send(quoteMessage);
  long sentTime = getSystemTime();
  long sendLatency = sentTime - serializeTime;
  logStats(parseLatency, profileLatency, computeLatency,
           logLatency, serializationLatency, sendLatency);
}
```

Note that the elapsed time measured by the client application includes the time to transmit the request over the network, as well as the time for the response to be transmitted back. The quote server, on the other hand, measures the time elapsed only from the arrival of the quote to when it is sent (or more precisely, when the `send` method returns). The 350-microsecond discrepancy between the average response time measured by the client and the equivalent measurement by the quote server could be caused by the network, but it might also be the result of delays within the client or server. Moreover, depending on the programming language and operating system, checking the system clock and logging the latency statistics may introduce material delays.

This approach is simplistic, but when combined with code-profiling tools to find the most commonly executed code and resource contention, it is usually good enough to identify the first (and often easiest) targets for latency optimization. It's important to keep this limitation in mind, though.

Measuring distributed systems latency via network traffic capture

Distributed systems pose some additional challenges to latency measurement—as well as some opportunities. In cases where the system is distributed across multiple servers it can be hard to correlate timestamps of related events. The network itself can be a significant contributor to the latency of the system. Messaging middleware and the networking stacks of operating systems can be complex sources of latency.

At the same time, the decomposition of the overall system into separate processes running on independent servers can make it easier to measure certain interactions accurately between components of the system over the network. Many network devices (such as switches and routers) provide mechanisms for making timestamped copies of the data that traverse the device with minimal impact on the performance of the device. Most operating systems provide similar capabilities in software, albeit with a somewhat higher risk of delaying the actual traffic. Timestamped network-traffic captures (often called *packet captures*) can be a useful tool to measure more precisely when a message was exchanged between two parts of the system. These measurements can be obtained without modifying the application itself and generally with very little impact on the performance of the system as a whole. (See http://wireshark.org and http://tcpdump.org.)

Clock synchronization

One of the challenges of measuring performance at short time scales across distributed systems is clock synchronization. In general, to measure the time elapsed from when an application on server A transmits a message to when the message reaches a second application on server B, it is necessary to check the time on A's clock when the message is sent and on B's clock when the message arrives, and then subtract those two timestamps to determine the latency. If the clocks on A and B are not in sync, then the computed latency will actually be the real latency plus the clock skew between A and B.

When is this a problem in the real world? Real-world drift rates for the quartz oscillators that are used in most commodity server motherboards are on the order of 10^-5, which means that the oscillator may be expected to drift by 10 microseconds each second. If uncorrected, it may gain or lose as much as a second over the course of a day. For systems operating at time scales of milliseconds or less, clock skew may render the measured latency meaningless. Oscillators with significantly lower drift rates are available, but without some form of synchronization, they will eventually drift apart. Some mechanism is needed to bring each server's local clock into alignment with some common reference time.

Developers of distributed systems should understand NTP (Network Time Protocol) at a minimum and are encouraged to learn about PTP (Precision Time Protocol) and usage of external signals such as GPS to obtain high-accuracy time synchronization in practice. Those who need time accuracy at the sub-microsecond scale will want to become familiar with hardware implementations of PTP (especially at the network interface) as well as tools for extracting time information from each core's local clock. (See https://tools.ietf.org/html/rfc1305, https://tools.ietf.org/html/rfc5905, http://www.nist.gov/el/isd/ieee/ieee1588.cfm, and http://queue.acm.org/detail.cfm?id=2354406.)

ABSTRACTION VERSUS REALITY

Modern software engineering is built upon abstractions that allow programmers to manage the complexity of ever-larger systems. Abstractions do this by simplifying or generalizing some aspect of the underlying system. This doesn't come for free, though—simplification is an inherently lossy process and some of the lost details may be important. Moreover, abstractions are often defined in terms of function rather than performance.

Somewhere deep below an application are electrical currents flowing through semiconductors and pulses of light traveling down fibers. Programmers rarely need to think of their systems in these terms, but if their conceptualized view drifts too far from reality they are likely to experience unpleasant surprises.

Four examples illustrate this point:

• TCP provides a useful abstraction over UDP (User Datagram Protocol) in terms of delivery of a sequence of bytes. TCP ensures that bytes will be delivered in the order they were sent even if some of the underlying UDP datagrams are lost. The transmission latency of each byte (the time from when it is written to a TCP socket in the sending application until it is read from the corresponding receiving application's socket) is not guaranteed, however. In certain cases (specifically when an intervening datagram is lost) the data contained in a given UDP datagram may be delayed significantly from delivery to the application, while the missed data ahead of it is recovered.

• Cloud hosting provides virtual servers that can be created on demand without precise control over the location of the hardware. An application or administrator can create a new virtual server "on the cloud" in less than a minute—an impossible feat when assembling and installing physical hardware in a data center. Unlike the physical server, however, the location of the cloud server or its location in the network topology may not be precisely known. If a distributed application depends on the rapid exchange of messages between servers, the physical proximity of those servers may have a significant impact on the overall application performance.

• Threads allow developers to decompose a problem into separate sequences of instructions that can be allowed to run concurrently, subject to certain ordering constraints, and that can operate on shared resources (such as memory). This allows developers to take advantage of multicore processors without needing to deal directly with issues of scheduling and core assignment. In some cases, however, the overhead of context switches and passing data between cores can outweigh the advantages gained by concurrency.

• Hierarchical storage and cache-coherency protocols allow programmers to write applications that use large amounts of virtual memory (on the order of terabytes in modern commodity servers), while experiencing latencies measured in nanoseconds when requests can be serviced by the closest caches. The abstraction hides the fact that the fastest memory is very limited in capacity (e.g., register files on the order of a few kilobytes), while memory that has been swapped out to disk may incur latencies in the tens of milliseconds.

Each of these abstractions is extremely useful but can have unanticipated consequences for low-latency applications. There are some practical steps to take to identify and mitigate latency issues resulting from these abstractions.

Messaging and Network Protocols

The near ubiquity of IP-based networks means that regardless of which messaging product is in use, under the covers the data is being transmitted over the network as a series of discrete packets. The performance characteristics of the network and the needs of an application can vary dramatically—so one size almost certainly does not fit all when it comes to messaging middleware for latency-sensitive distributed systems.

There's no substitute for getting under the hood here. For example, if an application runs on a private network (you control the hardware), communications follow a publisher/subscriber model, and the application can tolerate a certain rate of data loss, then raw multicast may offer significant performance gains over any middleware based on TCP. If an application is distributed across very long distances and data order is not important, then a UDP-based protocol may offer advantages in terms of not stalling to resend a missed packet. If TCP-based messaging is being used, then it's worth keeping in mind that many of its parameters (especially buffer sizes, slow start, and Nagle's algorithm) are configurable and the "out-of-the-box" settings are usually optimized for throughput rather than latency (http://queue.acm.org/detail.cfm?id=2539132).

Location

The physical constraint that information cannot propagate faster than the speed of light is a very real consideration when dealing with short time scales and/or long distances. The two largest stock exchanges, NASDAQ and NYSE, run their matching engines in data centers in Carteret and Mahwah, New Jersey, respectively. A ray of light takes 185 microseconds to travel the 55.4-km distance between these two locations. Light in a glass fiber with a refractive index of 1.6 and following a slightly longer path (roughly 65 km) takes almost 350 microseconds to make the same one-way trip. Given that the computations involved in trading decisions can now be made on time scales of 10 microseconds or less, signal propagation latency cannot be ignored.

Threading

Decomposing a problem into a number of threads that can be executed concurrently can greatly increase performance, especially in multicore systems, but in some cases it may actually be slower than a single-threaded solution.

Specifically, multithreaded code incurs overhead in the following three ways:

• When multiple threads operate on the same data, controls are required to ensure that the data remains consistent. This may include acquisition of locks or implementations of read or write barriers. In multicore systems, these concurrency controls require that thread execution is suspended while messages are passed between cores. If a lock is already held by one thread, then other threads seeking that lock will need to wait until the first one is finished. If several threads are frequently accessing the same data, there may be significant contention for locks.

• Similarly, when multiple threads operate on the same data, the data itself must be passed between cores. If several threads access the same data but each performs only a few computations on it, the time required to move the data between cores may exceed the time spent operating on it.

• Finally, if there are more threads than cores, the operating system must periodically perform a context switch in which the thread running on a given core is halted, its state is saved, and another thread is allowed to run. The cost of a context switch can be significant. If the number of threads far exceeds the number of cores, context switching can be a significant source of delay.

In general, application design should use threads in a way that represents the inherent concurrency of the underlying problem. If the problem contains significant computation that can be performed in isolation, then a larger number of threads is called for. On the other hand, if there is a high degree of interdependency between computations or (worst case) if the problem is inherently serial, then a single-threaded solution may make more sense. In both cases, profiling tools should be used to identify excessive lock contention or context switching. Lock-free data structures (now available for several programming languages) are another alternative to consider (http://queue.acm.org/detail.cfm?id=2492433).

It's also worth noting that the physical arrangement of cores, memory, and I/O may not be uniform. For example, on modern Intel microprocessors certain cores can interact with external I/O (e.g., network interfaces) with much lower latency than others, and exchanging data between certain cores is faster than others. As a result, it may be advantageous explicitly to pin specific threads to specific cores (http://queue.acm.org/detail.cfm?id=2513149).

Hierarchical storage and cache misses

All modern computing systems use hierarchical data storage—a small amount of fast memory combined with multiple levels of larger (but slower) memory. Recently accessed data is cached so that subsequent access is faster. Since most applications exhibit a tendency to access the same memory multiple times in a short period, this can greatly increase performance. To obtain maximum benefit, however, the following three factors should be incorporated into application design:

• Using less memory overall (or at least in the parts of the application that are latency-sensitive) increases the probability that needed data will be available in one of the caches. In particular, for especially latency-sensitive applications, designing the app so that frequently accessed data fits within the CPU's caches can significantly improve performance. Specifications vary but Intel's Haswell microprocessors, for example, provide 32 KB per core for L1 data cache and up to 40 MB of shared L3 cache for the entire CPU.

• Repeated allocation and release of memory should be avoided if reuse is possible. An object or data structure that is allocated once and reused has a much greater chance of being present in a cache than one that is repeatedly allocated anew. This is especially true when developing in environments where memory is managed automatically, as overhead caused by garbage collection of memory that is released can be significant.

• The layout of data structures in memory can have a significant impact on performance because of the architecture of caches in modern processors. While the details vary by platform and are outside the scope of this article, it is generally a good idea to prefer arrays as data structures over linked lists and trees and to prefer algorithms that access memory sequentially since these allow the hardware prefetcher (which attempts to load data preemptively from main memory into cache *before* it is requested by the application) to operate most efficiently. Note also that data that will be operated on concurrently by different cores should be structured so that it is unlikely to fall in the same cache line (the latest Intel CPUs use 64-byte cache lines) to avoid cache-coherency contention.

A note on premature optimization

The optimizations just presented should be considered part of a broader design process that takes into account other important objectives including functional correctness, maintainability, etc. Keep in mind Knuth's quote about premature optimization being the root of all evil; even in the most performance-sensitive environments, it is rare that a programmer should be concerned with determining the correct number of threads or the optimal data structure until empirical measurements indicate that a specific part of the application is a hot spot. The focus instead should be on ensuring that performance requirements are understood early in the design process and that the system architecture is sufficiently decomposable to allow detailed measurement of latency when and as optimization becomes necessary. Moreover (and as discussed in the next section), the most useful optimizations may not be in the application code at all.

CHANGES IN DESIGN

The optimizations presented so far have been limited to improving the performance of a system for a given set of functional requirements. There may also be opportunities to change the broader design of the system or even to change the functional requirements of the system in a way that still meets the overall objectives but significantly improves performance. Latency optimization is no exception. In particular, there are often opportunities to trade reduced efficiency for improved latency.

Three real-world examples of design tradeoffs between efficiency and latency are presented here, followed by an example where the requirements themselves present the best opportunity for redesign.

Speculative precomputation

In certain cases trading efficiency for latency may be possible, especially in systems that operate well below their peak capacity. In particular, it may be advantageous to compute possible outputs in advance, especially when the system is idle most of the time but must react quickly when an input arrives.

A real-world example can be found in the systems used by some firms to trade stocks based on news such as earnings announcements. Imagine that the market expects Apple to earn between $9.45 and $12.51 per share. The goal of the trading system, upon receiving Apple's actual earnings, would be to sell some number of shares Apple stock if the earnings were below $9.45, buy some number of shares if the earnings were above $12.51, and do nothing if the earnings fall within the expected range. The act of buying or selling stocks begins with submitting an order to the exchange. The order consists of (among other things) an indicator of whether the client wishes to buy or sell, the identifier of the stock to buy or sell, the number of shares desired, and the price at which the client wishes to buy or sell. Throughout the afternoon leading up to Apple's announcement, the client would receive a steady stream of market-data messages that indicate the current price at which Apple's stock is trading.

A conventional implementation of this trading system would cache the market-price data and, upon receipt of the earnings data, decide whether to buy or sell (or neither), construct an order, and serialize that order to an array of bytes to be placed into the payload of a message and sent to the exchange.

An alternative implementation performs most of the same steps but does so on every market-data update rather than only upon receipt of the earnings data. Specifically, when each market-data update message is received, the application constructs two new orders (one to buy, one to sell) at the current prices and serializes each order into a message. The messages are cached but not sent. When the next market-data update arrives, the old order messages are discarded and new ones are created. When the earnings data arrives, the application simply decides which (if either) of the order messages to send.

The first implementation is clearly more efficient (it has a lot less wasted computation), but at the moment when latency matters most (i.e., when the earnings data has been received), the second algorithm is able to send out the appropriate order message sooner. Note that this example presents application-level precomputation; there is an analogous process of branch prediction that takes place in pipelined processors which can also be optimized (via guided profiling) but is outside the scope of this article.

Keeping the system warm

In some low-latency systems long delays may occur between inputs. During these idle periods, the system may grow “cold.” Critical instructions and data may be evicted from caches (costing hundreds of nanoseconds to reload), threads that would process the latency-sensitive input are context-switched out (costing tens of microseconds to resume), CPUs may switch into power-saving states (costing a few milliseconds to exit), etc. Each of these steps makes sense from an efficiency standpoint (why run a CPU at full power when nothing is happening?), but all of them impose latency penalties when the input data arrives.

In cases where the system may go for hours or days between input events there is a potential operational issue as well: configuration or environmental changes may have “broken” the system in some important way that won’t be discovered until the event occurs—when it’s too late to fix.

A common solution to both problems is to generate a continuous stream of dummy input data to keep the system “warm.” The dummy data needs to be as realistic as possible to ensure that it keeps the right data in the caches and that breaking changes to the environment are detected. The dummy data needs to be reliably distinguishable from legitimate data, though, to prevent downstream systems or clients from being confused.

Redundant processing

It is common in many systems to process the same data through multiple independent instances of the system in parallel, primarily for the improved resiliency that is conferred. If some component fails, the user will still receive the result needed. Low-latency systems gain the same resiliency benefits of parallel, redundant processing but can also use this approach to reduce certain kinds of variable latency.

All real-world computational processes of nontrivial complexity have some variance in latency even when the input data is the same. These variations can be caused by minute differences in thread scheduling, explicitly randomized behaviors such as Ethernet’s exponential back-off algorithm, or other unpredictable factors. Some of these variations can be quite large: page faults, garbage collections, network congestion, etc., can all cause occasional delays that are several orders of magnitude larger than the typical processing latency for the same input.

Running multiple, independent instances of the system, combined with a protocol that allows the end recipient to accept the first result produced and discard subsequent redundant copies, both provides the benefit of less-frequent outages and avoids some of the larger delays.

Stream processing and short circuits

Consider a news analytics system whose requirements are understood to be “build an application that can extract corporate earnings data from a press release document as quickly as possible.” Separately, it was specified that the press releases would be pushed to the system via FTP. The system was thus designed as two applications: one that received the document via FTP, and a second that parsed the document and extracted the earnings data. In the first version of this system, an open-source FTP server was used as the first application, and the second application (the parser) assumed that it would receive a fully formed document as input, so it did not start parsing the document until it had fully arrived.

Measuring the performance of the system showed that while parsing was typically completed in just a few milliseconds, receiving the document via FTP could take tens of milliseconds from the arrival of the first packet to the arrival of the last packet. Moreover, the earnings data was often present in the first paragraph of the document.

In a multistep process it may be possible for subsequent stages to start processing before prior stages have finished, sometimes referred to as *stream-oriented* or *pipelined processing*. This can be especially useful if the output can be computed from a partial input. Taking this into account, the developers reconceived their overall objective as "build a system that can deliver earnings data to the client as quickly as possible." This broader objective, combined with the understanding that the press release would arrive via FTP and that it was possible to extract the earnings data from the first part of the document (i.e., before the rest of the document had arrived), led to a redesign of the system.

The FTP server was rewritten to forward portions of the document to the parser as they arrived rather than wait for the entire document. Likewise, the parser was rewritten to operate on a stream of incoming data rather than on a single document. The result was that in many cases the earnings data could be extracted within just a few milliseconds of the *start* of the arrival of the document. This reduced overall latency (as observed by the client) by several tens of milliseconds without the internal implementation of the parsing algorithm being any faster.

CONCLUSION

While latency requirements are common to a wide array of software applications, the financial trading industry and the segment of the news media that supplies it with data have an especially competitive ecosystem that produces challenging demands for low-latency distributed systems.

As with most engineering problems, building effective low-latency distributed systems starts with having a clear understanding of the problem. The next step is measuring actual performance and then, where necessary, making improvements. In this domain, improvements often require some combination of digging below the surface of common software abstractions and trading some degree of efficiency for improved latency.

Related content at queue.acm.org

There's Still Some Life Left in Ada
Alexander Wolfe
http://queue.acm.org/detail.cfm?id=1035608

Principles of Robust Timing over the Internet
Julien Ridoux and Darryl Veitch
http://queue.acm.org/detail.cfm?id=1773943

Online Algorithms in High-frequency Trading
Jacob Loveless, Sasha Stoikov, and Rolf Waeber
http://queue.acm.org/detail.cfm?id=2534976

LOVE IT, HATE IT? LET US KNOW
feedback@queue.acm.org

ANDREW BROOK is the CTO of Selerity, a provider of realtime news, data, and content analytics. Previously he led development of electronic currency trading systems at two large investment banks and launched a pre-dot-com startup to deliver AI-powered scheduling software to agile manufacturers. His expertise lies in applying distributed, realtime systems technology and data science to real-world business problems. He finds Wireshark to be more interesting than PowerPoint.