Low Latency Web Programming

Sheldon Linker
sheldon.linker@faculty.umgc.edu

Abstract
Many modern coding practices are at odds with low-latency web-site response times and other response times. This paper discusses reasons and mitigations.

1. Introduction

Recently, there has been an increase in the number of consulting requests on Dice.com pertaining to the production of or reengineering for low-latency application programs, especially web sites. As there have been a number of popular coding practices over the last few years which lead to added and unneeded latency, this paper is intended to point out those practices, and how latency might be reduced.

The items that follow have been arranged in order from practices which cause the most latency to practices which cause the least latency. The reason for this arrangement is so that someone using this paper as a cookbook to reduce latency can make use of the "low-hanging fruit" first, and proceed only if more speed is needed.

2. Traffic

This first category of problems tends to cause the largest latency, and involves unneeded network transmissions and other work. Traffic issues often involve a server waiting idly for another server to finish work, or worse, both servers waiting idly for data to traverse the network.

2.1 Solution-Oriented Architecture (SOA)

In general, solution-oriented architectures are architectures in which each computer (or group of computers) in a network have a specific function, and the work progresses in stages from one computer to another, eventually getting finished. As with any decision point in a design, there are reasons to go with and against either side of each design issue. Unfortunately, also as with most decision points in design issues, there are those who believe that a certain decision is always warranted in all cases, and will thus make the wrong decision in half the cases.

Use of a SOA is one such decision point. As mentioned, there are good reasons to use an SOA, but there are also drawbacks. In general, there are two good reasons to use a SOA (1.1.1 and 1.1.2) and one bad reason (1.1.3).

2.1.1 Use SOA when systems cannot share resources

The situation in which two or more systems can't share resources is best shown by example. A certain eCommerce site needs to accept credit cards and have shipping done for it by a warehousing/shipping company. In the success case of a purchase, the customer puts items into a virtual shopping basket, and then initiates the purchase. The ecommerce site then contacts a banking site to place a lien on the credit card. The eCommerce site then contacts the shipping company, and tells it to ship the goods. The shipping company reports that the entire order will ship that day. The eCommerce site then tells the banking site to convert the lien to a sale, and finally reports a successful transaction to the customer. This is a necessary and thus proper use of SOA, because the eCommerce site cannot be combined with the banking site, nor can it be combined with the shipping service's site.

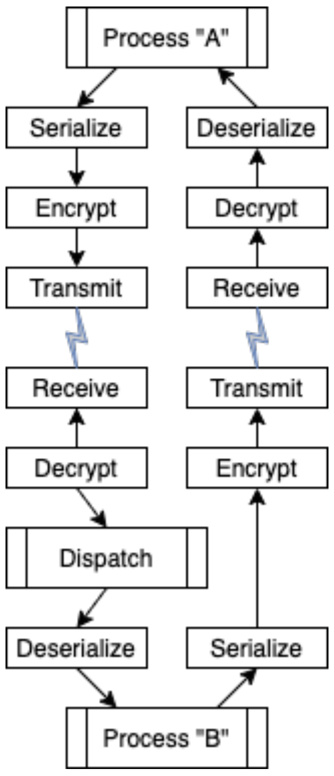
2.1.2 Use SOA when certain functions must occur at different tiers

The situation in which work must occur at different tiers most commonly occurs when two or more business tier servers must interact with a database, and especially when some sort of flow control, such as locking, must occur. It's fairly typical in most applications to have some web servers connect to a database server to do the database work for them. Two business tier systems can't easily coordinate locking between themselves, or at least can't do it significantly faster than by using a database or locking server. So, if data changed by one business tier server is to be seen by the other business tier servers, then a database tier server should be used. But, if data can be local to each business tier server (such as in the read-only case), then such data can be accessed faster by having local copies.

2.1.3 Don't use SOA if it's not required

Most of the other uses of SOA fall into the category of "SOA is the modern way to do things" or "SOA is the cool way to do things" or something of that nature. Unfortunately, it's also the slow way to do things. In a typical "call" type transaction, program A wants sub-program B to do something for it. A passes B some work package, which B handles, and passes a result back to A. In a monolithic application (or at least monolithic in the case of this specific call), A calls B as a function, passing in whatever data is required (or a reference to whatever data is required), B does the work, and passes back the result (or a reference to it). A takes some amount of time, and so does B. But, in the SOA model, A and B are in different processes, and likely even on different computers. Now, the processing looks more like this: A calls some proxy representing B, passing in the data. That proxy converts the request and data to a serialized form, perhaps binary, but more likely JSON or XML (listed in most-to-least efficient order), then encrypts the request and data, then transmits it. Once the request is received and then decrypted, it is queued and dispatched, and the other side of the proxy deserializes the data, and B finally runs, producing a result. That result again goes through the serialize, encrypt, transmit, receive, decrypt, deserialize process.

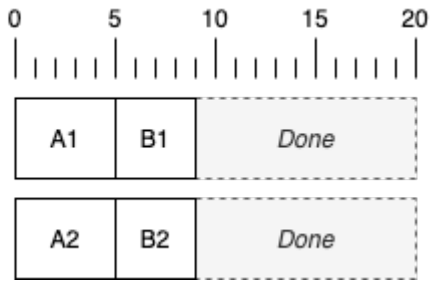
Graphically, that would look like this:



The above represents a lot of extra work. But, it's not the whole of the latency for SOA. Another source is lopsided work allocation. Let's say that what A does takes 5 ticks (some arbitrary unit of work time) to accomplish, that the whole A-to-B proxy work takes 2 ticks, that B's work takes 4 ticks, and that the proxy work going the other way takes 1 tick. At first, it would seem that using the pure call scheme takes 5+4 = 9 ticks, and that the SOA scheme takes 5+2+4+1 = 12 ticks.

But, if we have two such processes running on two computers, the numbers turn out to be even worse. First, let's look at the process in the Call scheme on two computers. At time 0, two A processes start. At time 5 ticks, two B processes start. At time 9 ticks, the entire process is complete.

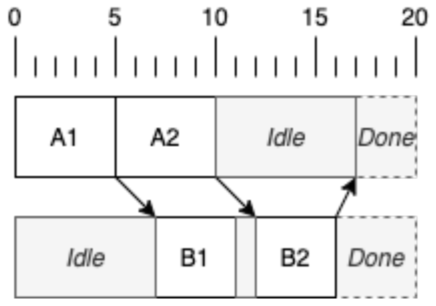
Here is the symmetric two-processor timing diagram:



Now, let's look at the same thing happening in the SOA architecture: At time 0, the first A process starts on the computer that runs the A processes. At time 5 ticks, we begin sending the first request to the first B process, and start the second A process. At time 7 ticks, the first B process starts. At time 10 ticks, the second A process begins sending to the second B process. At time 11 ticks, the first B process completes, and begins sending the result to the first A process. At time 12 ticks, the first A process receives its result, and the second B process kicks off. At time 16 ticks, the second B is finished, and begins sending the result to the second A. At time 17 ticks, the entire process is complete.

In general, not counting any transmission overhead, any such A/B SOA is going to have idle time on whichever processor is running the fastest component, meaning that some potential processor throughput is going to be wasted.

Here is the asymmetric two-processor timing diagram:



2.2 Do validation at every tier

At first, it might seem counterintuitive that doing validation at every tier might reduce latency, but it can. Here's how: Let's say that we have a typical 3-tier web application, with a web page containing HTML, CSS, and JavaScript running at the client tier, the language of your choice running at the web server tier, and SQL running at the database tier…

2.2.1 At the client tier

Run validation at the client tier, either on an attempted submission, or as data is entered. When running on attempted submission, a user clicking on a Submit button should run JavaScript code that validates whether in-field and cross-field criteria have been met for a valid request. An in-field criterion might be that a required value is present, that a field be properly formatted, and/or that a value is within some reasonable range and/or format. A cross-field criterion might be that two fields have reasonable values with respect to each other, such as a start date being before an end date. If all submission requirements have been met, then the submission occurs. If not, an explanation is made to the user. But, a more modern and less frustrating (to the user) interface is the live check method. In this method, each time a data entry field is modified, the field (or in the case of cross-checked field criteria, fields) is (or are) checked, and if the form is not in a submittable form, the Submit button is disabled, and some visible feedback is shown, typically as a message, and by field shading (such as 15% red to indicate an erroneous field value).

How this relates to web traffic is that no error-containing form submissions ever get transmitted. This can easily reduce form submissions by 10%. That decreases the number of hits on the web servers.

2.2.2 At the web tier

Since web requests can still arrive at the web server with improper form values (typically because JavaScript has been disabled, because of malicious intent, or possibly most dangerous, because a programmer at a client site has decided to automate things), a complete set of checks must also be made at the web server tier. However, this type of checking need not bother with a lot of explanation and formatting. Given that only unfiltered or hand-made erroneous requests arrive in this manner, a simple "Bad request" notice will do.

Similarly, any request made should have its values checked before attempting to pass the request to the database tier, so that all requests going to the database are valid.

2.2.3 At the database tier

There are two reasons that data should be checked at the database tier. One is that the web tier may contain a programming error (or missing or unimplemented functionality) that allows bad values or interrelationships to be sent to the database tier. Another reason is that programmers may issue SQL directly into the database, and the database should be able to protect itself, using a combination of constraints such as `NOT NULL`, `UNIQUE`, `PRIMARY KEY`, `FOREIGN KEY`, `CHECK`, and `TRIGGER BEFORE`.

2.3 Group up database requests

Group up database requests using stored procedures and/or multiple-action SQL (or other database type) calls. Consider the case of adding two records to the database. These two examples, both shown as coded in "`EXEC`" form are all equivalent:

Example 1:
```
EXEC SQL INSERT INTO myTable
   VALUES(:key1, :value1);
EXEC SQL INSERT INTO myTable
   VALUES(:key2, :value2);
```

Example 2:
```
EXEC SQL BEGIN
   INSERT INTO myTable VALUES(:key1, :value1);
   INSERT INTO myTable VALUES(:key2, :value2);
END;
```

In the first example, two separate transmit, process, respond transactions occur in the three-tier case (or four context switches in the two-tier case). In the second example, there is only one transmit, process, respond transaction in the three-tier case (or two context switches in the two-tier case). A good rule of thumb is that a transmit, process, respond transaction will take about 100 times as long as a simple SQL statement.

Just for the sake of completeness, the following examples also work in DB/2, and will give slightly better performance than will Example 2:

Example 3 (DB/2 only):
```
EXEC SQL BEGIN INSERT INTO myTable
   VALUES(:key1, :value1), (:key2, :value2);
```

Example 4 (DB/2 only):
```
EXEC SQL BEGIN INSERT INTO myTable
   VALUES(:record1), (:record2);
```

Example 5 (DB/2 only):
```
EXEC SQL BEGIN INSERT INTO myTable :count ROWS
   VALUES(:records);
```

2.4 Run two tiers in one computer?

Many web sites run in three tiers, the client tier on the users' browser, the web service tier, and the database tier. For web-sites that are small enough to use run all of their web servicing in a single computer (either because the load is light or because they're running a "big iron" computer such as an IBM i/Series or z/Series computer), and if the load is light enough to run the entire set of database work on a single computer, then the load may be light enough to combine the web service tier and the database tier into a single computer.

You can tell whether there's enough available capacity by examining the CPU time usage on the two existing computers. If the sum of the CPU loads from the two computers is than 100%, then the two tiers can be safely combined into a single computer, and the combined total CPU load will normally be less than the sum of the two-computer set of CPU loads.

The advantage comes from the fact that in the two-computer system, the steps involved in requesting a database action are to serialize the request, encrypt it, transmit it, decrypt it, deserialize it, act on it, and then follow that same set of actions for the response. In the single-computer case, the steps involved are to serialize the request, pass it, deserialize it, act on it, and then to do the same with the response. In the case of built-in databases (including DB/2 on "big iron"), the steps are context switch, act, and context switch back.

3. Delays

A smaller cause of system latency is any delay incurred within a single computer. This section lists a number of common ways such delays are caused, and ways to abate each of the problems.

3.1 Real computers are faster than VMs

Real computers are faster than virtual machines (VMs). That's necessarily true, since a VM is an abstraction layer between the real computer and the program running. Typically, a VM set-up involves a real computer running an operating system (often a minimal operating system), which then runs one or more VMs. Each VM has its own copy of memory, and its own operating system. Any services that an application requires from the operating system have to go through various levels of context switching and thus run much slower than a real computer. Additionally, if there are two or more VMs running in the same computer, then each VM slows down the performance of the other VMs running in the same computer. In the cloud, all computers are VMs, and the likelihood is that any VM started will be sharing real hardware with another VM.

If two or more computers are to be tied together, such as usually occurs in a two-or-more-tier service environment, then physical proximity and site security becomes important. Physical proximity becomes important because of message travel time. A computer that's guaranteed to be within the same facility as another will be able to communicate with that computer at the highest transmission speeds, and will not have to use routers or relays. Additionally, if the site is secure enough, then the two computers will not have to use any sort of encryption between the them.

There is a special case in the elimination of VMs. Often, an enterprise will be running several databases. A setup which is used in a lot of companies is to have "the database computer" run two VMs. Each VM will be running a database server. Each database server will be servicing a database. That involves three operating systems (one for the real computer and one for each VM), two database servers, and two databases. But, most database servers can handle multiple databases. If the configuration is rearranged so that the database service runs in the real machine, then there is only one operating system running, one database server running, and still two databases running. Latency is cut down because there is then no VM overhead, because the cache size can be increased (because there are two less operating systems running and one fewer database service running), and the cache itself can be put to better use. The reason that the cache can be put to better use is that it's possible that in the VM configuration, one computer might be using half its cache, while the other has filled its cache and doesn't have more cache space available; while in the combined case, one database might be using 25% of the larger, combined cache, and the other database can then use the remaining 75%, rather than having to stop at the 50% mark.

3.2 Retain cursors

Many web applications will show large groups of results one page at a time. The concept is fine, but often not implemented optimally. The problem is best explained using an example. Let's say that a given page of a given application shows up to 100 items per web page, and that a given query returns more than 100 results. One way to code this is to have a query run, and cut off the results at the 100 items mark. When the user asks to see the next page of results, the query is run again, the first 100 items are skipped, and the remaining items, or next 100 items, are then returned. Depending on the running time of the query, a considerable lag could be introduced. This time lag not only affects the user doing the Next Page operation, but since system resources are used, the action may well affect all other users, too. The solution to this is to retain cursors in an open state until the last page is shown.

Note, though, that retaining cursors in an open state requires some additional provisos: (1) Holding the cursor in this manner requires that the cursor stay in a given session, and so the session must be held too, increasing the required size of the session pool; (2) If there is load sharing between processors, there must be a user session to web server affinity (more on this later); and (3) there must be some sort of time-out policy and service so that cursors can eventually be released if the user doesn't do a Next Page operation.

3.3 Collect the garbage when you can

Managed memory systems such as Java and C# and many others will eventually fill the available memory with unused objects, and these objects will have to be removed through a process called "garbage collection". The more memory the system has available, the less often this garbage collection will occur, but the longer it will take. If the system being built or being sped up can detect that it has reached an idle state, then it can force garbage collection at that time. For instance, a web site takes requests and completes requests. It can count how many requests are active. If, at completion of a request, the system finds that no more requests are pending, it could preemptively perform a garbage collection operation, so that garbage collection doesn't happen during the processing of a request. In Java, forcing garbage collection is done with "System.gc();". In C#, the call is "GC.Collect();".

3.4 Allocate memory sparingly

Allocating memory can be a time-consuming operation. Any item that can get its own memory space during compilation (static storage) should be set up that way. Languages that don't have that capability should instead do whatever allocation that can at class-load time. Even if an item must be dynamically allocated at some point, when its use is over, consider whether that space can somehow be reused or saved. A good example of this is connection pooling. Setting up a connection takes to much system time that systems provide such pooling as a matter of course.

3.5 Don't wait for I/O

In Unix/Linux and Windows, when you ask for input, you wait for input. Then you get the input, and the operation is complete. Thus, many input-and-process operations either first read an entire file, then process an entire file, or read a chunk of then file, the process it, then read more of the file, and then process that. But, operating systems like z/OS have something called "double buffering". When an application uses double buffering (the default in most cases), the system just keeps reading the file. So, an application reading and processing a file can read (and wait for) the first chunk of the file, but while the application is processing the first chunk of the file, the second chunk is already being read. With a bit of extra effort, programmers on other operating systems can implement double buffering for better speed.

The situation can be similar for output. But, for output, there are more and simpler options. If the output comes at the end of a process, the thread can afford to wait for a single large output operation. If the pattern is process, output, process output, then using or implementing double-buffering would be in order. But, if the writing is not at the end of processing, and double-buffering is not built in, and the output can be sent as one large chunk, then another way of handling things is to start up a new thread to handle the output. Note that in some languages, like PL/1, the syntax handles this for you, such as adding an "`EVENT`" clause.

4. Speed

General speed is the cause of the least latency in web applications, but at the same time may be the cause of a little bit of latency in almost all applications. The general rule is that a program that runs faster necessarily responds faster.

4.1 Separate CSS and JavaScript into separate files

Unless a line of CSS is dynamic and page-specific, then that CSS should be in a separate CSS file. The reason for this is that your system will only have to deliver the CSS file once, rather than as a part of each page. The same is true of JavaScript.

4.2 Use efficient indexing

Your program will spend a lot of time waiting for database service. Efficient SQL operation is critical. Add any needed indices. Remove any unneeded indices. If the items in an index are a subset of another index, then the smaller index is probably unneeded. Rearrange multi-column indices to match the application's use of them. Note that portions of the schema that do a lot more reading than writing place a heavier reliance on indices, and portions of the schema that are write-heavy put less reliance on indices. When you make a change to indices, check the system load before or after the change. If the load is lower, then the change should stay in place. If the load is higher, then the index should be restored to its previous state. Since indices hold only redundant data, they can be edited, and even deleted, for purposes of experimentation.

There are tuning techniques for storage and retrieval in no-SQL databases too.

4.3 Use of `CURRENT OF` and `ROWID`

In SQL (in most DBMS servers), an SQL command that involves a `WHERE` clause using `CURRENT OF` or `ROWID` takes almost no time when executed as a part of a stored procedure.

4.4 Move processing to the client tier

This was touched upon before as a special case. Any processing that can be moved to the client tier should be moved to the client tier. For instance, if a table containing amounts is to be displayed at the client, formatted with commas and a total, consider whether the formatting should happen at the database tier, at the web service tier, or at the client tier. If you can move the processing to the client tier, your servers will run faster overall. But, this will take more JavaScript code.

Note that any processing done at the client tier on data coming from that tier to the web service tier can't be trusted.

4.5 Caching and machine affinity

It's possible to store little or no session data, but that means that your web application will have to work things out from scratch a lot. To get around that problem, the web server will cache certain items. In general, the more information it can keep about the session, the faster the system will run. If there is only a single web service computer, caching is simple. The cache can be stored on the server, and accessed by session ID. But, if there is more than one computer at the web service tier, caching becomes more difficult. There are several schemes widely in use. One scheme is to serialize and encrypt the session information and to send it to the client as a cookie, and then to retrieve the cookie in each request, and decrypt and deserialize it. That method takes up a lot of bandwidth, increasing latency. Another method is to have the web service computers share their cached data between them. That also costs some amount of overhead. A very simple approach is to use machine affinity. In that scheme, the user first navigates to www.*serverName*, *serverName*, www.*serverName*/*urlTrailer*, or *serverName*/*urlTrailer*. That computer then redirects the request to the least-used server, using a numbered variant of the "www" name, for instance, ww7.*serverName*/*urlTrailer*. The rest of the conversation between the client and the assigned server will use that name, and a single-computer cache can be used for each session.

4.6 Use the most efficient coding you can

Longer messages mean more latency. So, use the shortest transmission mechanism possible. XML transmissions are shorter than SOAP. JSON transmissions are shorter than XML. In most cases, a custom transmission scheme shorter than JSON can be developed. If you can manage it, binary messages are often the shortest.

4.7 Language speed

Fully compiled and linked code is faster than compiled code using DLLs. Fully compiled code is faster than byte-coded languages like Java and C#. Byte-coded languages are faster than interpreted languages. But, note that any web service scheme that involves starting up heavy-weight processes rather than running in the same process incurs a lot of extra time. That's one of the reasons Java got popular as a web service language over C. When architecting an enterprise web-site, consider what the best language will be, based on execution speed as well as coding cost and existing skill set.

4.8 Logging

The less logging an application does, the faster it will run.

4.9 File organization

ISAM I/O is faster than using SQL or no-SQL. Direct access is faster than ISAM. Sequential files are faster than direct access. Don't fall into the trap of thinking that all storage needs to be SQL or no-SQL.

4.10 File format

Binary I/O is faster than human-readable I/O.

4.11 Abstraction layers

Every abstraction layer added to a system makes that system a little bit slower. Common abstraction layers in this category include Angular, ASP[X], Hibernate, JSP and its tag libraries, React, Spring, Struts, and Vue.

One special case of abstraction actually fits into this categorization and into others: many programs communicate with each other using data carried over an HTTP layer, often using character-encoded data (such as SOAP, XML, JSON, or the like), sent over HTTP over TCP/IP. When the message is received, it's dispatched to the web server, which then dispatches it to the proper C#, Java, JavaScript, PHP, or Python code. This whole stack can often be sped up at each step along the process. If one program needs to communicate with another, the data can often be sent in binary form, which takes little or no encoding or decoding time (depending on the languages used at each end), and transmits very quickly. The point-to-point communication can be done in TCP/IP, eliminating the HTTP encoding and decoding time, as well as the dispatching time, since the serving application will be receiving the request directly. This also allows native code to handle the request, saving on all the overhead of maintaining a VM and/or interpreter. If the service program uses C*Plus or C/SQL combined with Oracle or DB/2 running on the same machine, then database calls can be made without encoding the data into an SQL request at all.

4.12 Preparation of SQL and stored procedures

Any SQL statement that's going to run more than once will run faster if prepared first. Stored procedures will run faster than sending and executing the code separately. An exception to each of these rules is that if something is going to be executed only once, then *ad hoc* SQL is the fastest.

4.13 Don't look up or transmit what's already known

That's a simple enough rule, but full implementation of it can occur in a number of places. For instance, most inversion-of-control systems involve looking up how the application fits together, even though that's known at coding time. If an application builds a country, state, or province menu, that often comes from the database tier. But, that data is so slow-changing that it could just as well come from the web service tier. Even better, client-side JavaScript could have a function to create such menus.

4.14 Use the built-ins

In most languages, built-in types run faster than classes. For instance `int` is likely faster than `Integer`. Just as likely, an array is likely faster than using a vector or a list or the like.

5. Big Data

Many database publishers offer "federated" database systems, which are effectively very large and fast big data systems. The hardware and software is relatively expensive, but fast. Possibly the biggest advantage of these federated systems is that little or no special code is required for their effective use.

4.15 Use the best database for the job

If your data is unstructured, store it as chunks of data. If your data is structured but not tabular, store it in an unstructured database, either special-purposed, or general, such as MongoDB. If your data is tabular and does not have a small numeric natural key, and is not logically sequential, use a relational database. If your data is tabular and has a numeric natural key, use random access without a database system. If your data is logically sequential, store it that way.

Afterward

Following any of these suggestions will, so a greater or lesser extent, increase the responsiveness, speed, and cost of your system. You were warned in the "engineering triangle", which states "Good, Fast, Cheap — Pick two".

There is also a monetary cost associated with the abstraction approach. I won't argue here whether using an abstraction stack makes writing an application cost more or less, but I can assure you that it makes it harder to hire in the future. This is best shown by example. Let's say that I have an application built on this stack: HTML, SASS, CSS, Angular.JS, JQuery, JavaScript, JSP with some tag libary, Java, Hibernate, Hadoop, and Oracle. When I need to hire a new person, my job listing requires that the person know 12 items. If I exclude abstraction layers and go with HTML, CSS, JavaScript, Java, and Oracle federated server from Day 1, my job listing will only require 5 items. I could make the list even shorter by trading Java for server-side JavaScript (Node.JS), but then I'd take a speed hit and a debugging time hit.