NORTHWEST NAZARENE UNIVERSITY

The Brain (Concurrent, In-Memory Caching)

THESIS

Submitted to the Department of Mathematics and Computer Science
in partial fulfillment of the requirements for the degree of
BACHELOR OF SCIENCE COMPUTER SCIENCE with a focus in DATA SCIENCE

Orion Trotter

2022

THESIS

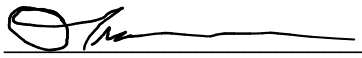
Submitted to the Department of Mathematics and Computer Science in partial fulfillment of the requirements for the degree of

BACHELOR OF SCIENCE COMPUTER SCIENCE with a focus in DATA SCIENCE

Orion Trotter

2022

The Brain (Concurrent, In-Memory Caching)

Author:  4/28/2022

Orion Trotter

Approved: Kevin S McCarty 4/28/2022

Kevin McCarty, Ph. D., Department of Mathematics & Computer Science, Faculty Advisor

Approved: Dean L. Matlock 4/29/2022

Dean Matlock, Department of Criminal Justice, Assistant Professor

Approved: Barry Myers 4/29/2022

Barry Myers, Ph. D., Chair, Department of Mathematics & Computer Science

ABSTRACT

The Brain (Concurrent, In-Memory Caching).

TROTTER, ORION - Department of Mathematics & Computer Science

Databases serve as a frequent medium for applications requiring the parsing of big-data. Concurrent cloud database services ease and expediate the sequence and complexity from data-source through the database to the application's domain logic. Applications that require near real-time precision in the timing of their data acquisition, like the trading application explored, are overtly hindered by distance from their source and any additional machine-level complexity introduced by establishing new connections with external connections. We see efforts to mitigate such latency in stock trading with companies such as Polygon.io, that establish servers that are physically present and connected to the servers at the various stock exchanges. Where, therein, one disconnects latency inducing intermediaries to receive and parse the raw data is the problem addressed here. Here we utilize principles of computer architecture and operating systems to store the data in-memory, serving as a cache for the domain logic and user-interface. Once the cache reaches a specified capacity, the data is stored in a database for analysis and reference on a parallel thread. This resulted in a more than 7000x reduction in latency and simplify integration with future plug-ins.

ACKNOWLEDGEMENTS

Thank God for the blessing of curiosity that provides the foundation for all exploration and discovery, resulting in the joy we feel during and after, as well as the mutual benefit we all receive from offering such discovery to others. Without my wife's patience and support, there would be little progress with this project. She's always proven vital to progress within any work I've done. Dr. McCarty, thank you for your constant guidance and bringing me into this project! Your faith and trust in me not only benefitted the work of this project but brought me to a point in my careers I don't believe would've been possible (in so short of a time at least) otherwise. Dr. Hamilton, for seemingly unlimited reasons as well. Your support and passion behind all subjects Computer Science brought me to Northwest Nazarene University and bolstered my confidence throughout my years at the university. Dr. Myers, thanks for holding the bar tight. Of course, my parents and brothers, without whom I would not be the man I am today.

TABLE OF FIGURES

TABLE OF FIGURES

PROJECT BACKGROUND

Initial Idea

An automated trading application requires recent market data to evaluate, an algorithm to execute such evaluation, and a service to execute trades. Architecturally this becomes more complicated. Since the amount of market data incoming can be around 100 gigabytes per hour, it's typically sifted algorithmically, and important data stored in a database. The process of live information arriving to a system (in this case from New York to Idaho), storing that information in a database, calling procs to process such information from the application, and retrieving multiple datapoints, then making a REST call from the application to San Francisco (which then uses a TCP socket to New York), introduces significant latency over a simulation with data points loaded and 0 delay between call and execution. Latency, in this case refers to the amount of time from when one wishes to execute a trade and the trade is executed. A simple example: if it takes 4 seconds to retrieve a large group of stocks and identify the stock for which to execute a trade, then 3 seconds to send and execute such a trade, we have 7 seconds of latency. The major issue with algorithmic trading latency is that when algorithms turn around stocks quickly based on very small profit margins, the margin can disappear quickly with volatile stocks. An example of such: AAA is trading at $7.05. 4 seconds later it's at $7.11 and the algorithm picks it up as a trade (still at $7.05), sending off to purchase with a sell point of greater than 7.11, the stock has raised to 7.15 in the 3 seconds it takes to execute and begins to decrease. The 7 second cycle can execute again with the stock on a downward trend. In simulation purchases and sales occur on the timestamp of when the data is collected. Live trading had a discrepancy akin to the aforementioned.

An initial need was to minimize any latency in retrieving market data from New York and executing trades. This need led to the idea to develop an in-memory cache for all real time data. The cache would be developed for concurrency and small-footprint performance. Signals, in the application's case delegates or events, are sent to multiple locations immediately and the same information is moved to long-term storage according to preset intervals. Just like the human brain's deviation of responsibility between the prefrontal cortex (short-term memory) and neocortex (long-term memory), the responsibility of the in-memory cache is to render the most recently retrieved information immediately available, and via another thread store slowly retrieved long-term information in a database. The project is thus referred to as "the Brain".

Other Caches

.NET provides two caching libraries: MemoryCache, which serves as a similar in-memory cache, and DistributedCache, which serves as a concurrent, heap-allocated distributed cached (Microsoft, 2021). Both caching libraries require interface implementations along with heavy generic libraries which introduce several rounds of boxing and unboxing. This process of boxing, converting value types to heap-allocated, generic objects and reallocating those objects to other value or reference types, is burdensome and exponentially slower than working with direct value-types. The MemoryCache, while still boxing objects by virtue of its genericism, rapidly relinquishes heap space by replacing values of the same type or capacity. It further allows an option to set time-limits for objects stored withing memory. With the performance requisites of live-trading, a more catered solution was required.

PROJECT OBJECTIVE

Brain Objective

The Brain's objective can be summed up to reduce the latency to the lowest amount of time possible while maintaining all of the integral data points for analysis and later processing. Given the desire for expanding tools and integrate later functionality or algorithms, responsibility between classes (or in this case projects) needs clear distinction. The Brain (a single project) will process and store data in large concurrent collections, make any connections external to the application, signal changes via subscriptions, and maintain macro level trades. Macro trades become important when rules are desired regardless of the trading algorithm implemented. Macro trades become especially important when multiple instances of trading algorithms are utilized.

Tertiary "Core" Objective

The trading logic, hereafter referred to as the "Core", will need refactoring to handle only data available in the Brain. REST Calls and DB queries need removal from all portions of the application's logic, to include its non-Brain dependencies. This allows for one point of focus for improving network performance (the Brain), and one point of focus for improving machine-efficiency (the Core), and one point of focus for improved concurrent output and data structures (the Brain). The separation of logic allows for easier distinction of responsibility and a great reduction in the time intensive operations required by the trading logic.

Successful concurrent operations in the Brain and reducing heap allocations are key to making this successful. The initial idea diagram for the Brain's architecture can be found in figure 1.

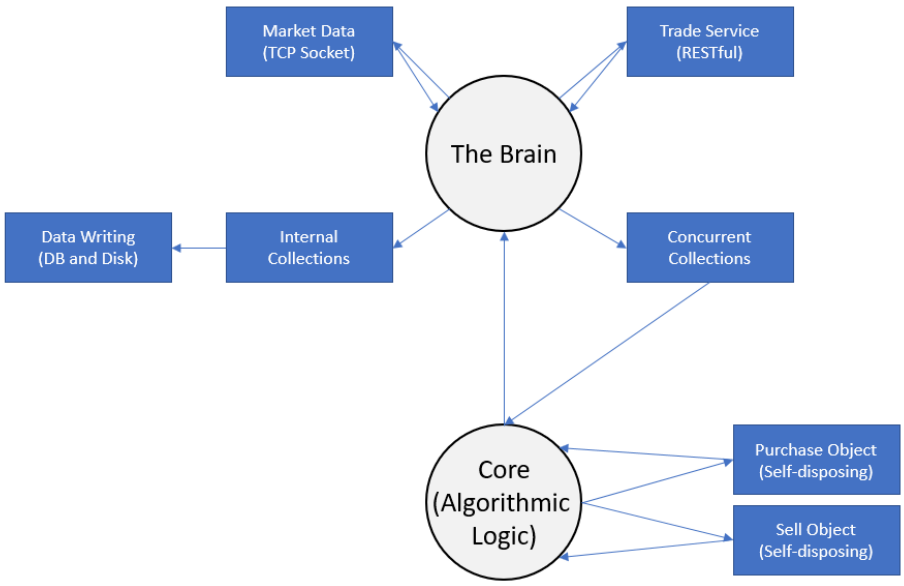


Figure 1 Macro Level Diagram

IMPLEMENTATION

The initial approach utilizes principles of object-oriented design and event driven concurrent programming. While no strict paradigm was kept in mind, the first step in designing was to template a division of responsibility between the Brain and the Core. Interdependent objects with singular responsibilities are created within both applications. Concurrent event handlers hold the responsibility of communicating events in the Brain. Upon initiation of a Core application, methods external to the Brain subscribe to pertinent events in the Brain. Given the outline provided in figure 1, creating the Brain was the first step, removing any non-trading logic from the core application the second, and then creating new data structures for concurrent operations in the Brain, reducing heap allocations, and improving performance in both the Brain and Core applications final.

Creating the Brain

An abstract base class entitled “MemberBase” is utilized for all responsibilities of the Brain. This ensures expansions follow the same rule set for scalability. Each member contained a pointer to its “Master” to contact other parts of the Brain and utilized configured variables. Furthermore, each class was divided into a minimum of 3 partial classes for easy navigation: the unnamed base for storing variables, an initiator to initiate variables and pertinent operations, and a closer to dispose of variables and run closing operations. A rule of simplicity within a document was used, “KISS” (keep it simple stupid) being the guiding principle even throughout a complex application. An example of the abstract base class can be found in figure 2.

```
namespace Brain {
    11 references
    public abstract class MemberBase {
        5 references
        protected MemberBase(Master master) {
            Master = master;
            Master.Members.Add(this);
        }
        66 references
        protected Master Master { get; init; }
        5 references
        internal abstract void RegisterToken(CancellationToken token);
        5 references
        internal abstract void Initiate();
        5 references
        internal abstract Task Close();
    }
}
```

Figure 2 MemberBase.cs

While the “Master” class does not inherit from MemberBase, it follows the same structure of divvying into partial classes with its unnamed variable store, an initiator, and a closer. Beyond the three mandatory partials are two socket configurations and MacroCalls which allows for external applications to make a call across the Brain’s application.

The “Data” member class holds the responsibility of processing and storing the data of the Brain. Aside from its initiator, closer, and unnamed partial classes, Data holds a “CollectionAccess” member for application-external and application-internal access to its collections and two socket data processors for receiving raw data and deciding which data types to use and collections to store such data.

The “DBAndDisk” member class expands upon its basic responsibilities with a DataDump class for emptying the collections in “Data” and storing them in any database or disk based upon settings passed on the Brain’s initiation. “Orders” processes order data for storage. “Snapshots” processes market data for storage. Given historical data is needed for analysis and potentially for application strategies, a concurrent system of storing historical data as close as possible and without impacting the general speed is necessary.

The member class “External” maintains responsibility for any calls external to the over-arching application (i.e., to other services), parsing such data, and passing it to the correct processor. Aside from its basic member partial classes, it contains “MacroTrades” which applies proprietary trading algorithms, “PolygonOps” which obtains market data, “REST” which makes RESTful calls, and “Trades” which makes trades.

“Subscriptions” (a member) handles both internal and external subscriptions to events in the Brain. In addition to its member functions, “Data” handles internal data related events and “Master” handles events that impact operations across the brain.

Non-members (classes that do not inherit from “MemberBase”) include a modified CancellationToken for cancellation of parallel operations, and a proprietary class for trading (non-essential to this project).

The final solution structure can be found in figure 3.

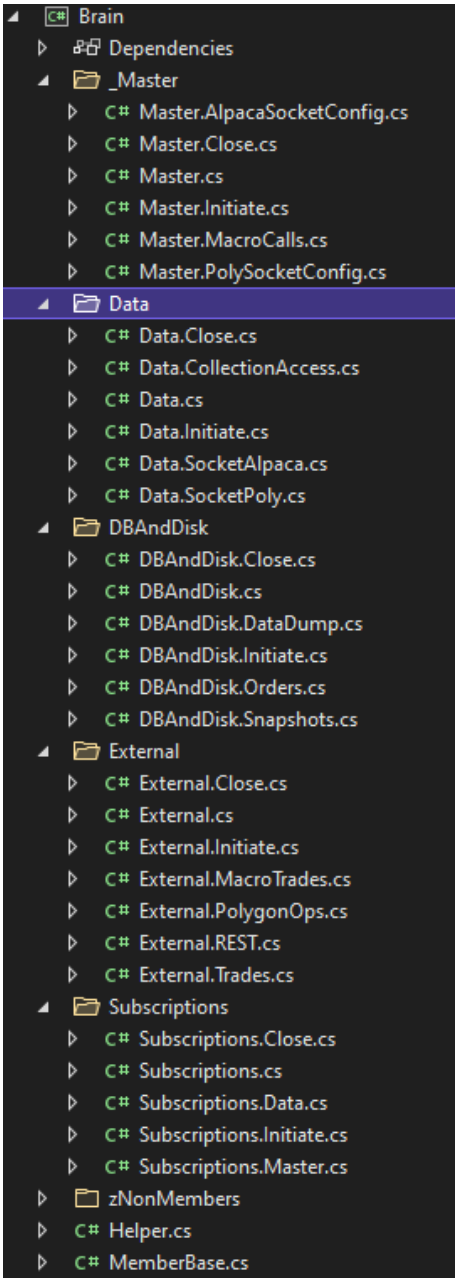


Figure 3 Final Solution Structure

Modified Data Structures, Networking, and Data Types

Careful parallelism is necessary for implementation of the Brain. Concurrent data structures are necessary in to allow for multiple algorithms to read data from the same

collections while others write data to them. The market data socket received new data in nanoseconds rendering a need for parallel events in the case that certain subscribed methods could not finish executing prior to another need to invoke an event. A means to prevent freezing the application until data was stored is needed. Standard library data structures are too broad and thus, proved too slow for our purposes. As a result, several standard data structures were replaced with reduced footprint, concurrent imitations that may implement memory manipulation and copying instead of traditional heap allocation and iteration. Furthermore, after implementing a socket that could parse raw market data, random access memory (RAM) becomes an immediate concern if running on a single server.

To resolve the concurrent collections dilemma, required data structures were created in a GeneralLibrary for use across the application and prefaced by the name “ThreadSafe”. The underlying data structure to use is stored, privately, in each thread-safe class with an additional monitor. Typically, a monitor, similar to a mutex, but for internal use only and significantly faster processing, is used to queue thread-sensitive operations in public methods. For iteration across the collections, copies of the collection at that time iteration is necessary are made and an iterator is returned for the copy until iteration has completed and the copy is disposed.

```
/// <summary>
/// Returns an Enumerable of an in memory copy of this dictionary.
/// </summary>
1 reference
public IEnumerable<KeyValuePair<Key, Value>>GetEnumerable() {
    lock (syncLock) {
        var copy = new Dictionary<Key, Value>(dictionary.Count);
        copy = dictionary;
        return copy;
    }
}
/// <summary>
/// Returns an Enumerable of an in memory copy of the Keys in this dictionary.
/// </summary>
0 references
public IEnumerable<Key> GetEnumerableOfKeys() {
    lock (syncLock) {
        var copy = Keys;
        return Keys.AsEnumerable();
    }
}
/// <summary>
/// Returns an Enumerable of an in memory copy of the Values in this dictionary.
/// </summary>
0 references
public IEnumerable<Value> GetEnumerableOfValues() {
    lock (syncLock) {
        var copy = Values;
        return Values.AsEnumerable();
    }
}

1 reference
public IEnumerator<KeyValuePair<Key, Value>> GetEnumerator() {
    var copy = GetEnumerable();
    foreach (var thing in copy) {
        yield return thing;
    }
}
```

Figure 4 Example for Iteration

This allows for simple C# iteration across the data structures like so:

foreach (var thing in dictionary)

Events are created in a similar fashion with an empty object (to serve as a monitor) for locking, and a base delegate and event. A simple thread safe event with no parameters and returning void looks like figure 5.

```
/// <summary>
/// Operates as a thread-safe version of the Action delegate
/// </summary>
9 references
public class ThreadSafeAction {
    object eventLock = new();
    public delegate void Event();
    event Event _threadSafeAction;
    5 references
    public static ThreadSafeAction operator +(ThreadSafeAction eve, Event method) {
        eve.Add(method);
        return eve;
    }
    2 references
    public static ThreadSafeAction operator -(ThreadSafeAction eve, Event method) {
        eve.Remove(method);
        return eve;
    }
    1 reference
    public void Add(Event method) {
        lock (eventLock) {
            _threadSafeAction += method;
        }
    }
    1 reference
    public void Remove(Event method) {
        lock (eventLock) {
            _threadSafeAction -= method;
        }
    }
    3 references
    public void Handle() {
        lock (eventLock) {
            if (_threadSafeAction != null)
                _threadSafeAction();
        }
    }
}
```

Figure 5 ThreadSafeAction

Allowing one to add a method as simply as show in figure 6.

```
Parent.Brain.Subscriptions.OnSnapshotGathered += TradeMeister;
```

Figure 6 Adding a void method to a thread-safe action

SemaphoreSlim, a slimmed down version of a semaphore provided by the .NET standard library, allows for locking faster than a mutex and timeout operations across applications. These were used to process several macro-operations across the Brain.

C# allocates classes, by default, to the heap. Structs and local primitives are typically stored in the stack. Several data types were then converted to structs when possible, arrays of structs, and structs of arrays, to limit heap allocations within methods. Raw data is parsed as memory, instead of converting to strings or collections, if modification is necessary. An example from the market socket displays such practice:

```
ArraySegment<byte> bytesReceived;
var buffer = new byte[6000000];
byte[] resArr;
int offset = 0;
WebSocketReceiveResult result;
int total;
JsonDocument jDoc;
PolygonJsonDoc.PolygonMessage msg;

while (!_cToken.IsCancellationRequested) {
    bytesReceived = new(buffer, offset, _maxFrag);

    try {
        result = await _socket.ReceiveAsync(bytesReceived, _cToken); //converts bytes to UTF8
    }
    catch (Exception e) {
        handleException(e);
        break;
    }

    if (!result.EndOfMessage) {
        offset += result.Count; //if it's not the final frag, then increase by the number of bytes from the result
        continue;
    }

    total = offset + result.Count; //total count of bytes including prior fragments and the result
    resArr = pool.Rent(total);
    Buffer.BlockCopy(buffer, 0, resArr, 0, total);
    using var stream = new MemoryStream(resArr, 0, total);
    offset = 0; //reset the offset

    try {
        jDoc = await JsonDocument.ParseAsync(stream, default, _cToken);
    }
    catch (Exception e) {
        handleException(e);
        break;
    }
    finally {
        pool.Return(resArr);
    }

    try {
        msg = PolygonJsonDoc.ParseMessage(jDoc.RootElement.EnumerateArray()); //returns a struct with 2 parts: message & and array of tickers
        jDoc.Dispose();

        if (msg.Message is not null and not "")
            OnMessage.Handle(msg.Message);
        if (msg.Tickers is not null)
            OnUpdate.Handle(msg.Tickers);
    }
```

Figure 7 Example pipe from socket

The ArraySegment<T> represents a pointer to a segment of an array (for example, indexes 0 to 200). In this case a large buffer is created in the method's stack memory upon the initialization of the socket, and bytesReceived points to a section of the buffer in memory. The fragment from the socket is received as an asynchronous stream (WebSocketReceiveResult) which allows processing before all the data is received. Based on header information we can increment the offset and continue receiving data until the final fragment is received with new pointers to each offset in the array. This allows us to begin processing from multiple parts of the array before all of the information is allocated. Another array is rented from a global array pool and the buffer is then copied as a block of memory to the rented array (resArray) so the buffer is free to begin processing other messages. The rented array is then passed to a MemoryStream which creates a stream of binary data for processing by a custom 8 bit Unicode Transformation Format (UTF-8) parser that finally passes it to a thread-safe delegate that the Brain subscribes to.

Streaming and array re-use allow for exponentially reduced time for execution and heap allocation. The call to an asynchronous UTF-8 stream creates a token to the first UTF-8 segment and allows processing of the message into an in-memory struct before the message is fully-received or converted. Once processing is finished, it is disposed. The renting of arrays allows the re-use of arrays which prevents the need for reallocation of up to 6 megabytes every time they're needed. This concept is used throughout the application.

CONCLUSION

The Brain is successful given the objective of removing latency discrepancies from the original application. Latency from the algorithm's execution of collecting market data to making a decision is reduced from 7.2 seconds to <1 ms. The retrieval of data is so fast that it's

intentionally throttled to 500ms. The resulting reduction of trade time is from 2.1 seconds to 49 milliseconds. Much of the latency was the amount of time it took for:

- a collection of market data via REST
- process such data in the application
- store data in the database
- process information in the database
- retrieve processed information in the application
- process information in the application
- execute trade via REST call based on processing.

The new process is:

- retrieve data via socket
- process data concurrently in memory
- execute trades concurrently via socket.

Disparities between simulation and live trading still exist but identified as algorithmic distinctions and profit is typically in-line with simulation (as opposed to prior).

FUTURE WORK

The Brain is currently designed to require a new instance for each user. This means we can only support a single instance on one machine However, this requires maintaining a separate instance with the data provider and added stress to our internet connection. A better alternative would be to acquire data via the single external connection and propagate it over the network as a distributed repository. Apart from separating customized user settings, this would require

converting several of the monitors to semaphores and modifying some collections. The upside to doing so would be reduced costs in storing data in memory (each instance requires about 64 GB of RAM per user), the downside is reduced performance (with more concurrent reads to the same collections, the performance will inevitably go down). Performance deterioration can be mitigated via concurrent messaging systems like MassTransit.

REFERENCES

Microsoft. (2021, 12 11). *Caching.* Retrieved from Microsoft Documentation: https://docs.microsoft.com/en-us/dotnet/core/extensions/caching