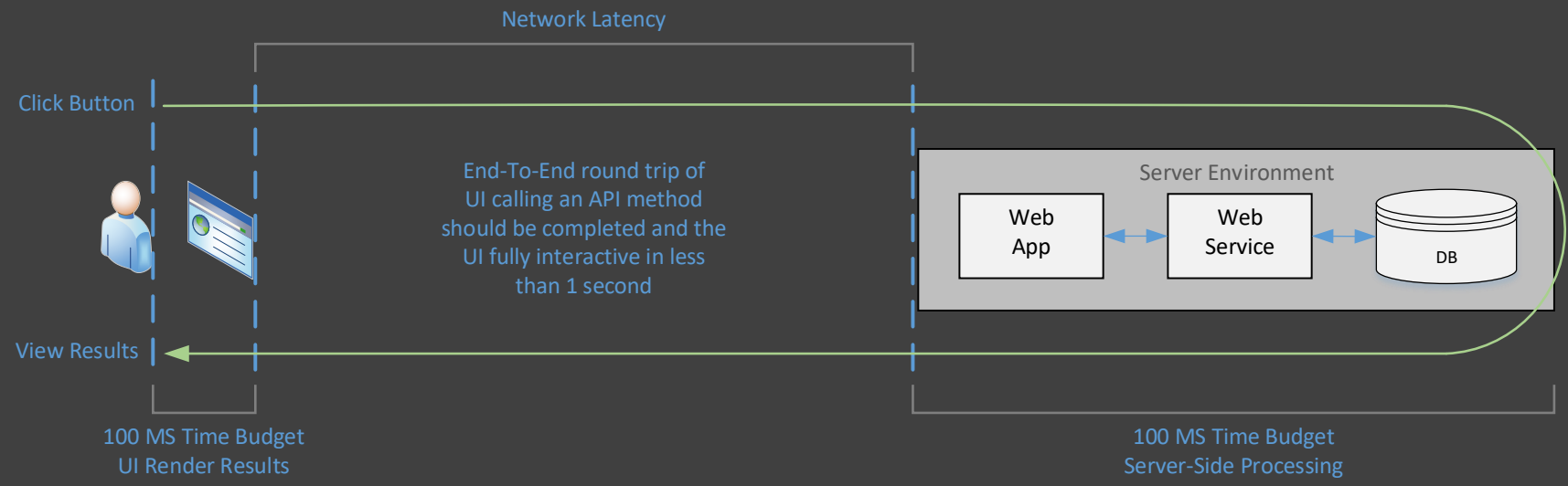
Performance Engineering

DGP has been designed and built to deliver high performance executables. This is necessary because of the end of Moore's Law, which means that using newer, faster hardware to cover for the problem of poor software performance has not been an option for many years. Going forward, performance improvements can only be achieved by improving the design of the architecture and code of the software systems themselves. High performance is also a necessary prerequisite to enable the centralized hub-and-spoke architecture to be a viable alternative (slow web service APIs would result in a system that was effectively unusable).

The overall standard for system performance is measured end-to-end from the user of a system. Each action of the user should be completed, with the UI returned to an interactive state, in less than one second. In addition, these performance measurements should be made while a system is under realistic workloads whenever possible (almost all systems perform well for a single user).

One second is an important standard due to Human physiology, which is explained in this video from a Google engineer (Ilya Grigorik). The video is geared toward mobile web apps, but is applicable to all systems.

https://www.youtube.com/watch?v=Il4swGfTOSM&list=PL1B4F4863AEE2B122&index=32&t=29s



In this context, one second becomes a time budget. For example, due to the limited bandwidth and high latency of mobile networks, 100 MS is usually going to be the maximum server-side duration to meet the overall one second response time requirement, and 100 MS is reserved for the UI to render the results. This leaves 800 MS maximum for network latency.

DGPDrive measures the end-to-end duration of each API request and response, plus the unit testing and server-side performance monitoring functionality built into each API method to display the performance of each API method call in the status bar at the bottom of the UI. Typical server-side processing of the DGP API methods generally averages from 15 MS down to 2 MS, depending on how many cache hits occur at the web server, web service and database server levels. Once the various caches are populated, the average end-to-end latency is basically the same as the network latency between the client app and the web server.

Designing and building software systems that have very high performance provides multiple benefits to the organization:

- High performance systems help to provide an excellent user experience, which is beneficial in and of itself.
- High performance enables the use of the centralized hub-and-spoke architecture that consolidates all of the logic and data of a system in the servers of the middle tier, which improves the security of a system and greatly simplifies maintenance, debugging and deployments. However, a hub-and-spoke architecture is only feasible if the services of the middle tier are very fast.
- High performance executables improve the scalability of a system, decreasing the amount of hardware and software infrastructure needed to support a given number of concurrent users, thereby reducing the total cost of a system.
- Finally, the simplification of a system's architecture and code is a necessary prerequisite for high performance systems:
    - a smaller code base decreases the number of staff needed to develop and test the system
    - a simpler code base allows junior developer to maintain it, which reduces the cost of the developers
    - the smaller, simpler code base decreases the TTM for releases and MTTR for maintenance

High Performance Engineering References

An excellent book that has served as a guide for many of the performance improvements in DGP is Ben Watson's *Writing High Performance .NET Code* [https://www.writinghighperf.net/].

High Performance Distributed Systems

In practice, delivering high performance systems can only be achieved using a process to both simplify and consolidate a system's architecture and code. Consistently measuring the end-to-end performance of a system is the best way to verify that unnecessary complexity is not being added to a system over time as it constantly evolves to meet new requirements. In addition, simplicity and other related characteristics are extremely important to the overall quality and reliability of a software system. Having the ability to measure and verify an approximation of the simplicity of a software system as it constantly evolves over time is extremely useful.

High Performance Engineering Overview

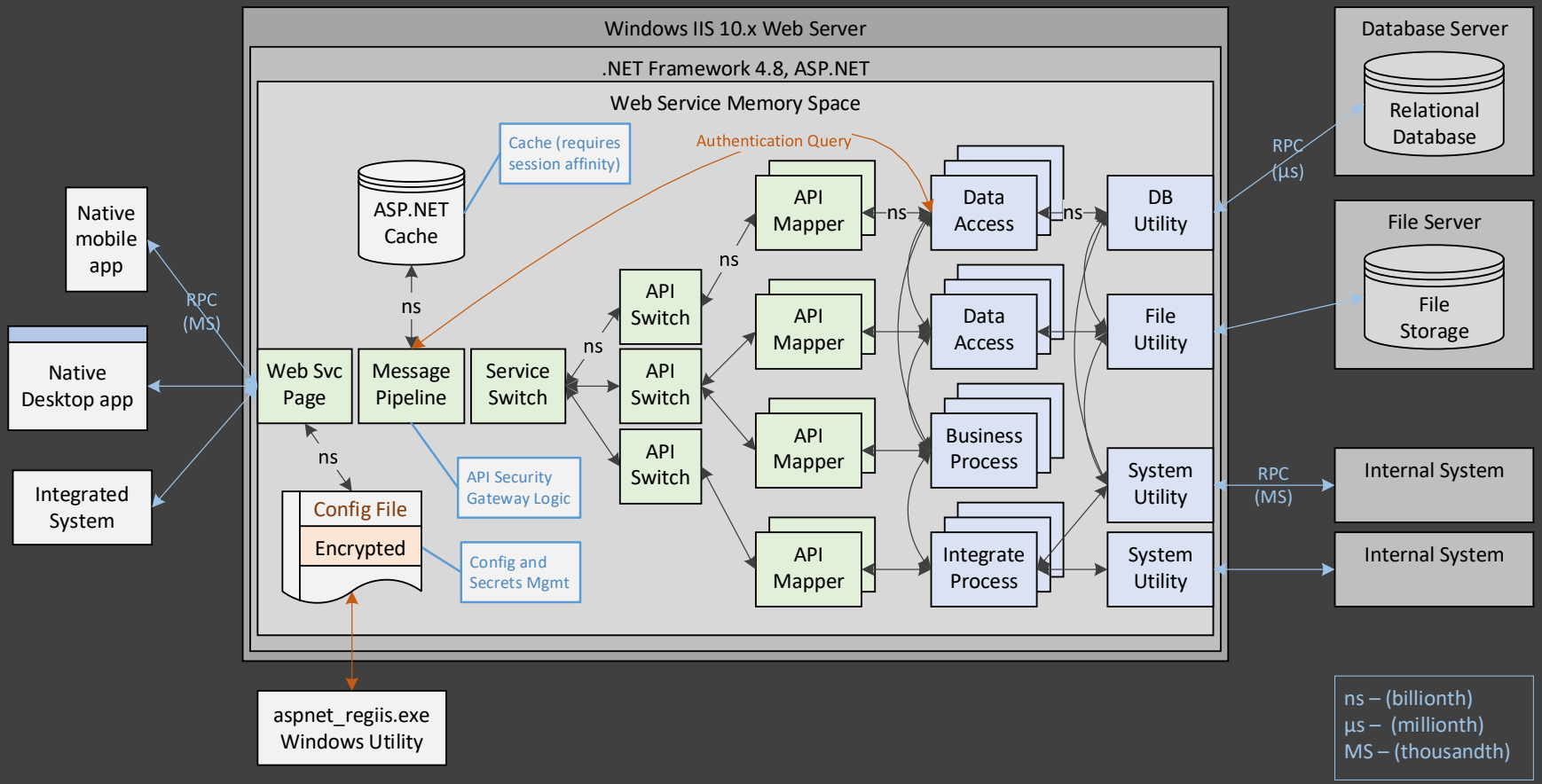
Improving the performance of software systems is primarily achieved via two mechanisms: simplification and consolidation.

Simplification is the process of designing software systems to reduce unnecessary complexity to an absolute minimum (a certain amount of complexity is unavoidable). There are multiple levels of simplification. First, at the architecture level, a system is simplified by finding ways to eliminate the need for as many subsystems as possible. This practice will streamline the architecture so that it only includes the functionality that is truly needed by the system. Second, the remaining subsystems should be designed so that they serve multiple purposes within the system whenever possible – further reducing complexity and the number of "moving parts".

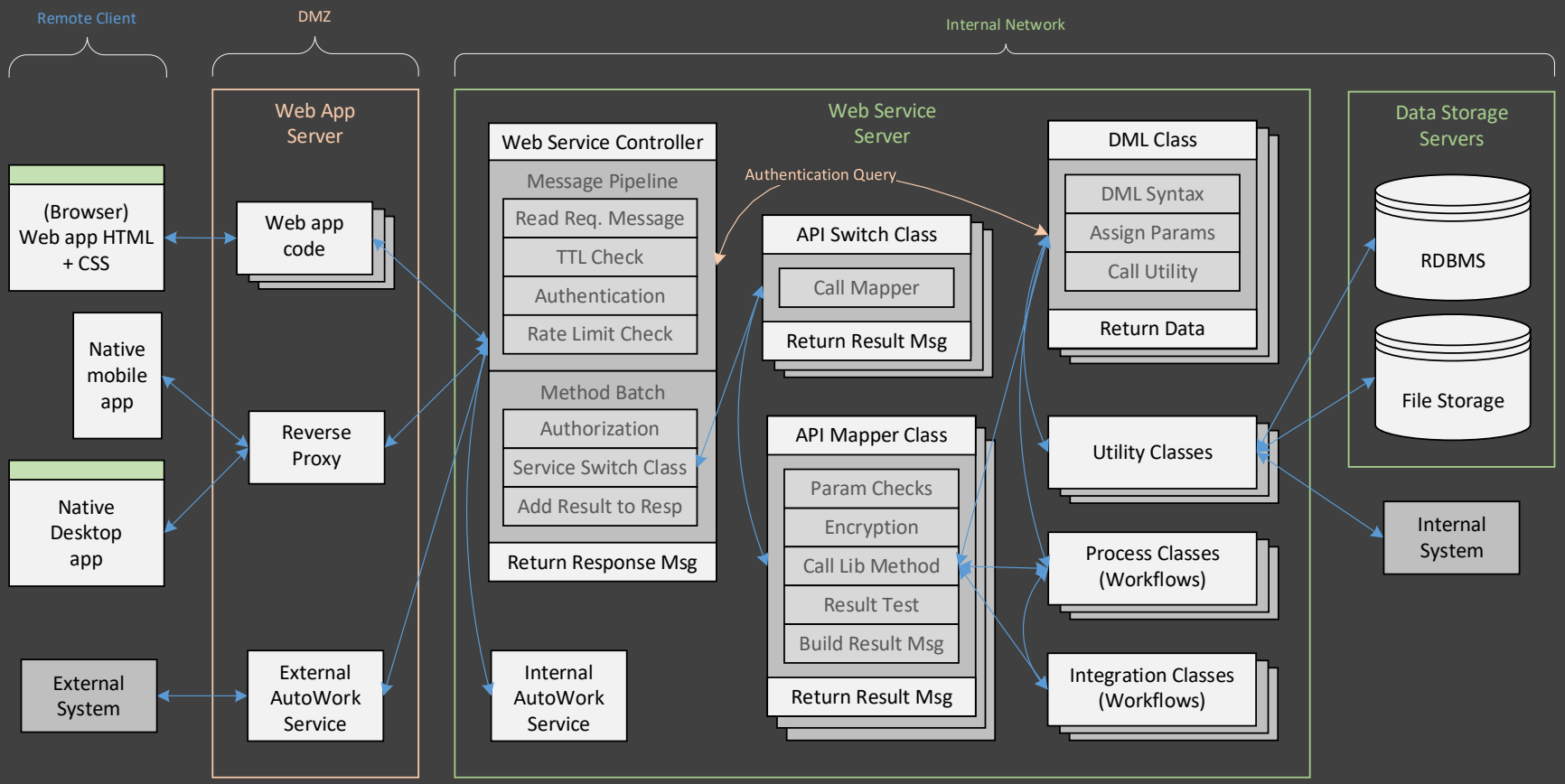
Consolidation of system logic in a hub-and-spoke architecture:

- The majority of the logic in a hub-and-spoke system is centralized to run on the servers of the middle tier, which significantly reduces the latency of inter-process communication within a location, and also improves the security of those interactions since they occur within the memory space of each server or within the private internal LAN.
- As much of that server-side logic as possible is implemented as reusable code libraries that run within the same memory space of each web service worker thread, reducing the latency of inter-process communication to nanoseconds.
- Fourth, the code that is normally scattered among many classes and methods of a typical OO system is consolidated into significantly fewer classes/methods in order to help reduce the rate of object instantiation and simplify the work of garbage collection (very important to improve the performance and efficiency of the .NET web service APIs).

The diagrams below show the consolidation of the system architecture, replacing discrete subsystems for security, caching, configuration data management, and secret data management with built-in functionality, and then getting as much of that functionality to run within a single memory space as possible. The relative speed of each type of call between the components is shown as contrast to the slow inter-process communication of decentralized distributed systems.



For example, inter-process communication of web services calling other web services over a WAN or the Internet have end-to-end performance measured in milliseconds (thousandths of a second). Those same RPC calls through an access switch within a rack result in end-to-end performance that is measured in single-digit milliseconds, or even as microseconds (millionths of a second) in some cases once the various caches have been populated.



Finally, the end-to-end performance of libraries of code calling each other when running in the same memory space is measured in nanoseconds (billionths of a second). This type of inter-process communication is made possible by the fact that all the logic in a system is written as reusable libraries of code (shown in blue in the diagram above). There is very little code in the web services themselves (shown in green), which are mainly used as a secure mechanism to remotely call the functionality of the code libraries.

The various parts of ASP.NET that are used for managing configuration data (web.config file, app.config file) and caching of system data (the Application object, the Cache object, the Session object) are also available within the same memory space of the .NET Framework. This built-in functionality not only simplifies the system architecture by eliminating the need for discrete subsystems, but this level of consolidation reduces the cost and greatly improves both the performance and efficiency of the system.

The difference in performance experienced by the end users can be dramatic. Systems with widely distributed and slow inter-process communication (common in micro-service architectures, for example) will frequently see users waiting between 2 to 5 seconds for UI actions to be completed. In contrast, the wait time for a high-performance hub-and-spoke system is typically a small fraction of one second over broadband networks and approximately half a second over slower mobile networks.

Server-side processing (web service APIs)

In terms of the code itself, the best ways to improve performance is to a) use languages that produce natively compiled binaries, and b) move from pure OO techniques to a hybrid of OO/procedural style of code (similar to Go, Rust, etc.).

General Types of Programming Languages:

- Interpreted (Python, JavaScript, TypeScript, etc.) – high developer productivity
- Managed (Java, .NET, Go) – a balance of both high developer productivity and high-performance native executables
- Low level (C/C++, Rust) – high performance native executables with lower developer productivity

For most Line-of-Business (LOB) applications, managed code provides the best balance of delivering high performance native executables and also high developer productivity. The examples below apply to the .NET framework and C# programming language. Additional info is available in the Implementation Overview documentation showing the NFR's and implementation techniques.

High Performance System Architecture Summary

1. Simplification of the Architecture
   a) Remove all possible subsystems from the design – high level simplification
   b) Remove all possible layers of abstraction from the design – high level simplification
   c) Modify the architecture so that each part of the design serves multiple purposes in the system, further reducing the total number of components in the design.
2. Consolidation of System Logic
   a) Centralize the architecture such that as much of the required functionality as possible can be run within the LAN of the middle tier servers (hub-and-spoke architecture).
   b) Consolidate the logic on the server tiers so that as much of the required functionality as possible can be run within a single memory space of each web service worker thread.
3. Consolidation of the Code
   a) Reduce the rate of object instantiation
      i) Move beyond pure OO techniques to a hybrid approach that combines OO with procedural programming in order to consolidate code into fewer classes and methods.
      ii) Avoid the creation of objects in the LOH or long-lived objects that are promoted to Gen2 memory segments.
      iii) Reuse long-lived objects in Gen2 instead of instantiating new ones (public static, pools, etc.).
4. Native Binary Executables
   a) Choose development tools that deliver natively compiled binaries rather than interpreted code.
   b) .NET Framework memory management.
5. IO Completion Ports (multi-threaded scalability)
   a) Server-side processing time partially determines the number of requests that can be processed per second by each CPU core.
   b) All IO-bound processes are handled as IO completion ports / network handles, allowing the process threads to be freed and returned to the thread pool for the duration of the network communication. All calls made over a network are IO bound.

Rules for Measuring Performance Metrics

1. Only end-to-end measurements of the latency (time spent waiting) experienced by users of the software are useful. Only these types of end-to-end metrics can be used to compare very different software systems to each other. That means that the performance of a system must be measured by the distributed remote applications provided to users, and testing the system's functionality should also work in the same way.
2. Only measurements collected while a system is under realistic workloads are accurate and meaningful. Collecting performance measurements will small numbers of concurrent users will produce unrealistically good metrics. Generating synthetic workloads at sufficient scale to simulate realistic workloads can be very difficult and expensive. The best way to accomplish this in practice is to collect end-to-end performance measurements during the actual use of a system in production, without affecting the system.

Note1: when considering the performance of distributed software systems, Little's Law (queuing theory) is very useful. The takeaway from Little's law is that the latency experienced by users is the single most important factor, and that is what should be minimized.

Note2: variations in the performance measurements of a given system will occur due to the unpredictable combination of several features:

- Caching. To state the obvious, cache hits will improve performance, while cache misses will cause a noticeable decrease in performance. There are multiple levels of caching that are built into the server-side tiers of DGP Lattice:
    - IIS web server – by default, IIS application initialization can take a significant amount of time. From version 8.0 and later, IIS can be configured to pre-load a web service, greatly reducing the time needed to reply to a first request.
    - .NET framework – by default, the .NET framework JIT compiles and loads each method into memory the first time it is called. NGEN and .Net Native offer ahead-of-time (AOT) compilation to significantly improve application startup times.
    - DGP – the core web services cache reusable data in memory using the ASP.NET Application and Cache objects. A cache miss requires an RPC query to the database server.
    - SQL Server – SQL Server caches queries, query plans, pages of data, indexes, etc. in memory as much as possible.
- Network congestion will vary from time to time, and can have a big effect on the network latency metrics.
- Concurrent processes sharing host resources will vary from time to time, and can have a big effect on server processing time (the infamous "noisy neighbor" problem for VM's is one example).

Verification of High Performance

1. *The status bar at the bottom of the DGP Lattice UI shows the end-to-end round trip time and the server-side processing time for the API methods called by the application during normal use in order to prove that the standards are being met in every environment of a DGP system. The calculation of the server performance metrics is designed and built into each web service, and are returned to the client app in every response message. The end-to-end performance metrics are calculated by each client application.*
2. *For members of the RemoteMonitor role, the Lattice application saves the performance metrics displayed in the status bar to the LatticeMetrics table of the SysMetrics database. This is also a form of monitoring the health of a system.*
3. *The API Tester shows the end-to-end round trip time and server processing time for every API method tested in each environment. The results of test runs can be saved to the TestResults table of the SysMetrics database for analysis. Those results can also be saved to a CSV file for import into Excel, etc.*
4. *The AutoWork Tester shows the duration of each automated process, and in many cases also shows the duration of the individual steps within those processes. The performance of the automated processes directly affects the number of concurrent threads needed to handle the automated workload of a system.*
5. *Caching authorization data in each account record results in authentication and authorization querying for a single record from a single table (APIUser) in the SysInfo database. This greatly improves the performance of the message pipeline class.*
6. *Caching the UserInfo object in each web service not only eliminates the query to the APIUser table, but also the need to decrypt the returned password value stored in the table. This results in a significant performance improvement for each API request*