Diagnosing Issues with ASP.NET Core Applications

David Fowler

Damian Edwards

ASP.NET Core

- 2 Major versions
- 1000+ open issues on Github (and more in emails)
- Production Ready

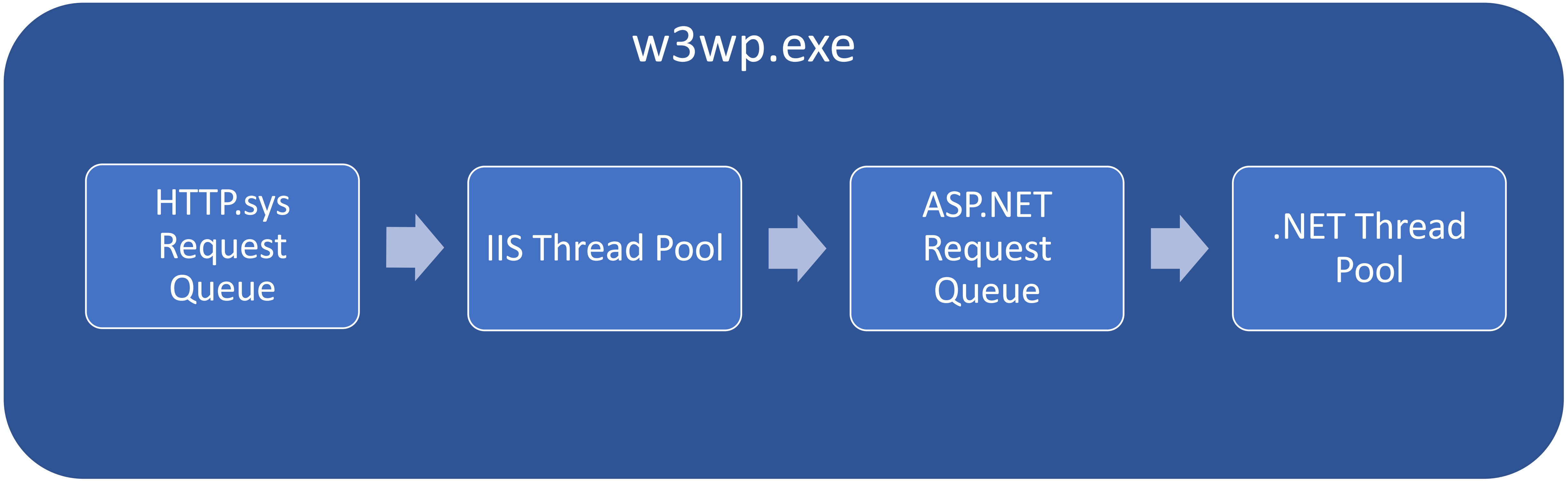
Architectural changes

- ASP.NET Core runs outside of the IIS process (w3wp)
- ASP.NET Core doesn’t have a SynchronizationContext
- ASP.NET Core doesn’t have a request queue
- By default, static files are served from ASP.NET Core itself (not IIS or nginx)
- ASP.NET Core doesn’t shadow copy assemblies
    - Assemblies end up being locked during deployment
    - The IIS module for ASP.NET Core supports app_offline.htm but there’s no cross platform solution
    - WebDeploy automates dropping app_offline.htm, and overwriting files with new files (it also renames the files if they are locked)
- ASP.NET Core doesn’t use AppDomains
    - There’s a single process and single app domain

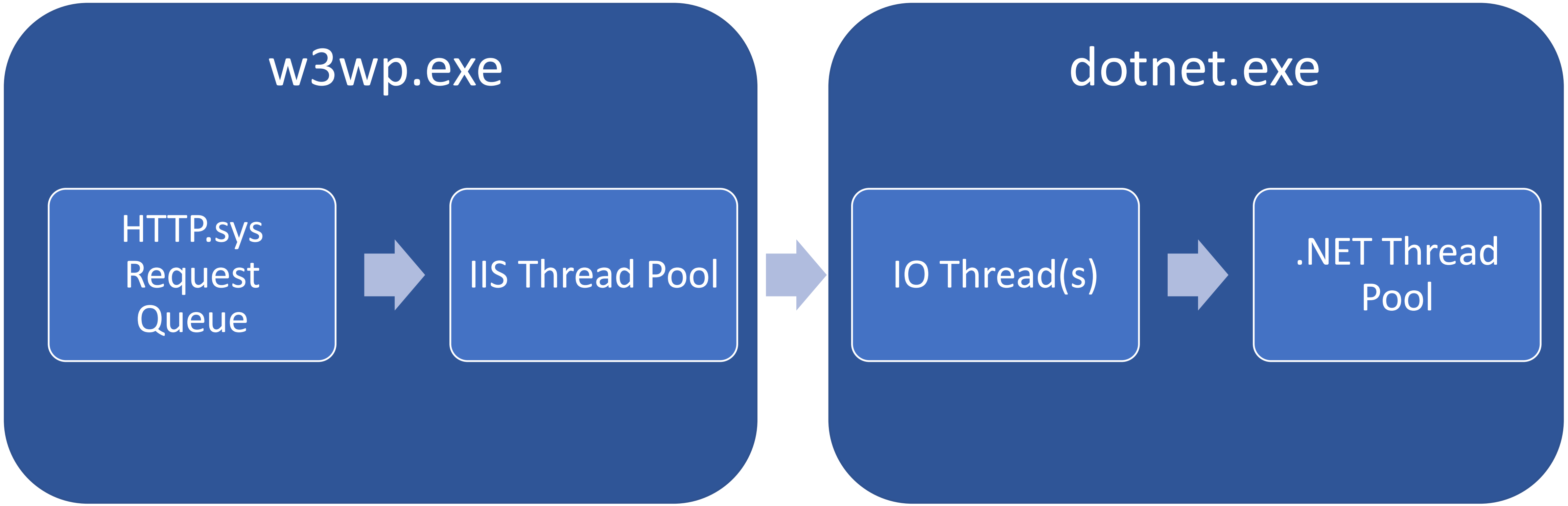
Architectural changes – IIS (ANCM)

- ANCM is the ASP.NET Core Module for IIS
- ANCM uses WinHTTP to communicate with the dotnet process running Kestrel.
- ANCM manages the lifetime of the dotnet process using Windows job objects.
- Port exhaustion can stop ANCM from successfully communicating with the dotnet process.
- Requests that take too long to complete result in a 502.3. ANCM will disconnect the client.

ASP.NET Request Dispatching



ASP.NET Core (IIS) Request Dispatching



No SynchronizationContext

- No deadlocks if you block a Task (Task.Wait, Task.Result)
  - This is **NOT** an excuse to block
- ConfigureAwait(false) has no effect
- Task continuations in ASP.NET Core are queued to the thread pool and can run in parallel
  - This can break code that use to work when porting for ASP.NET to ASP.NET Core
- The HttpContext is **NOT** threadsafe
  - Accessing the HttpContext from parallel threads can cause corruption
- The IHttpContextAccessor makes it harder to detect incorrect code
  - There was a bug in application insights - https://github.com/Microsoft/ApplicationInsights-aspnetcore/issues/373

No Request Queue

- The request queue in ASP.NET is designed to prevent thread pool starvation http://referencesource.microsoft.com/#System.Web/RequestQueue.cs,8
- Blocking thread pool threads can be problematic.
- Handling bursts of traffic can be problematic.
- Moving code from ASP.NET to ASP.NET Core can unveil problems.
  - Code that overuses the thread pool performs more poorly on ASP.NET Core (like waiting synchronously for asynchronous work via Task.Wait/Task.Result)

No Request Queue

- Thread injection rate is slow (2 per second)
- You can increase the minimum threads in the pool by calling ThreadPool.SetMinThreads
  - Beware of the memory limits, 1MB of stack per thread on Windows, 2MB on Linux

Thread Pool Starvation

- Sync over async
  - APIs that masquerade as synchronous but are actually blocking async methods.
- Async over sync
  - Dispatching synchronous operation to thread pool threads (offloading) can have scalability issues that lead of thread pool starvation.
- Blocking APIs
  - Avoid blocking APIs where possible e.g. Task.Wait, Task.Result, Thread.Sleep, GetAwaiter.GetResult()
- Excessive blocking on thread pool threads can cause starvation.
- Diagnose blocking using various tools
  - https://github.com/benaadams/Ben.BlockingDetector
  - Set AllowSychronousIO to false on KestrelServerOptions

Thread Pool Starvation Case studies

- https://forums.couchbase.com/t/frozen-application-after-application-pool-starts-or-recycles-under-load/14573
- https://github.com/aspnet/KestrelHttpServer/issues/2104#issuecomment-337185808
- https://github.com/aspnet/KestrelHttpServer/issues/2104#issuecomment-337318722
- https://github.com/aspnet/Home/issues/1950#issuecomment-312456522

Razor is async!

- Razor rendering is now fully async
- Always use **Html.xxxAsync** or the new <partial /> tag helper (2.1)
- It’s not safe to execute views/partials/view components synchronously
  - https://github.com/aspnet/Mvc/issues/7083

HttpClient

- Use a singleton HttpClient
  - https://aspnetmonsters.com/2016/08/2016-08-27-httpclientwrong/
- Make sure the concurrent connection limit is what you expect
  - ServicePointManager.DefaultConnectionLimit (.NET Framework)
  - HttpClientHandler.MaxConnectionsPerServer (.NET Core)
- Always use async methods
- HttpWebRequest on .NET Core uses HttpClient under the covers (https://source.dot.net/#System.Net.Requests/System/Net/HttpWebRequest.cs,1111)
  - It doesn’t use a singleton HttpClient and HttpClientHandler
  - Porting code from .NET Framework to .NET Core could end up being extremely inefficient
- We recommend not using HttpWebRequest on .NET Core.
- Do **NOT** use the string APIs when dealing with large responses.
- We’re introducing an IHttpClientFactory in 2.1 to help some of these issues

Async all the things

- Use async APIs when available
- Audit 3rd party code (and even code in .NET itself) to make sure things are truly async.
- Asynchrony is viral
  - You can’t really do it in stages.
  - You need to flow it “all the way up”

Diagnostics and Logging

- Always make sure application logs are viewable
- The log level can be changed without restarting the application
- Logs can show up in various places by default
  - The event log
  - The console
  - Files on disk
  - Application insights
- Use 3rd party loggers like Serilog that provide plugins for more logging sinks like Elastic Search (ELK stack etc).
- ASP.NET Core integrates with various Azure services (when running on azure)
  - Application insights
  - App service streaming logs
  - App service tracing (blob or file logger)