.NET Core Networking stack and Performance

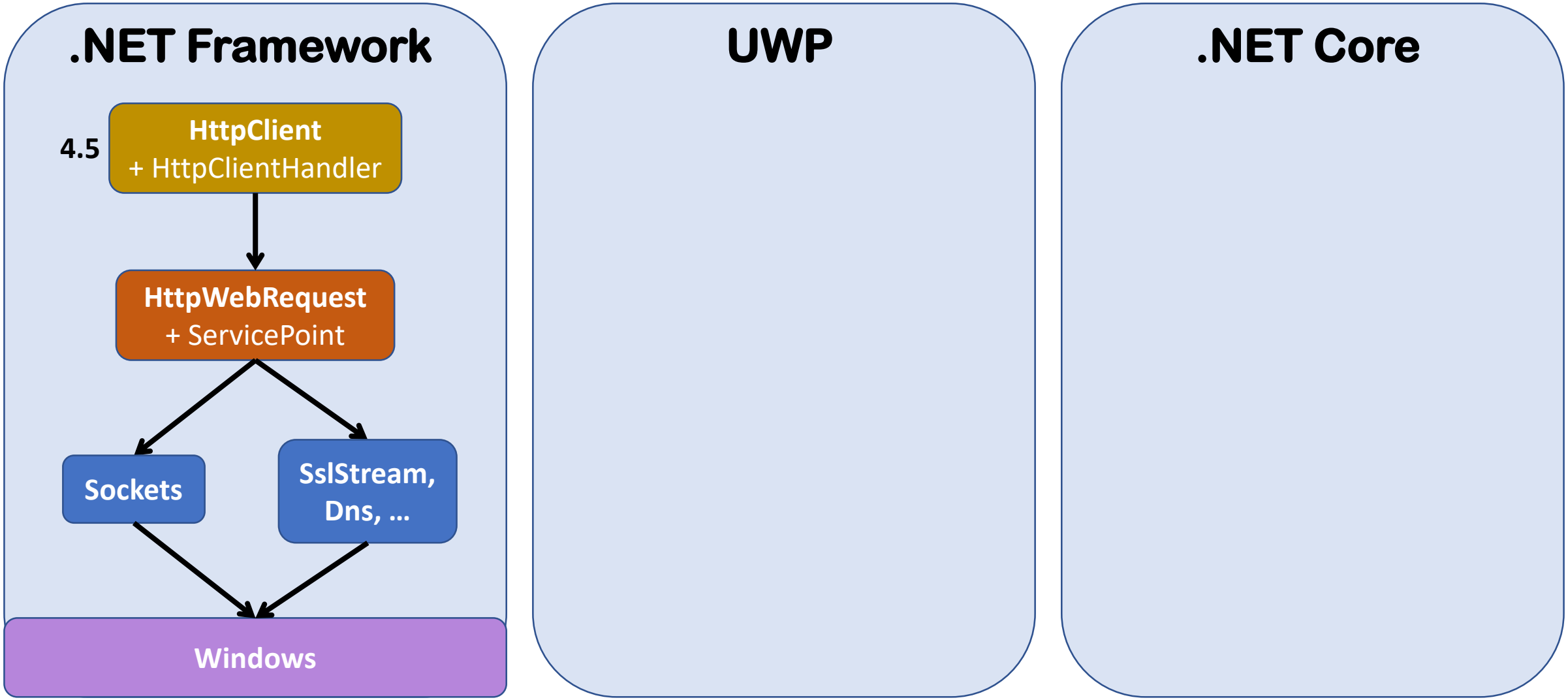
DotNext in Moscow, RU (2017/11/12)

Karel Zikmund ( @ziki_cz)

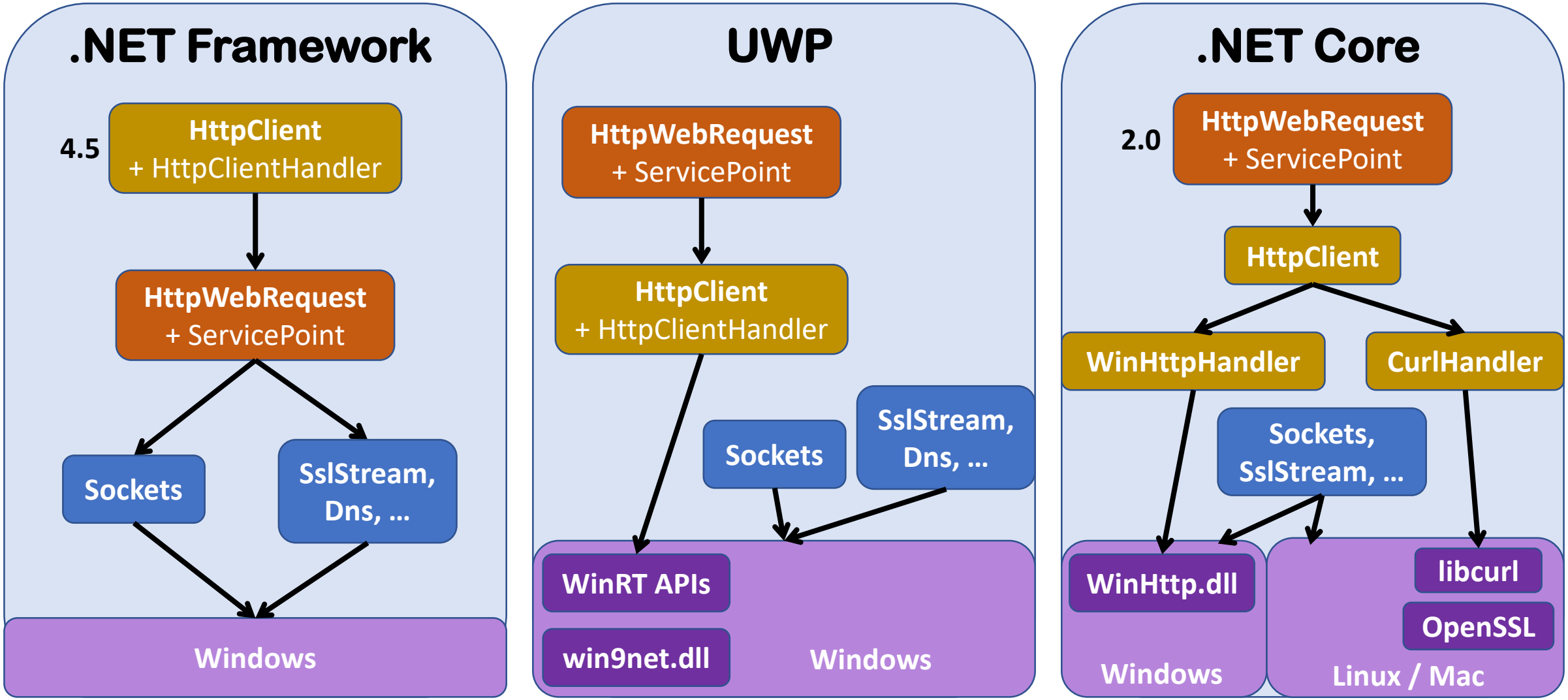
Agenda

- Networking stack architecture evolution
  - .NET Framework, UWP and .NET Core
- Networking stack in .NET Core
  - Direction and plans
  - Status & perf results
- General BCL performance thoughts & observations

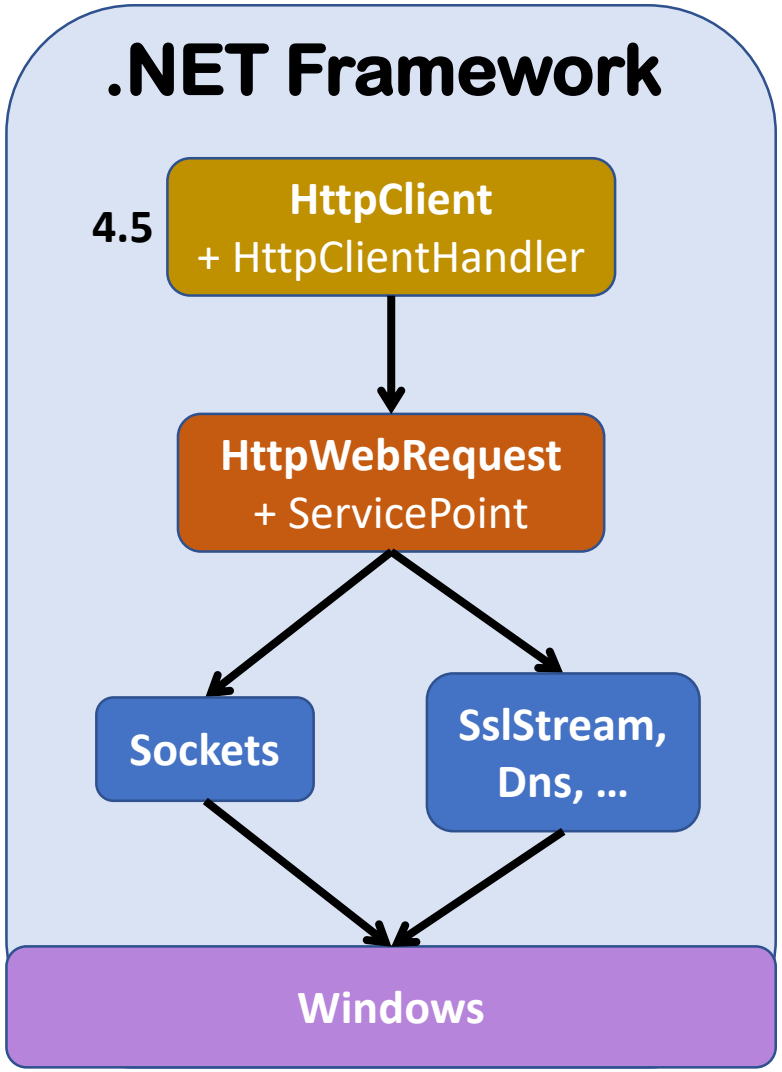
Networking – Architecture Evolution

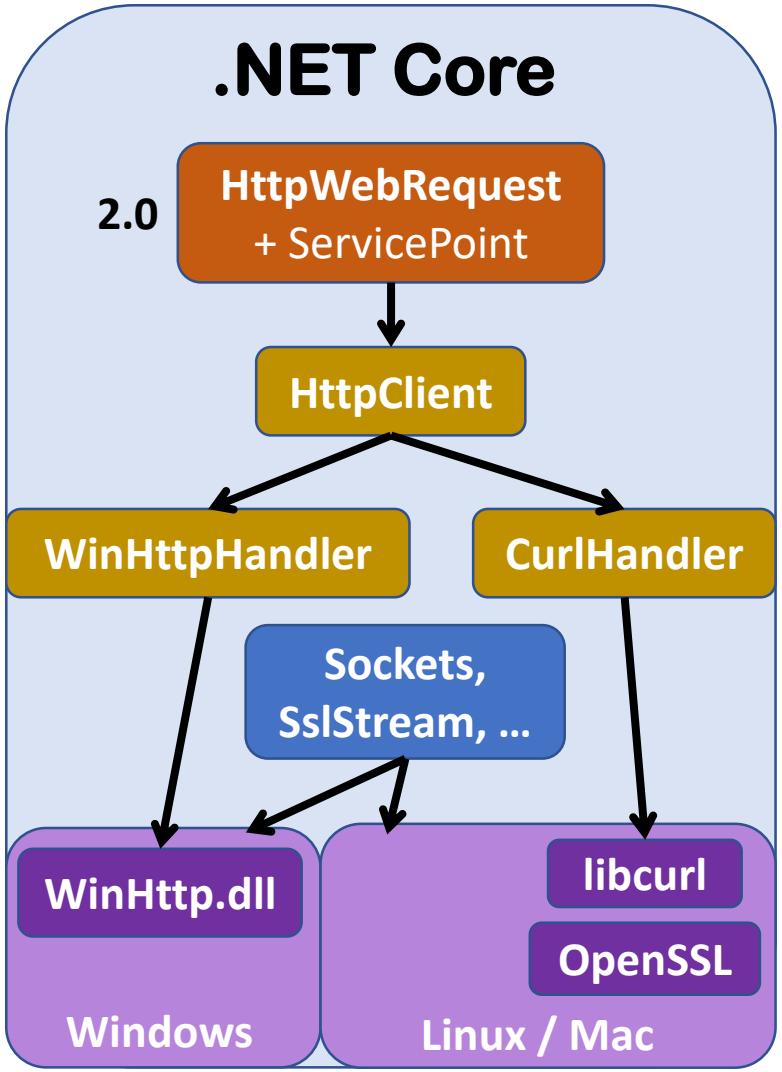


Networking – Architecture Evolution

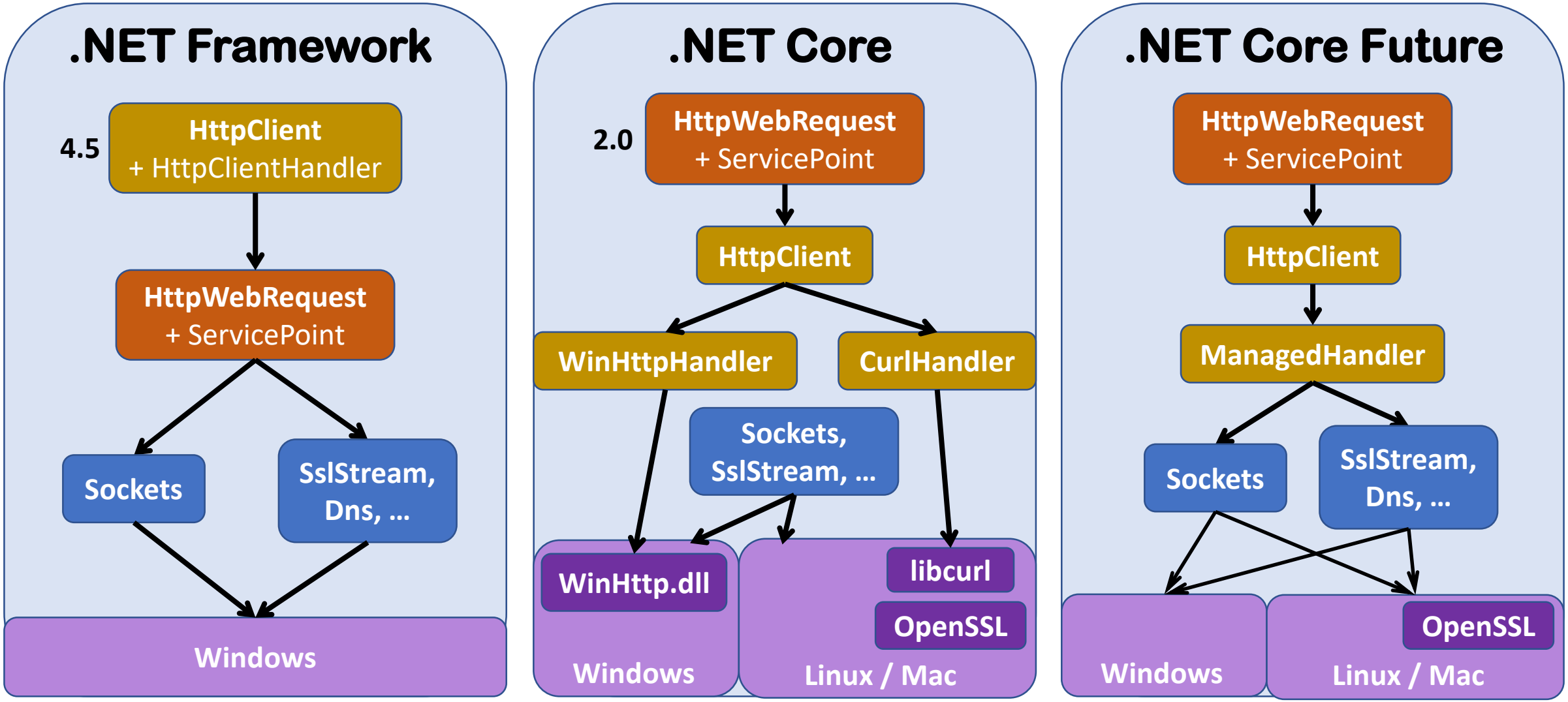


Networking – Architecture Evolution





Networking – Architecture Evolution



Networking – Technical Roadmap

https://github.com/dotnet/designs/issues/9

1. Foundation – rock solid
   - Sockets, SSL, DNS
2. Web stack (client) – high perf & consistency
   - HttpClient, ClientWebSocket
3. Emerging technologies
   - HTTP/2, RIO, QUIC
4. Maintenance components
   - (Http/Ftp/File)WebRequest + ServicePoint, Mail, HttpListener

Networking – Focus on Perf

Scenarios / workloads:

- Micro-benchmarks vs. benchmarks vs. real-world scenarios (feedback)

Metrics:

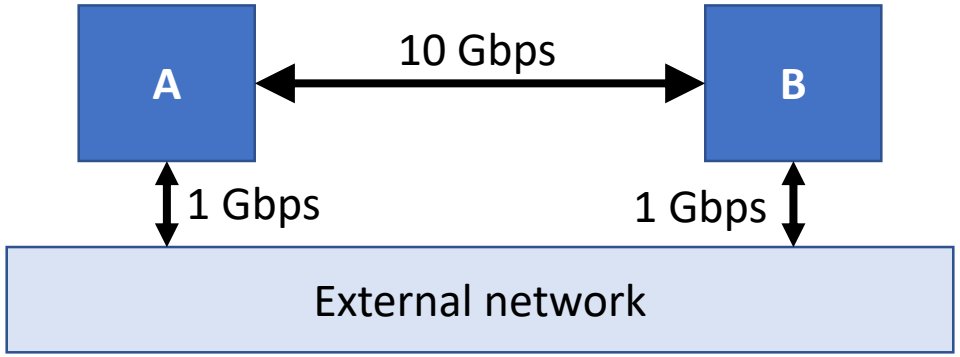
- RPS (Response per second)
- Throughput
- Latency
- Connection density

Important properties:

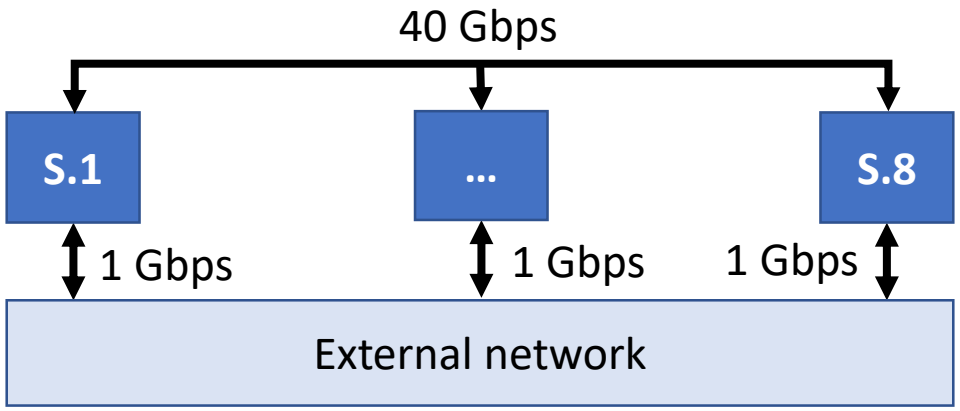
- Percentiles (95% / 99%)
- Scale up
- Resources (CPU) utilization (90%-95% ideal)

Networking – Perf Test Environment

- Repeatability – isolated environment (reduce noise)
- 2 machines:
  - 4-core
  - 16 GB RAM
  - 2x NIC: 1x 1 Gbps + 1x 10 Gbps



- 8 servers:
  - 12-core
  - 64 GB RAM
  - 2x NIC: 1x 1 Gbps + 1x 40 Gbps



Networking – Sockets Perf Results

- Micro-benchmark only (disclaimer: Netty/Go impl may be inefficient)
- Linux 2 CPUs:
  - Gap for small payloads
    (to be investigated)

GB/s	256 B	4 KB	64 KB	1 MB
.NET Core	**0.09**	0.77	1.09	**1.10**
Netty	**0.11**	0.48	0.66	0.67
Go	**0.12**	0.82	1.10	**1.11**

- SSL:
  - 22% gap (to be investigated)

SSL - GB/s	256 B	4 KB	64 KB	1 MB
.NET Core	0.04	0.31	0.71	**0.87**
Netty	0.03	0.12	0.15	0.15
Go	0.06	0.56	0.98	**1.12**

Networking – Sockets Perf on Server

- Kestrel server uses libuv -> Sockets prototypes
- Early Sockets prototype (with hacks):
  - 7% improvement over libuv + more potential
- Recent Sockets prototype (very preliminary data):
  - 13% worse than libuv on Linux
  - 8% worse than libuv on Windows
  - Investigation in progress

Networking – ManagedHandler Perf

- ManagedHandler (functional early stage)
  - Missing large features – authentication, proxy, http2

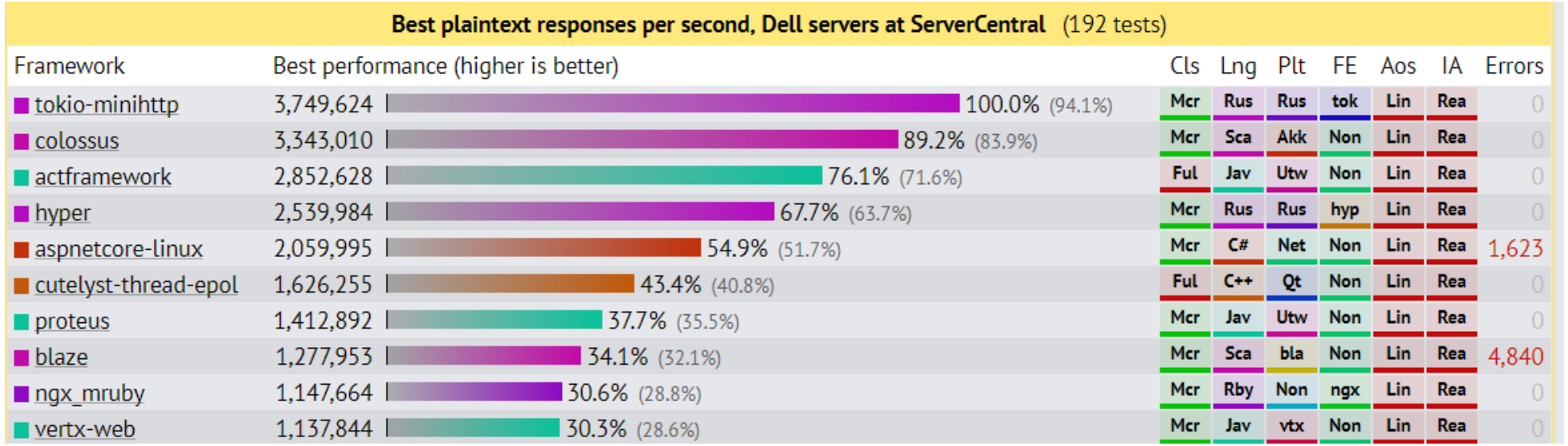
- Early measurements (simple http micro-benchmark):
  - Windows: Parity with Go
  - Linux: 15% gap (pending investigation)

Networking – SSL Perf

- Indicators:
  - Sockets micro-benchmarks (22% overhead)
  - TechEmpower benchmark
  - Customer reports
- Attempt for rewrite by community (@drawaes)
  - Some end-to-end wins, but no micro-benchmark wins
- Next steps:
  - Measure & analyze micro-benchmarks & end-to-end scenarios
  - Improve in .NET Core 2.1

Industry Benchmarks

- TechEmpower benchmark
  - More end-to-end, with DB, etc.
  - Useful for overall platform performance comparison
- Round 15 (preliminary data)
  - ASP.NET Core at #5 entry (jump from #14 in Round 14)



Best plaintext responses per second, Dell servers at ServerCentral (192 tests)

Framework	Best performance (higher is better)	Cls	Lng	Plt	FE	Aos	IA	Errors
tokio-minihttp	3,749,624 100.0% (94.1%)	Mcr	Rus	Rus	tok	Lin	Rea	0
colossus	3,343,010 89.2% (83.9%)	Mcr	Sca	Akk	Non	Lin	Rea	0
actframework	2,852,628 76.1% (71.6%)	Ful	Jav	Utw	Non	Lin	Rea	0
hyper	2,539,984 67.7% (63.7%)	Mcr	Rus	Rus	hyp	Lin	Rea	0
aspnetcore-linux	2,059,995 54.9% (51.7%)	Mcr	C#	Net	Non	Lin	Rea	1,623
cutelyst-thread-epol	1,626,255 43.4% (40.8%)	Ful	C++	Qt	Non	Lin	Rea	0
proteus	1,412,892 37.7% (35.5%)	Mcr	Jav	Utw	Non	Lin	Rea	0
blaze	1,277,953 34.1% (32.1%)	Mcr	Sca	bla	Non	Lin	Rea	4,840
ngx_mruby	1,147,664 30.6% (28.8%)	Mcr	Rby	Non	ngx	Lin	Rea	0
vertx-web	1,137,844 30.3% (28.6%)	Mcr	Jav	vtx	Non	Lin	Rea	0

Industry Benchmarks

Platform performance shows how fast your app could be, one day

… but it is not everything:

- Productivity
- Tooling
- Developer availability (in-house/to hire)
- Documentation
- Community
- etc.

Performance – Lessons Learned

- Plan for performance during design
  - Understand scenario, set goals
  - Prototype and measure early
- Optimize what's important – measure
  - Avoid micro-optimizations
  - Understand the big picture
- Don't guess root cause – measure
  - Minimize repro – it's worth it!

BCL Performance

- Fine-tuned over 15 years
  - Opportunities are often trade-offs (memory vs. speed, etc.)
- How to identify scenarios which matter
  - Customer feedback
  - OSS helps:
    - More eyes on code
    - More reports
    - Motivated contributors

- Perf improvements in .NET Core (.NET blog post by Stephen Toub)
  - Collections, Linq, Compression, Crypto, Math, Serialization, Networking
- Span<T> sprinkled in BCL

BCL Performance – What to not take?

- Specialized collections
  - BCL designed for usability and decent perf for 95% customers
- Code complexity (maintainability) vs. perf wins
- APIs for specialized operations (e.g. to save duplicate lookup)
  - Creates complexity
  - May leak implementation into API surface

Wrap Up

- Proactive investments into .NET Networking stack
  - Consistency across platforms
  - Great performance for all workloads
- Ongoing scenario/feedback-based improvements in BCL perf
- Performance in general is:
  - Important
  - But not the only important thing
  - Tricky to get right in the right place