Proxies for .NET Core using AssemblyLoadContext

Gordon Phoon

Introduction

- Rose-Hulman Institute of Technology
- .NET Core Runtime team (AppModel/Interop/Security)
- Manager: Jeff Schwartz
- Mentor: Sven Boemer

Agenda

- Motivations and goals coming into the internship
- Design process over 12 weeks
- Demo
- MSBuild work
- Performance metrics
- Limitations and Future work

AppDomains in .NET Framework

- Easy in-process solution for isolation
- Create an AppDomain, load an Assembly and create proxy objects easily with *CreateInstanceAndUnwrap*
- *TransparentProxy* looked like whatever type was desired
- Unloadable: Nice for use in a Plugin model

```
public static void Main()
{
    // Create an ordinary instance in the current AppDomain
    Worker localWorker = new Worker();
    localWorker.PrintDomain();

    // Create a new application domain, create an instance
    // of Worker in the application domain, and execute code
    // there.
    AppDomain ad = AppDomain.CreateDomain("New domain");
    Worker remoteWorker = (Worker) ad.CreateInstanceAndUnwrap(
        typeof(Worker).Assembly.FullName,
        "Worker");
    remoteWorker.PrintDomain();
}
```

AppDomains Example

```
#if FEATURE_APPDOMAIN
            if (loadedType.Assembly.AssemblyFile != null)
            {
                taskInstanceInOtherAppDomain = (ITask)taskAppDomain.CreateInstanceFromAndUnwrap(loadedType.Assembly.AssemblyFile, loadedType.Type.FullNa

                // this will force evaluation of the task class type and try to load the task assembly
                Type taskType = taskInstanceInOtherAppDomain.GetType();

                // If the types don't match, we have a problem. It means that our AppDomain was able to load
                // a task assembly using Load, and loaded a different one. I don't see any other choice than
                // to fail here.
                if (taskType != loadedType.Type)
                {
                    logError
                    (
                    taskLocation,
                    taskLine,
                    taskColumn,
                    "ConflictingTaskAssembly",
                    loadedType.Assembly.AssemblyFile,
                    loadedType.Type.GetTypeInfo().Assembly.Location
                    );

                    taskInstanceInOtherAppDomain = null;
                }
            }
            else
            {
                taskInstanceInOtherAppDomain = (ITask)taskAppDomain.CreateInstanceAndUnwrap(loadedType.Type.GetTypeInfo().Assembly.FullName, loadedType.
            }

            return taskInstanceInOtherAppDomain;
```

AppDomains Example

Code | Issues 1,038 | Pull requests 23 | Actions | Projects 4 | Wiki | Security | Insights

MSBuild on .NET Core does not allow tasks with same assembly name, but different assembly versions #3572

New issue

Open **nguerrera** opened this issue on Aug 1, 2018 · 8 comments

nguerrera commented on Aug 1, 2018 • edited ▾ Member

With desktop MSBuild / VS, we've gotten around issues with node reuse and varying versions of the SDK by increasing the assembly version of the SDK tasks on every build. (Aside: we've regressed that a bunch of times with infrastructure changes.)

However, this does not work on .NET Core. I suspect we haven't noticed this until now because node reuse has only made its way to .NET Core recently.

Steps to reproduce

Assignees

No one assigned

Labels

.NET Core

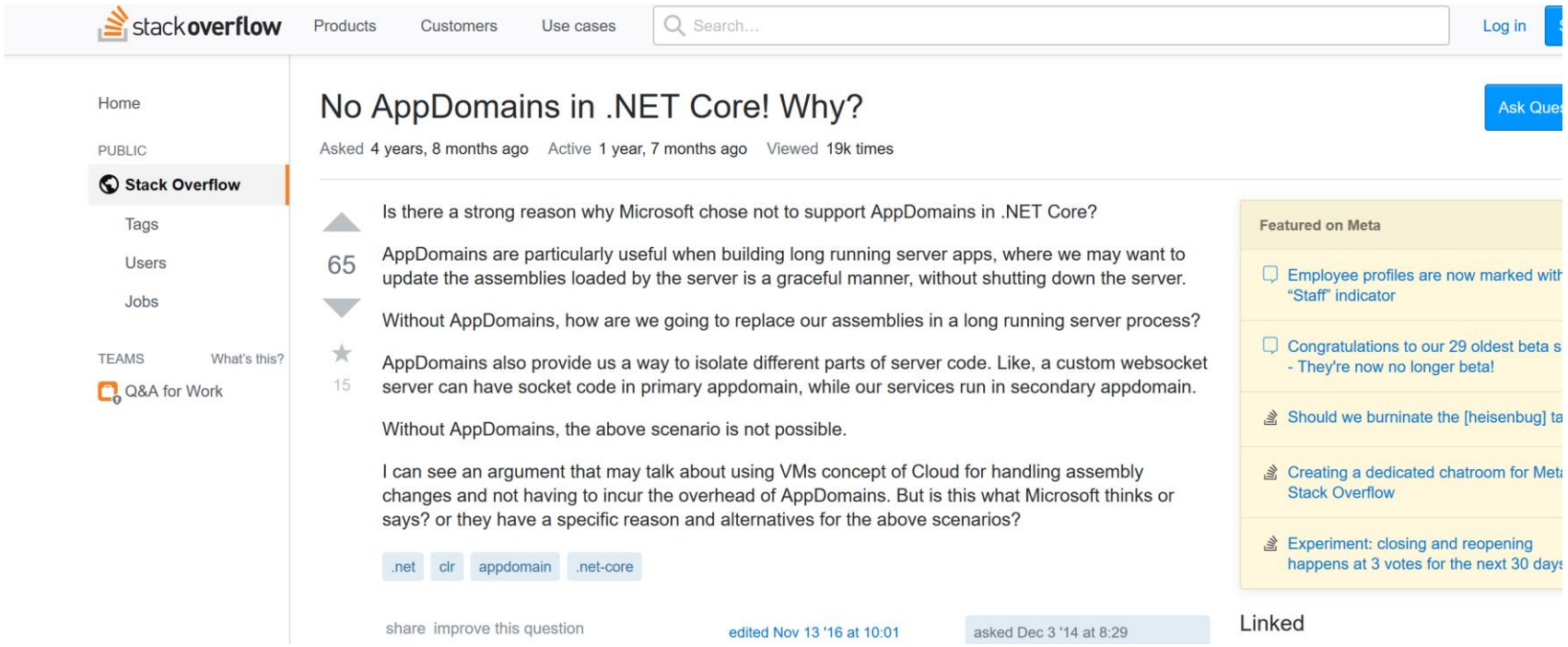
Area: Engine

bug

```
                        taskAppDomain.Load(loadedType.LoadedAssembly.GetName());
                    }

#if FEATURE_APPDOMAIN_UNHANDLED_EXCEPTION
                    // Hook up last minute dumping of any exceptions
                    taskAppDomain.UnhandledException += new UnhandledExceptionEventHandler(ExceptionHandling.UnhandledExceptionHandler);
#endif
                }
            }
            else

#endif
            {
                // perf improvement for the same appdomain case - we already have the type object
                // and don't want to go through reflection to recreate it from the name.
                return (ITask)Activator.CreateInstance(loadedType.Type);
            }

#if FEATURE_APPDOMAIN
```

Motivation

- AppDomains “removed” in .NET Core
  - Technically Default is still there
  - No way to make proxies like this in-proc



Motivation

- AppDomains “removed” in .NET Core
  - Technically Default is still there
  - No way to make proxies like this in-proc
- Want an easy-to-use proxy system that developers from .NET Framework can transition to easily
- For a plugin model, we want some form of unloadability

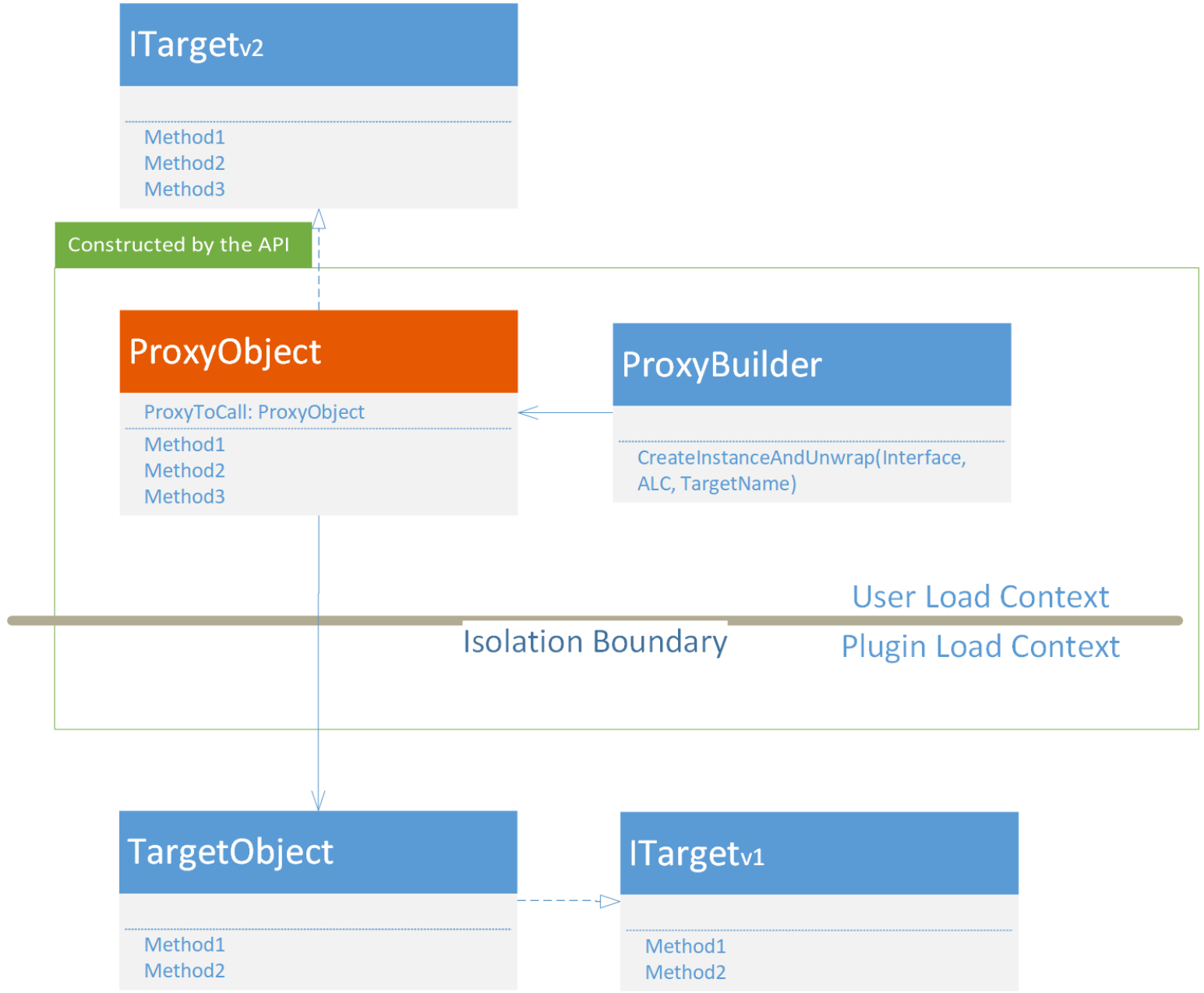
AssemblyLoadContext

- Used for dynamic assembly loading
- Type-Resolution isolation, vs full object isolation
- ALC.Unload()
  - Reliability over consistency
  - Lets the GC clean up after references are removed
- 3.0 unload can be used similarly to AppDomain.Unload()

Goals

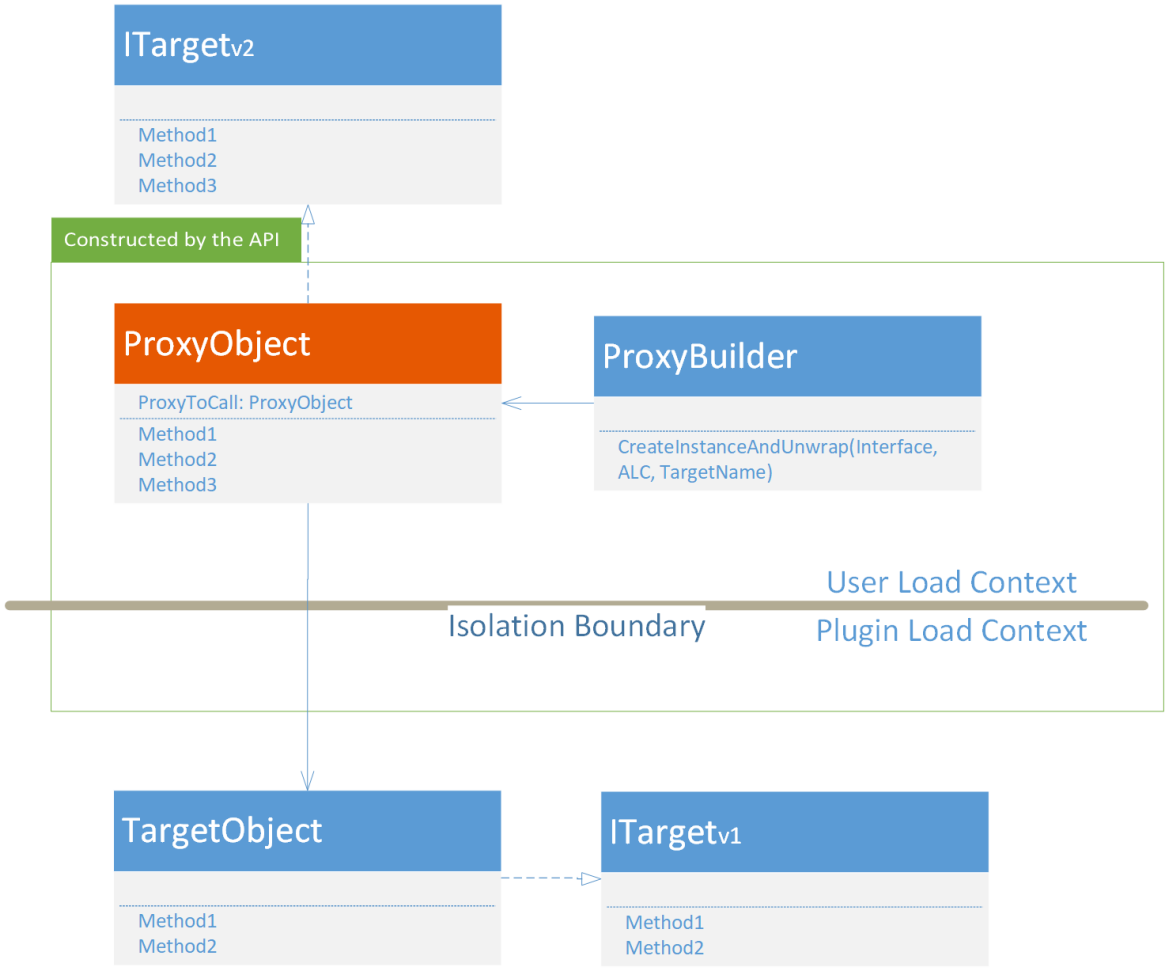
- Proxy System, with similar *CreateInstanceAndUnwrap* function for easy proxy creation
- Supports ALC unloadability
- Performance metrics to compare to old AppDomain work

- Extra Goals:
  - Support different versioning scenarios
  - Try to get an out-of-proc version working


ITargetv2
Method1
Method2
Method3
Constructed by the API
ProxyObject
ProxyToCall: ProxyObject
Method1
Method2
Method3
ProxyBuilder
CreateInstanceAndUnwrap(Interface, ALC, TargetName)
User Load Context
Isolation Boundary
Plugin Load Context
TargetObject
Method1
Method2
ITargetv1
Method1
Method2

Initial Design

- IL Generated ProxyObject
  - Pretends to be our Target by implementing an interface
- Non-shared types to support versioning
  - Load assembly with type and interface into Plugin ALC
- Single point of contact across ALCs



DispatchProxy

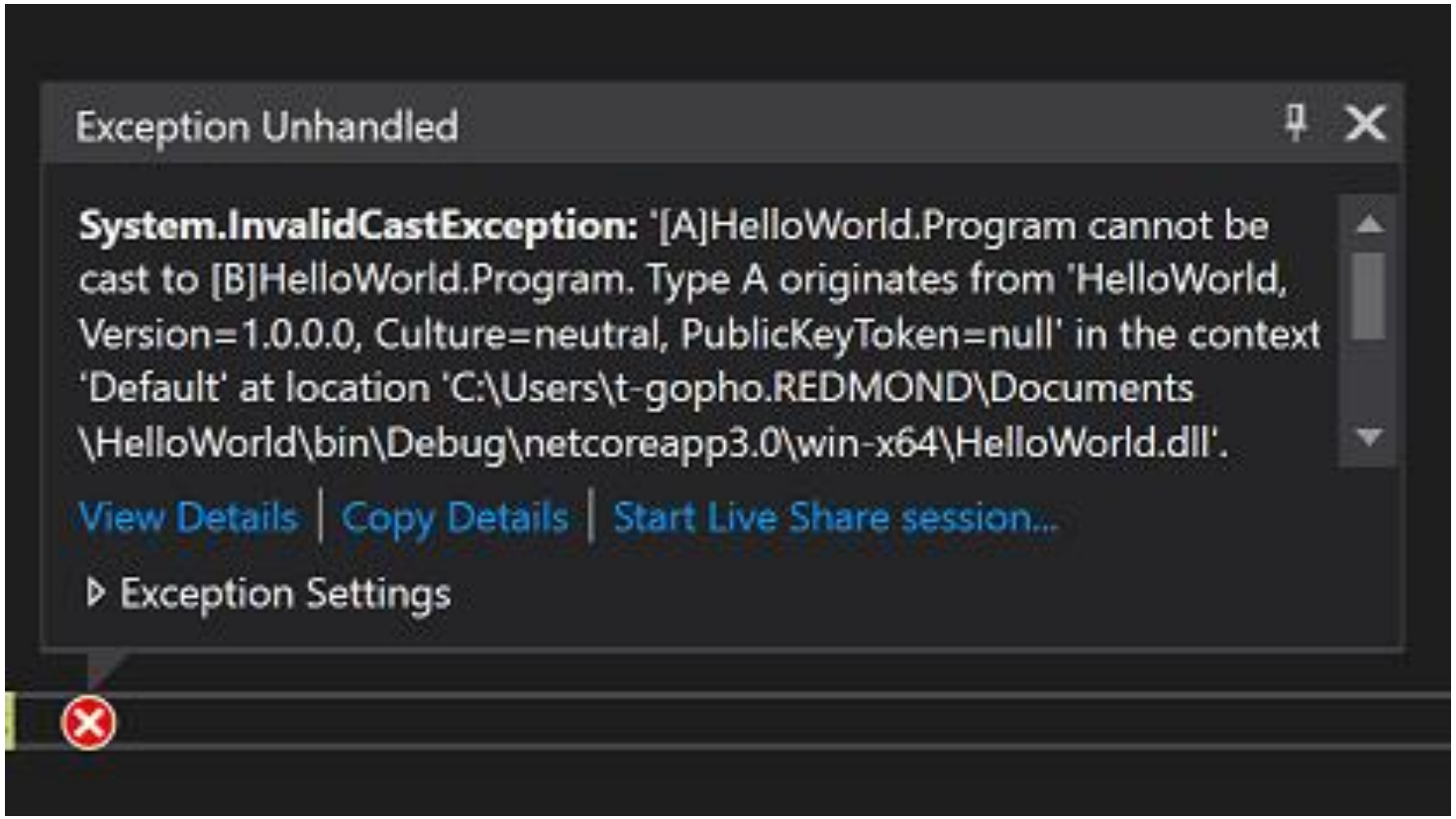
- Does the IL generation for us
- Interface + BaseType -> Proxy which implements the interface
- Acts the same as a normal object, but calls Invoke override instead

- Two Problems
  - It’s created as a dynamic assembly, but non-collectible
  - We’ll get to the other later

Type-Resolution Boundary

- When a type is loaded into 2 ALCs, they are treated like different types at runtime

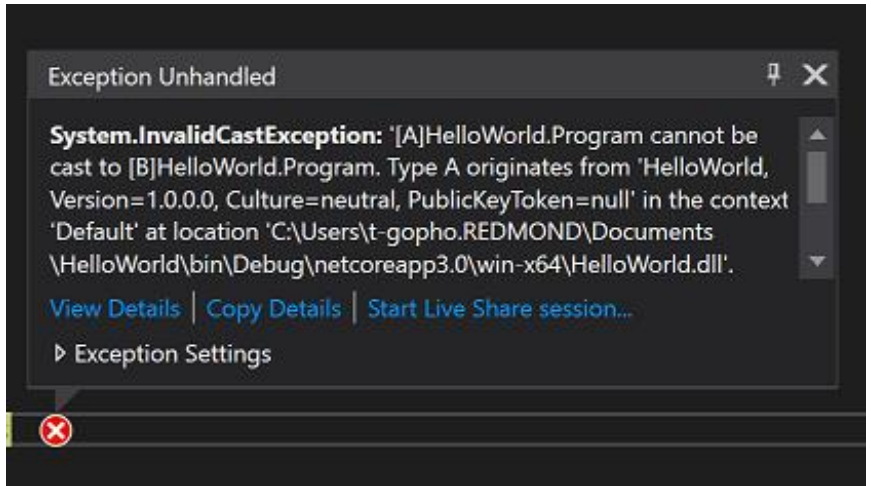
Type-Resolution Boundary



Type-Resolution Boundary

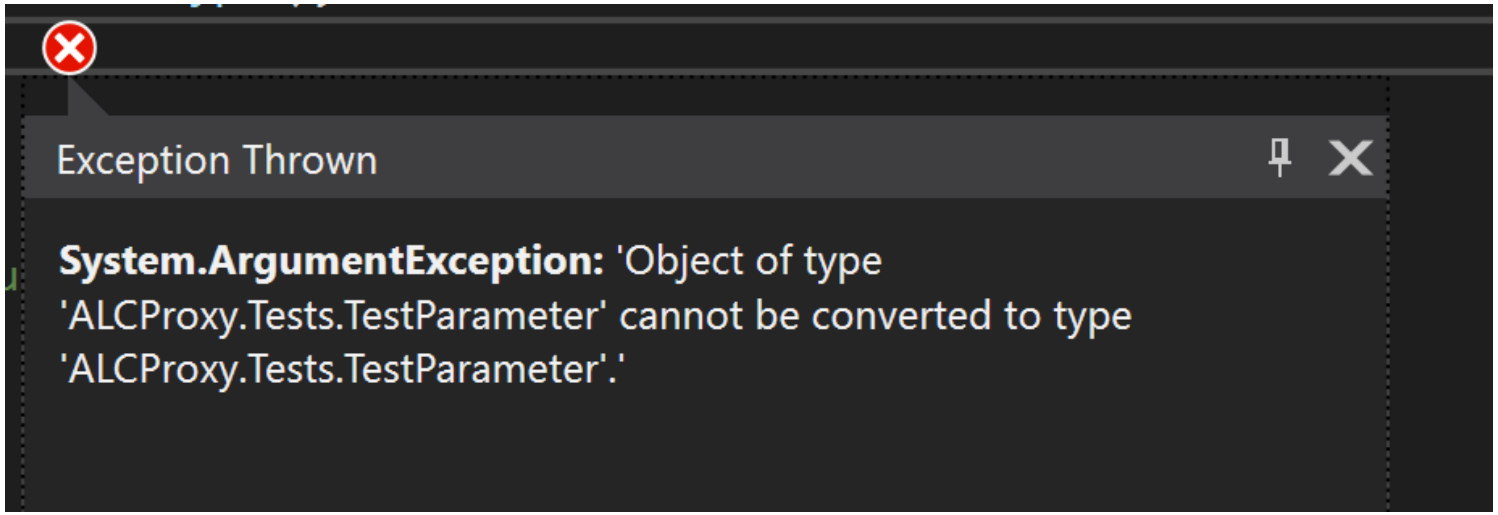
- When a type is loaded into 2 ALCs, they are treated like different types at runtime

- Use Reflection.Invoke instead of casting directly, handled by the DispatchProxy

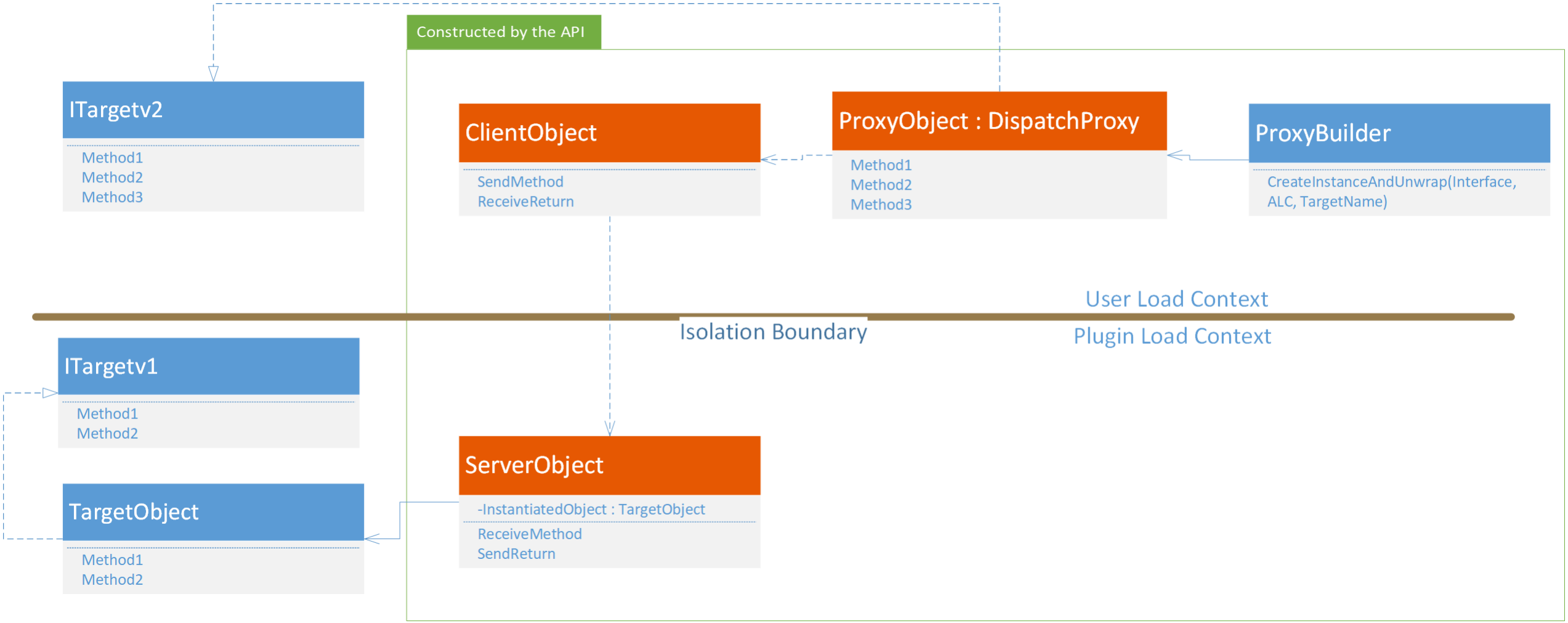


Passing Additional Objects Across the Boundary

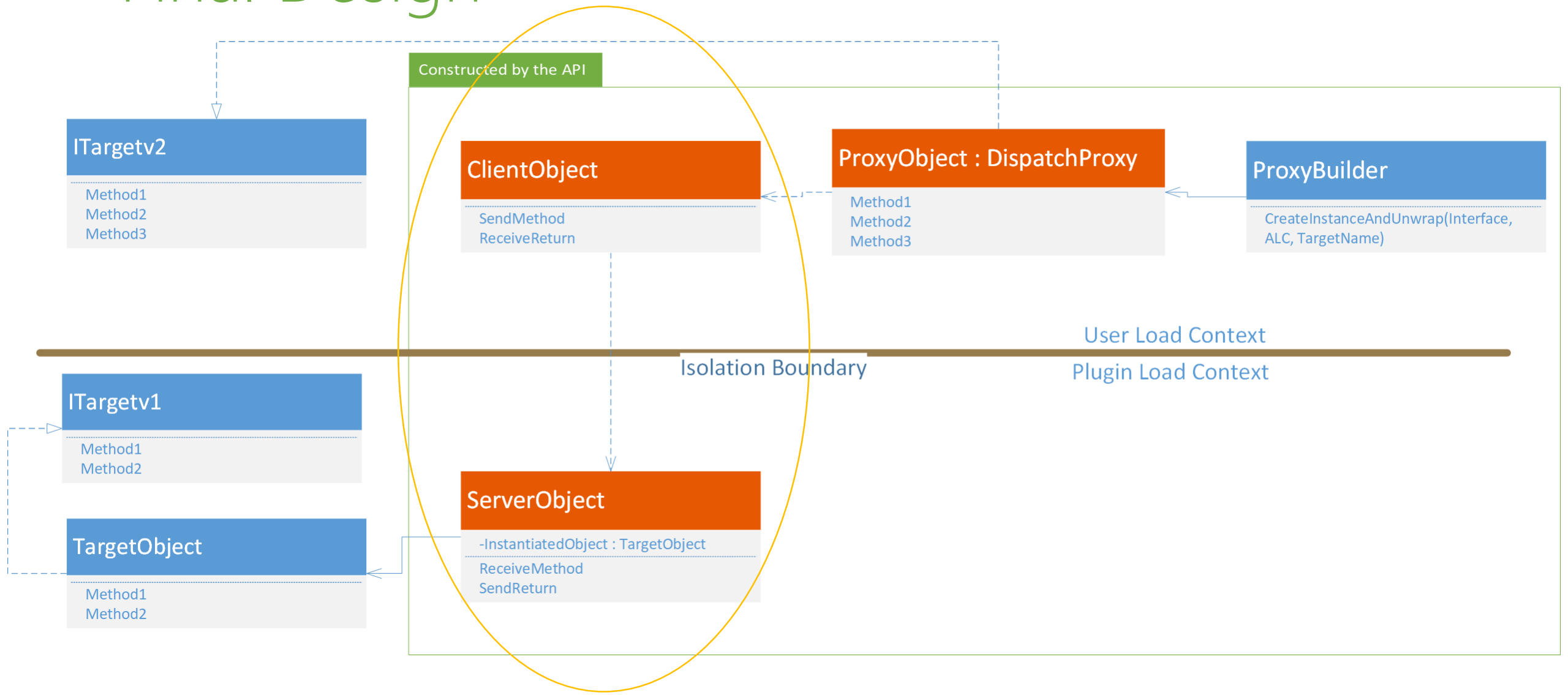
- Still have casting issues for any objects passed to/from the new ALC
- Some structure required to perform passing over
- Two options for solving the problem
  - Serialize all objects being passed between ALCs (call-by-value)
  - Create new proxies for each new object passed (call-by-ref)



Final Design



Final Design



Serialization

- Binary serialization isn’t a good option nowadays
- Use DataContractSerializer, XMLSerializer, or some other option
- Design allows for slotting in a different serializer

Unloadability

- ALC unload sets up for collection by GC
  - Reliable, ensures no objects cleaned up while in use

- More consistent unload behavior desired for plugin model
- Cut off the reference from client to server when ALC.Unload() event fires

- Still reliant on GC to unload fully

Demo

MSBuild – It doesn't work ☹

- ALCProxy code can be added in TaskLoader.cs pretty easily
- However, some types can't be serialized, so it crashes during the attempt
- Shared types may be a solution

```
            return taskInstanceInOtherAppDomain;
#elif NETCOREAPP3_0
            if (loadedType.Assembly.AssemblyFile != null)
            {
                taskInstanceInOtherContext = ProxyBuilder<ITask, ClientDispatch>.CreateInstanceAndUnwrap(
                    taskLoadContext, AssemblyName.GetAssemblyName(loadedType.Assembly.AssemblyLocation), loadedType.Type.FullName);
                //taskInstanceInOtherAppDomain = (ITask)taskAppDomain.CreateInstanceFromAndUnwrap(
                    //loadedType.Assembly.AssemblyFile, loadedType.Type.FullName);
                // this will force evaluation of the task class type and try to load the task assembly
                Type taskType = taskInstanceInOtherContext.GetType();


            }
            else
            {
                taskInstanceInOtherContext = ProxyBuilder<ITask, ClientDispatch>.CreateInstanceAndUnwrap(
                    taskLoadContext, AssemblyName.GetAssemblyName(loadedType.Type.GetTypeInfo().Assembly.Location), loadedType.Type.FullName);
                //taskInstanceInOtherContext = (ITask)taskAppDomain.CreateInstanceAndUnwrap(
                    //loadedType.Type.GetTypeInfo().Assembly.FullName, loadedType.Type.FullName);
            }
            Console.WriteLine(PrintMessage("Load context: " + loadedType.Type.Name));
            return taskInstanceInOtherContext;
#endif
```

Performance

```
BenchmarkDotNet=v0.11.3, OS=Windows 10.0.18950
Intel Core i7-8650U CPU 1.90GHz (Kaby Lake R), 1 CPU, 8 logical and 4 physical cores
.NET Core SDK=3.0.100-preview8-013437
  [Host]     : .NET Core 3.0.0-preview8-28373-17 (CoreCLR 4.700.19.37204, CoreFX 4.700.19.37208), 64bit RyuJIT
  DefaultJob : .NET Core 3.0.0-preview8-28373-17 (CoreCLR 4.700.19.37204, CoreFX 4.700.19.37208), 64bit RyuJIT
```

Method	Mean	Error	StdDev	Median
CreateProxyObject	17,932,139.635 ns	377,708.9706 ns	449,635.6611 ns	17,842,922.969 ns
CreateExternalAssemblyProxyObject	18,353,773.737 ns	381,520.6028 ns	1,112,913.5061 ns	18,045,958.750 ns
CreateControlObject	1.576 ns	0.1245 ns	0.2760 ns	1.473 ns
CallSimpleMethodThroughProxy	3,641.442 ns	162.7061 ns	293.3927 ns	3,493.676 ns
CallSimpleMethodControl	3.188 ns	0.3236 ns	0.3463 ns	3.056 ns
CreateGenericProxy	17,442,147.272 ns	302,162.3193 ns	252,319.2216 ns	17,395,209.531 ns
CreateGenericControl	3.441 ns	0.1384 ns	0.1294 ns	3.368 ns
CallSimpleMethodGeneric	5,222.213 ns	98.1293 ns	81.9424 ns	5,184.011 ns
CallSimpleMethodGenericControl	706.292 ns	6.4399 ns	5.0278 ns	706.868 ns
UserTypeParameters	19,068.864 ns	186.4137 ns	165.2509 ns	19,031.007 ns
UserTypeParametersControl	6.439 ns	0.1279 ns	0.1068 ns	6.458 ns
UserTypeParameters2	19,105.781 ns	451.3815 ns	400.1379 ns	19,053.287 ns
UserTypeParametersControl2	3.327 ns	0.0668 ns	0.0593 ns	3.317 ns

```
BenchmarkDotNet=v0.11.5, OS=Windows 10.0.18950
Intel Core i7-8650U CPU 1.90GHz (Kaby Lake R), 1 CPU, 8 logical and 4 physical cores
  [Host]     : .NET Framework 4.7.2 (CLR 4.0.30319.42000), 32bit LegacyJIT-v4.8.3921.0
  DefaultJob : .NET Framework 4.7.2 (CLR 4.0.30319.42000), 32bit LegacyJIT-v4.8.3921.0
```

Method	Mean	Error	StdDev	Median
CreateProxyObject	633,775.047 ns	57,139.2880 ns	165,771.4352 ns	658,227.142 ns
CreateControlObject	11.598 ns	1.3510 ns	3.8326 ns	10.475 ns
CallSimpleMethodThroughProxy	7,771.980 ns	628.1640 ns	1,802.3202 ns	6,955.483 ns
CallSimpleMethodControl	4.128 ns	1.5589 ns	4.3968 ns	1.955 ns
CreateGenericProxy	615,069.780 ns	116,621.4572 ns	338,339.9937 ns	579,898.792 ns
CreateGenericControl	19.068 ns	1.2528 ns	3.4506 ns	18.333 ns
CallSimpleMethodGeneric	25,551.128 ns	1,307.3666 ns	3,644.4156 ns	24,720.659 ns
CallSimpleMethodGenericControl	8,118.753 ns	676.1748 ns	1,950.9188 ns	7,602.008 ns
UserTypeParameters	720,413.916 ns	54,520.3760 ns	158,173.4966 ns	738,401.312 ns
UserTypeParametersControl	28.522 ns	0.6977 ns	0.9551 ns	28.450 ns
UserTypeParameters2	21,728.934 ns	432.5692 ns	967.5030 ns	21,646.248 ns
UserTypeParametersControl2	16.342 ns	1.4584 ns	4.1135 ns	15.511 ns

Performance

- Measured using Benchmark.NET
  - Running on local machine
- vs normal object creation:
  - ~10,000,000x slower on object creation
  - 10x-1000x slower when running methods
- vs AppDomain proxies:
  - 100x-1000x slower on object creation
  - Around the same performance times for running methods

Performance Bottlenecks

- Serialization/deserialization
  - More parameters = worse performance
  - Tested with DataContractSerializer
  - Alternatives may be faster
- Reflection.Invoke
- Assembly loading during object creation
  - May have to do with how we're loading assemblies, but need to check that

Limitations and Learnings

- Everything is limited to call-by-value
- Serialization and reflection are both slow
- Non-shared types means while we do have more isolation, we also spend more time doing Assembly loading
- Unloading isn’t instant, even though it seems like it
- Serializing big projects is hard.

Future Expansion

- Out-of-proc proxies with gRPC
- Alternative serialization options
- Call-by-ref
- Merging all proxies into 1 client that holds all the ALCs and objects that we're sending though

Conclusion

- Built an API that can create proxies between AssemblyLoadContexts
  - Unloadable contexts
  - Versioning supported
  - Good for small projects, worse for larger ones
- Tested performance of new proxy system
  - Perf is similar to AppDomains, except for creating the proxies themselves
  - Limited by serialization, invoke, and assembly loading