Improper Instantiation antipattern

06/05/2017 • 6 minutes to read • Contributors all

In this article

It can hurt performance to continually create new instances of an object that is meant to be created once and then shared.

Problem description

Many libraries provide abstractions of external resources. Internally, these classes typically manage their own connections to the resource, acting as brokers that clients can use to access the resource. Here are some examples of broker classes that are relevant to Azure applications:

- `System.Net.Http.HttpClient`. Communicates with a web service using HTTP.
- `Microsoft.ServiceBus.Messaging.QueueClient`. Posts and receives messages to a Service Bus queue.
- `Microsoft.Azure.Documents.Client.DocumentClient`. Connects to a Cosmos DB instance
- `StackExchange.Redis.ConnectionMultiplexer`. Connects to Redis, including Azure Redis Cache.

These classes are intended to be instantiated once and reused throughout the lifetime of an application. However, it's a common misunderstanding that these classes should be acquired only as necessary and released quickly. (The ones listed here happen to be .NET libraries, but the pattern is not unique to .NET.) The following ASP.NET example creates an instance of `HttpClient` to communicate with a remote service. You can find the complete sample here.

C# Copy

```csharp
public class NewHttpClientInstancePerRequestController : ApiController
{
    // This method creates a new instance of HttpClient and disposes it for every call to Get-
ProductAsync.
    public async Task<Product> GetProductAsync(string id)
    {
        using (var httpClient = new HttpClient())
        {
            var hostName = HttpContext.Current.Request.Url.Host;
            var result = await
httpClient.GetStringAsync(string.Format("http://{0}:8080/api/...", hostName));
            return new Product { Name = result };
        }
    }
}
```

In a web application, this technique is not scalable. A new `HttpClient` object is created for each user request. Under heavy load, the web server may exhaust the number of available sockets, resulting in `SocketException` errors.

This problem is not restricted to the `HttpClient` class. Other classes that wrap resources or are expensive to create might cause similar issues. The following example creates an instance of the `ExpensiveToCreateService` class. Here the issue is not necessarily socket exhaustion, but simply how long it takes to create each instance. Continually creating and destroying instances of this class might adversely affect the scalability of the system.

C#

Copy

```csharp
public class NewServiceInstancePerRequestController : ApiController
{
    public async Task<Product> GetProductAsync(string id)
    {
        var expensiveToCreateService = new ExpensiveToCreateService();
        return await expensiveToCreateService.GetProductByIdAsync(id);
    }
}

public class ExpensiveToCreateService
{
    public ExpensiveToCreateService()
    {
        // Simulate delay due to setup and configuration of ExpensiveToCreateService
        Thread.SpinWait(Int32.MaxValue / 100);
    }
    ...
}
```

How to fix the problem

If the class that wraps the external resource is shareable and thread-safe, create a shared singleton instance or a pool of reusable instances of the class.

The following example uses a static `HttpClient` instance, thus sharing the connection across all requests.

C#

Copy

```csharp
public class SingleHttpClientInstanceController : ApiController
{
    private static readonly HttpClient httpClient;

    static SingleHttpClientInstanceController()
    {
        httpClient = new HttpClient();
    }

    // This method uses the shared instance of HttpClient for every call to GetProductAsync.
    public async Task<Product> GetProductAsync(string id)
    {
        var hostName = HttpContext.Current.Request.Url.Host;
        var result = await httpClient.GetStringAsync(string.Format("http://{0}:8080/api/...",
hostName));
        return new Product { Name = result };
    }
}
```

Considerations

- The key element of this antipattern is repeatedly creating and destroying instances of a *shareable* object. If a class is not shareable (not thread-safe), then this antipattern does not apply.
- The type of shared resource might dictate whether you should use a singleton or create a pool. The `HttpClient` class is designed to be shared rather than pooled. Other objects might support pooling, enabling the system to spread the workload across multiple instances.

- Objects that you share across multiple requests *must* be thread-safe. The `HttpClient` class is designed to be used in this manner, but other classes might not support concurrent requests, so check the available documentation.

- Be careful about setting properties on shared objects, as this can lead to race conditions. For example, setting `DefaultRequestHeaders` on the `HttpClient` class before each request can create a race condition. Set such properties once (for example, during startup), and create separate instances if you need to configure different settings.

- Some resource types are scarce and should not be held onto. Database connections are an example. Holding an open database connection that is not required may prevent other concurrent users from gaining access to the database.

- In the .NET Framework, many objects that establish connections to external resources are created by using static factory methods of other classes that manage these connections. These objects are intended to be saved and reused, rather than disposed and re-created. For example, in Azure Service Bus, the `QueueClient` object is created through a `MessagingFactory` object. Internally, the `MessagingFactory` manages connections. For more information, see Best Practices for performance improvements using Service Bus Messaging.

How to detect the problem

Symptoms of this problem include a drop in throughput or an increased error rate, along with one or more of the following:

- An increase in exceptions that indicate exhaustion of resources such as sockets, database connections, file handles, and so on.
- Increased memory use and garbage collection.
- An increase in network, disk, or database activity.

You can perform the following steps to help identify this problem:

1. Performing process monitoring of the production system, to identify points when response times slow down or the system fails due to lack of resources.
2. Examine the telemetry data captured at these points to determine which operations might be creating and destroying resource-consuming objects.
3. Load test each suspected operation, in a controlled test environment rather than the production system.
4. Review the source code and examine the how broker objects are managed.

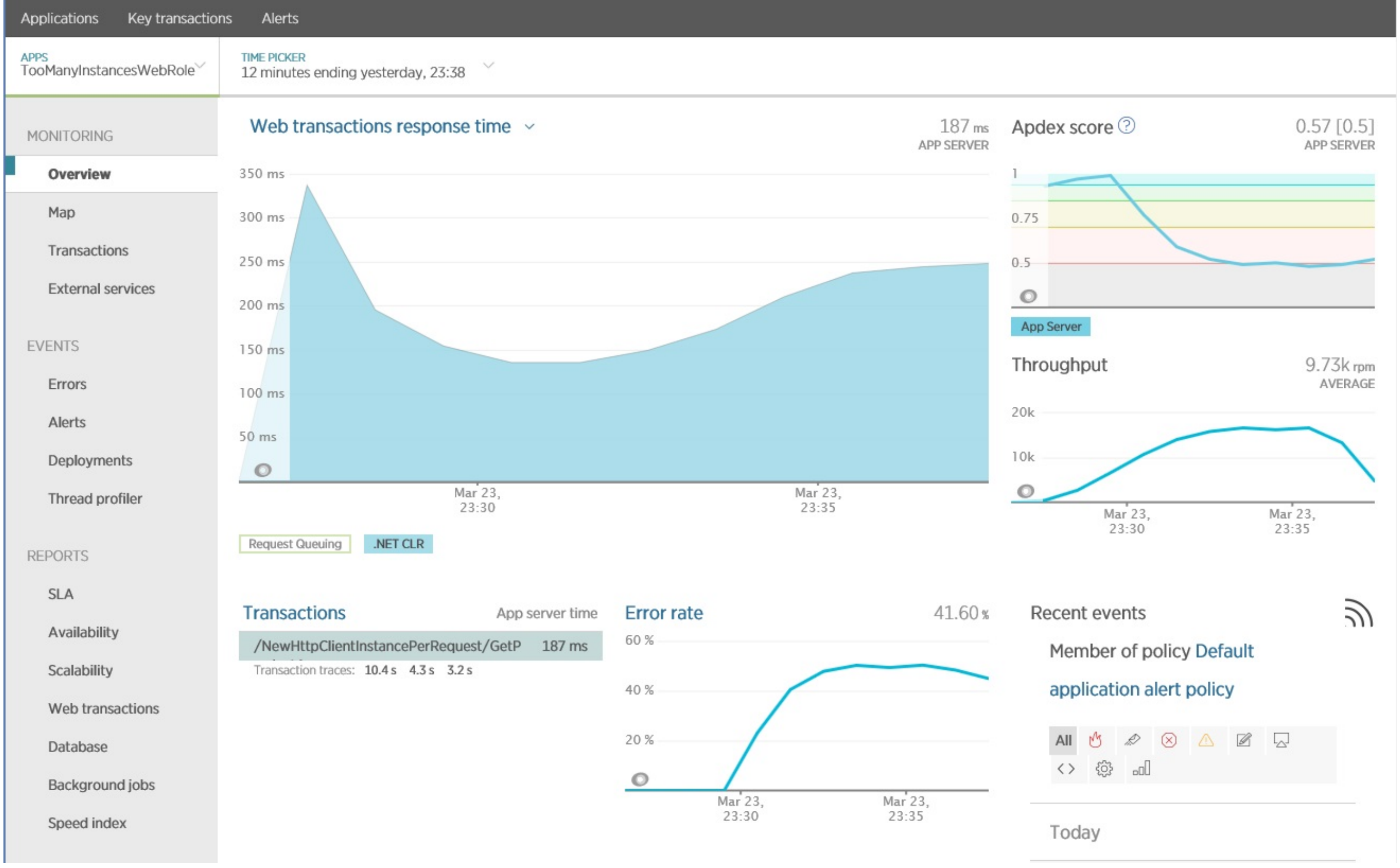
Look at stack traces for operations that are slow-running or that generate exceptions when the system is under load. This information can help to identify how these operations are using resources. Exceptions can help to determine whether errors are caused by shared resources being exhausted.

Example diagnosis

The following sections apply these steps to the sample application described earlier.

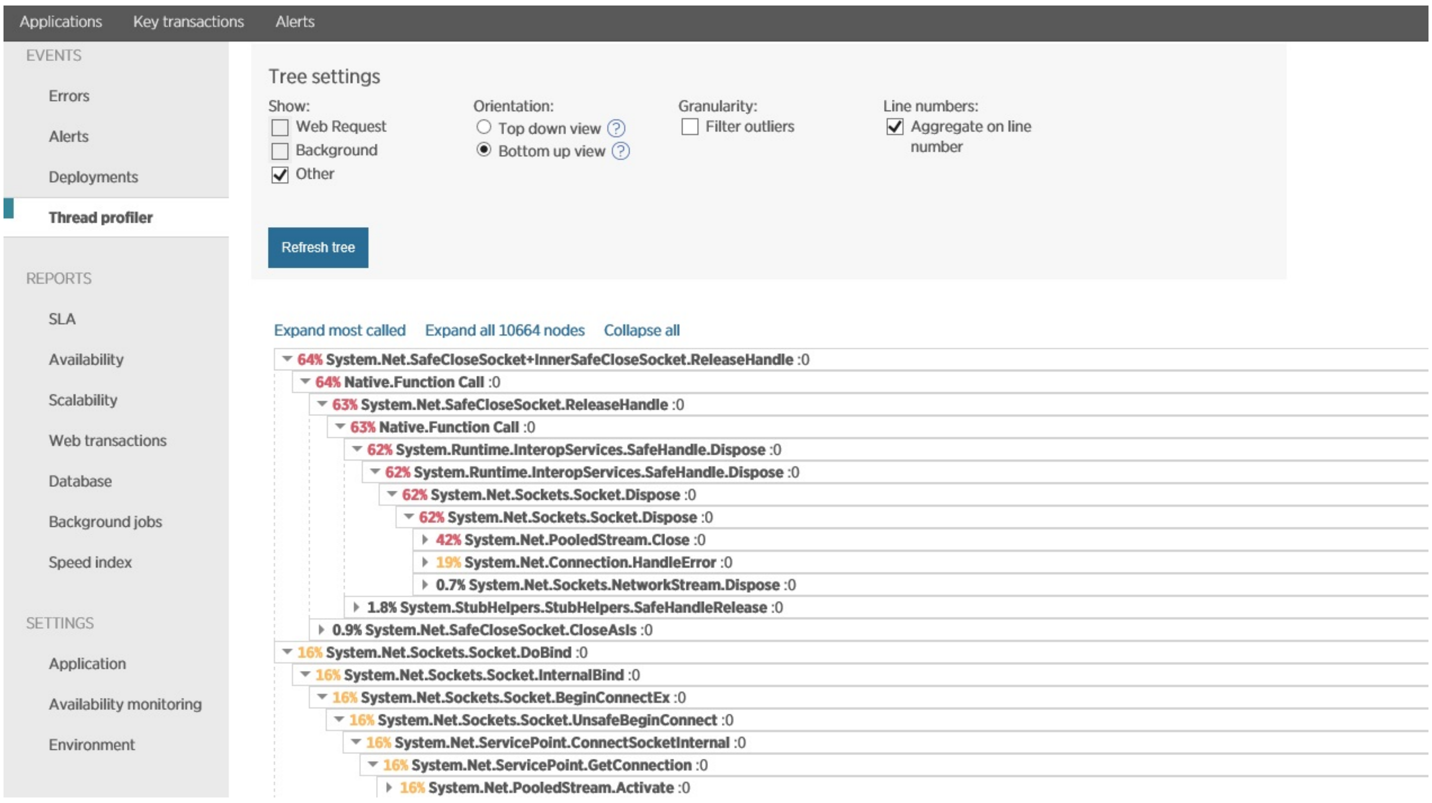
Identify points of slowdown or failure

The following image shows results generated using New Relic APM, showing operations that have a poor response time. In this case, the `GetProductAsync` method in the `NewHttpClientInstancePerRequest` controller is worth investigating further. Notice that the error rate also increases when these operations are running.



Examine telemetry data and find correlations

The next image shows data captured using thread profiling, over the same period corresponding as the previous image. The system spends a significant time opening socket connections, and even more time closing them and handling socket exceptions.



Tree settings

Show: ☐ Web Request ☐ Background ☑ Other

Orientation: ○ Top down view [✓ Bottom up view]

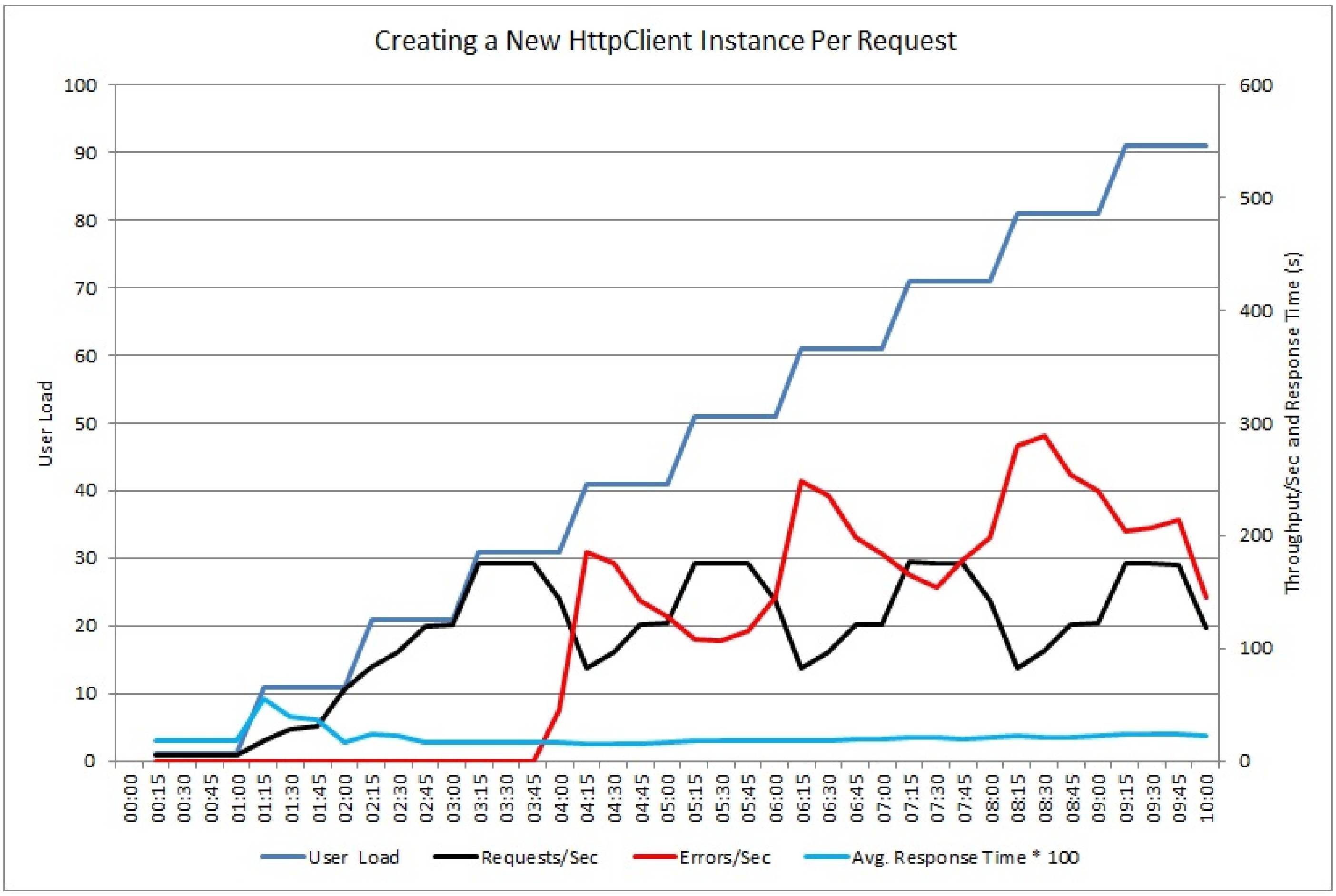
Granularity: ☐ Filter outliers

Line numbers: ☑ Aggregate on line number

```
▾ 64% System.Net.SafeCloseSocket+InnerSafeCloseSocket.ReleaseHandle :0
  ▾ 64% Native.Function Call :0
    ▾ 63% System.Net.SafeCloseSocket.ReleaseHandle :0
      ▾ 63% Native.Function Call :0
        ▾ 62% System.Runtime.InteropServices.SafeHandle.Dispose :0
          ▾ 62% System.Runtime.InteropServices.SafeHandle.Dispose :0
            ▾ 62% System.Net.Sockets.Socket.Dispose :0
              ▾ 62% System.Net.Sockets.Socket.Dispose :0
                ▸ 42% System.Net.PooledStream.Close :0
                ▸ 19% System.Net.Connection.HandleError :0
                ▸ 0.7% System.Net.Sockets.NetworkStream.Dispose :0
        ▸ 1.8% System.StubHelpers.StubHelpers.SafeHandleRelease :0
    ▸ 0.9% System.Net.SafeCloseSocket.CloseAsIs :0
▾ 16% System.Net.Sockets.Socket.DoBind :0
  ▾ 16% System.Net.Sockets.Socket.InternalBind :0
    ▾ 16% System.Net.Sockets.Socket.BeginConnectEx :0
      ▾ 16% System.Net.Sockets.Socket.UnsafeBeginConnect :0
        ▾ 16% System.Net.ServicePoint.ConnectSocketInternal :0
          ▾ 16% System.Net.ServicePoint.GetConnection :0
            ▸ 16% System.Net.PooledStream.Activate :0
```

Performing load testing

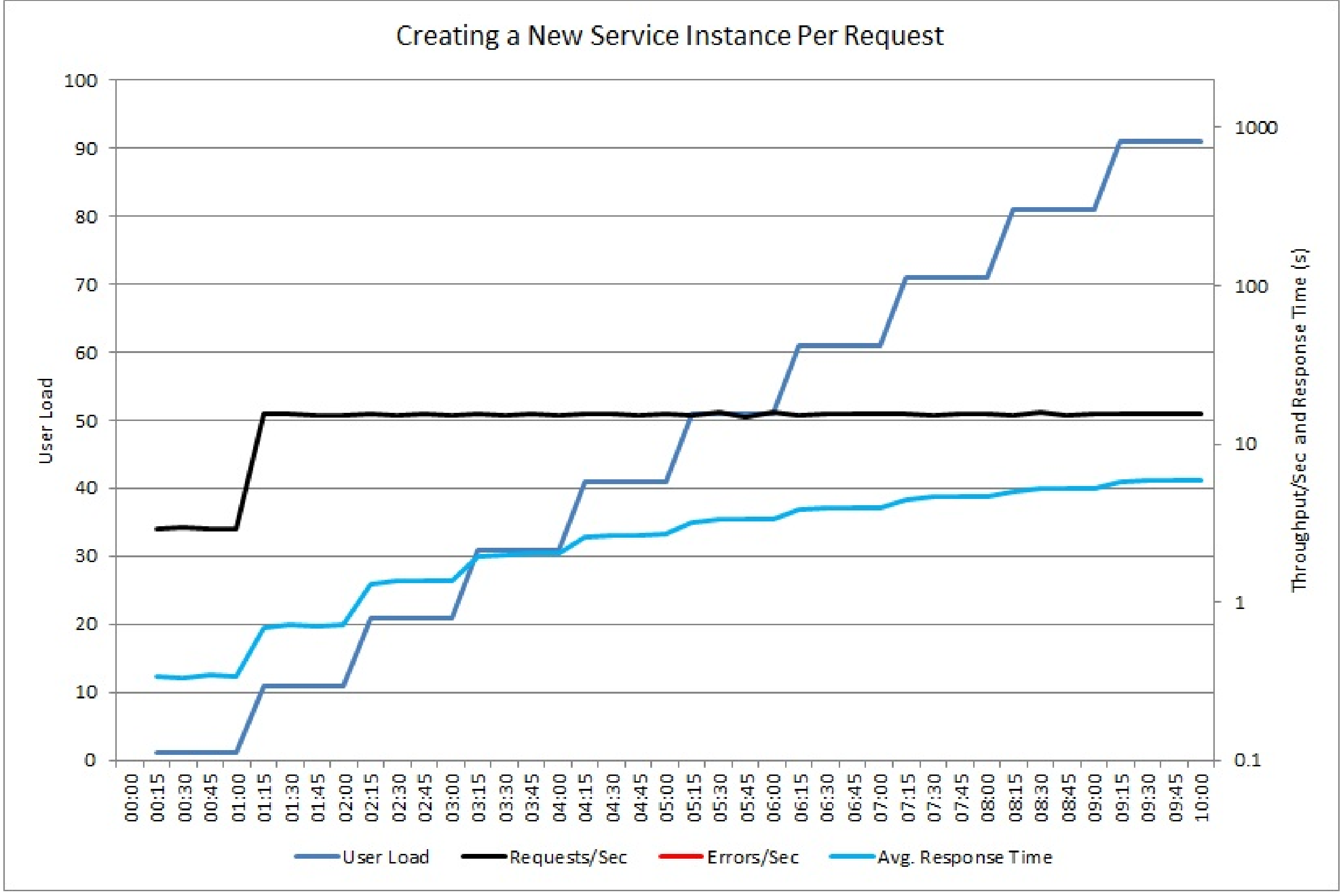
Use load testing to simulate the typical operations that users might perform. This can help to identify which parts of a system suffer from resource exhaustion under varying loads. Perform these tests in a controlled environment rather than the production system. The following graph shows the throughput of requests handled by the `NewHttpClientInstancePerRequest` controller as the user load increases to 100 concurrent users.



At first, the volume of requests handled per second increases as the workload increases. At about 30 users, however, the volume of successful requests reaches a limit, and the system starts to generate exceptions. From then on, the volume of exceptions gradually increases with the user load.

The load test reported these failures as HTTP 500 (Internal Server) errors. Reviewing the telemetry showed that these errors were caused by the system running out of socket resources, as more and more `HttpClient` objects were created.

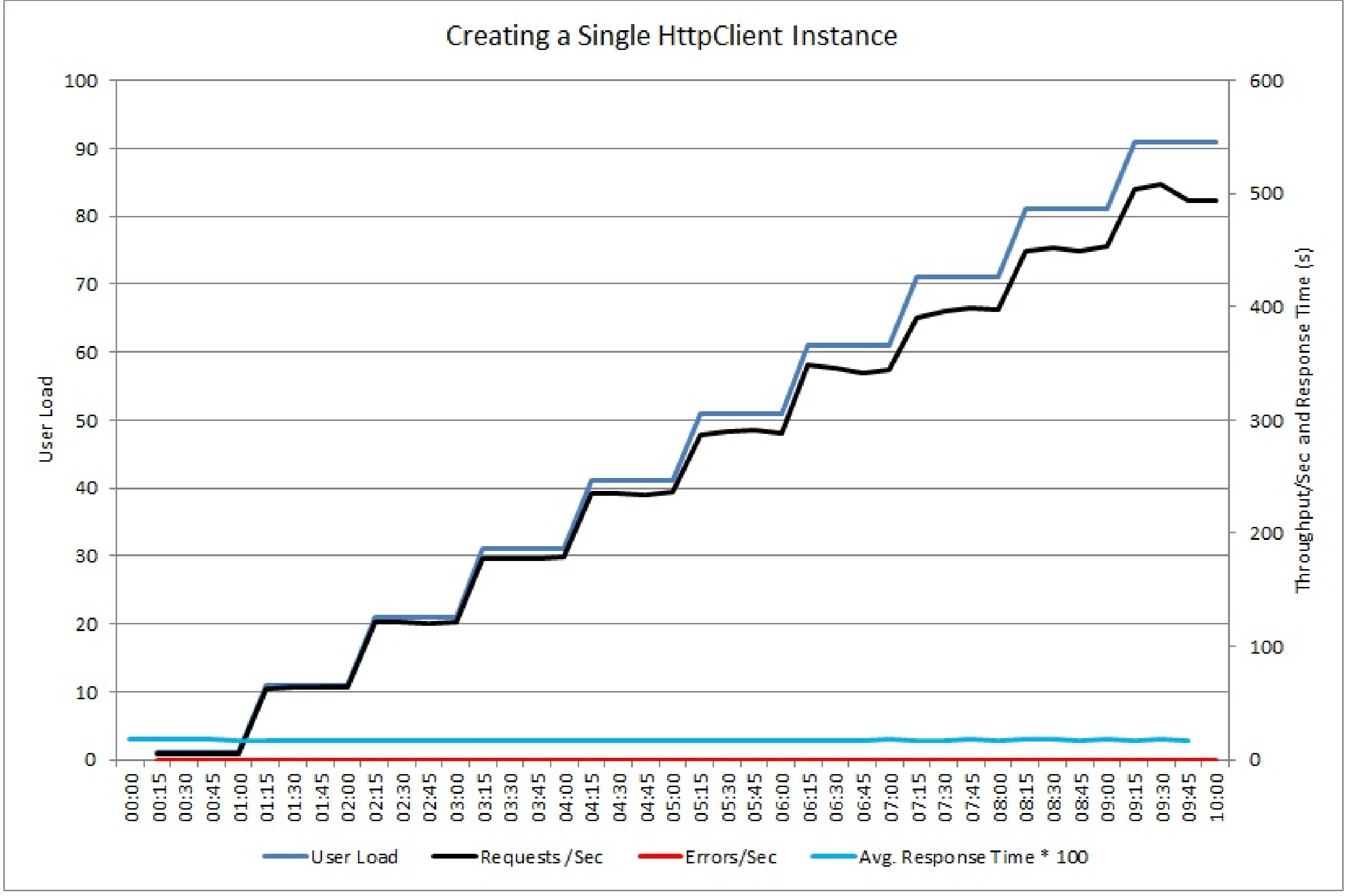
The next graph shows a similar test for a controller that creates the custom `ExpensiveToCreateService` object.



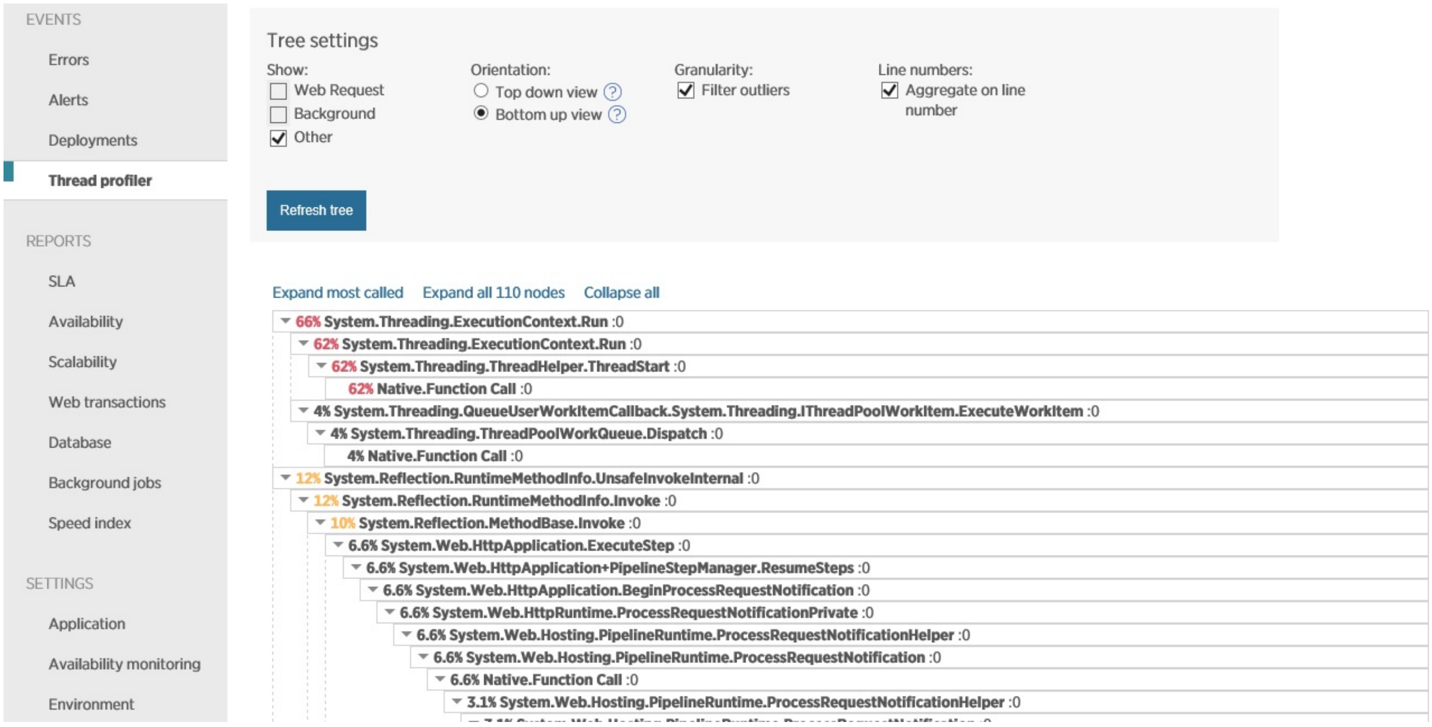
This time, the controller does not generate any exceptions, but throughput still reaches a plateau, while the average response time increases by a factor of 20. (The graph uses a logarithmic scale for response time and throughput.) Telemetry showed that creating new instances of the `ExpensiveToCreateService` was the main cause of the problem.

Implement the solution and verify the result

After switching the `GetProductAsync` method to share a single `HttpClient` instance, a second load test showed improved performance. No errors were reported, and the system was able to handle an increasing load of up to 500 requests per second. The average response time was cut in half, compared with the previous test.



For comparison, the following image shows the stack trace telemetry. This time, the system spends most of its time performing real work, rather than opening and closing sockets.



Tree settings

Show: Web Request [ ] Background [ ] [✓ Other]

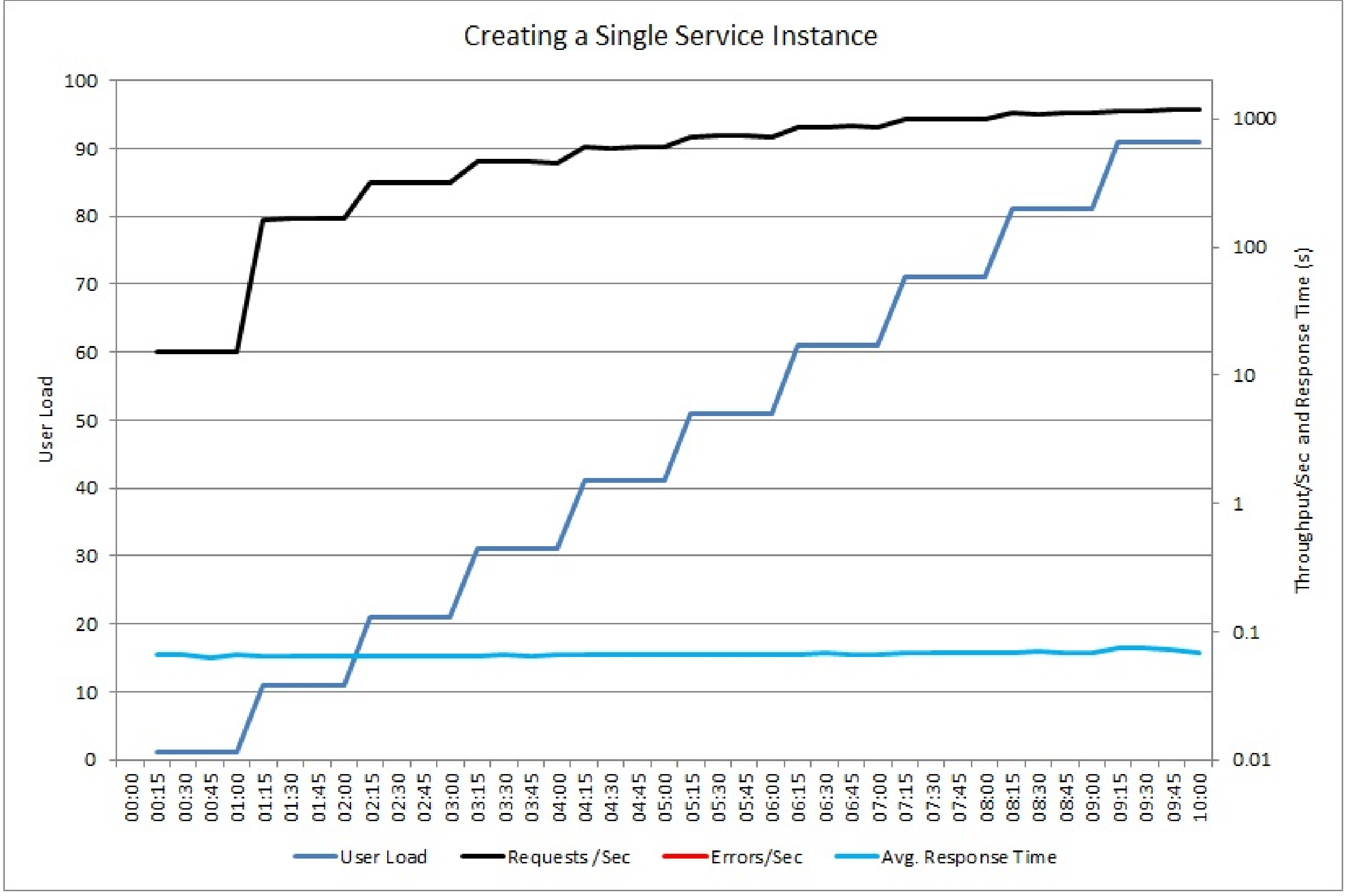
Orientation: Top down view [✓ Bottom up view]

Granularity: [✓ Filter outliers]

Line numbers: [✓ Aggregate on line number]

```
66% System.Threading.ExecutionContext.Run :0
  62% System.Threading.ExecutionContext.Run :0
    62% System.Threading.ThreadHelper.ThreadStart :0
      62% Native.Function Call :0
  4% System.Threading.QueueUserWorkItemCallback.System.Threading.IThreadPoolWorkItem.ExecuteWorkItem :0
    4% System.Threading.ThreadPoolWorkQueue.Dispatch :0
      4% Native.Function Call :0
12% System.Reflection.RuntimeMethodInfo.UnsafeInvokeInternal :0
  12% System.Reflection.RuntimeMethodInfo.Invoke :0
    10% System.Reflection.MethodBase.Invoke :0
      6.6% System.Web.HttpApplication.ExecuteStep :0
        6.6% System.Web.HttpApplication+PipelineStepManager.ResumeSteps :0
          6.6% System.Web.HttpApplication.BeginProcessRequestNotification :0
            6.6% System.Web.HttpRuntime.ProcessRequestNotificationPrivate :0
              6.6% System.Web.Hosting.PipelineRuntime.ProcessRequestNotificationHelper :0
                6.6% System.Web.Hosting.PipelineRuntime.ProcessRequestNotification :0
                  6.6% Native.Function Call :0
                    3.1% System.Web.Hosting.PipelineRuntime.ProcessRequestNotificationHelper :0
```

The next graph shows a similar load test using a shared instance of the `ExpensiveToCreateService` object. Again, the volume of handled requests increases in line with the user load, while the average response time remains low.


Creating a Single Service Instance
User Load
Throughput/Sec and Response Time (s)
100
90
80
70
60
50
40
30
20
10
0
1000
100
10
1
0.1
0.01
00:00
00:15
00:30
00:45
01:00
01:15
01:30
01:45
02:00
02:15
02:30
02:45
03:00
03:15
03:30
03:45
04:00
04:15
04:30
04:45
05:00
05:15
05:30
05:45
06:00
06:15
06:30
06:45
07:00
07:15
07:30
07:45
08:00
08:15
08:30
08:45
09:00
09:15
09:30
09:45
10:00
User Load
Requests /Sec
Errors/Sec
Avg. Response Time