

Übungsaufgabe: Distributed Unittesting

Allgemeines

- Die Übungsaufgabe ist eine Gruppenaufgabe – bitte schließen Sie sich in Gruppen von **2 oder 3 Personen** zusammen (beachten Sie aber weiter unten angeführten zusätzlichen Anforderungen für eine 3-Personen Gruppe!).
- Die Abgabe erfolgt als ein einzelnes Zip File über **Moodle**. Zum Inhalt des Zip Files siehe unten.
- Deadline für die Abgabe ist der **20.12.2015 23:55 Uhr**.

Aufgabe

Bauen Sie ein Unittest Framework, mit dessen Hilfe Unittests parallel und verteilt ausgeführt werden können. Eine kleine Unittest-Runner Applikation soll es ermöglichen eine Assembly mit Unittests auszuwählen – die Ausführung der Tests wird dann auf zuvor konfigurierte Server verteilt und die Ausführungsergebnisse dann von den Servern wieder zurückgemeldet. Auf den Servern selbst sollen die Tests ebenfalls parallelisiert laufen. Weiters sollen mehrere Clients gleichzeitig Tests zur Ausführung an die Server schicken können.

Der Unittest-Runner soll eine simple WPF Anwendung sein und in etwa wie folgt aussehen:

The screenshot shows a WPF application window with a blue border. It contains several UI elements:

- Servers:** A text label followed by a text box containing the text "server1:9000" and "server2:9000" on two lines.
- Select Unittest Assembly:** A button with the text "Select Unittest Assembly". Below it, the file path "c:\development\testassembly.dll" is displayed.
- Tests:** A text label followed by a large text box containing the following text:

```
Nameof_test_1: Success
Name_of_test_2: Failed - Expected value to be 1 but was 2
Name_of_test_3: Waiting...
Name_of_test_4: Waiting...
```
- Start:** A button with the text "Start" located at the bottom center of the window.

Unter „Servers“ können die Adressen der Unittest Server eingetragen werden. „Select Unittest Assembly“ erlaubt die Auswahl einer Assemblydatei (DLL oder EXE) mit Unittests. Wurde eine Datei ausgewählt, soll die Liste mit Unittests gefüllt werden. „Start“ startet schließlich den Test und verteilt die Unittests auf die Server. In der Liste wird der aktuelle Status der Tests angezeigt.

Bei der Serverapplikation reicht eine Konsolenanwendung. Diese soll als Kommandozeilenparameter den Parallelisierungsgrad, also die Anzahl paralleler Threads auf denen Unittests ausgeführt werden, übernehmen können. Ansonsten benötigt die Anwendung keine Konfigurations- oder Eingabemöglichkeiten.

Details zur Aufgabe

- Als Unittest-Framework können Sie entweder ein bestehendes wie NUnit verwenden, oder sich ein eigenes kleines Framework selbst schreiben. In zweiterem Fall reicht eine Unterstützung für sehr simple Unittests – es reicht wenn Methoden die mit [Test] markiert sind als Tests erkannt werden und das Ausführungsergebnis evaluiert wird, Dinge wie SetUp, Teardown und parametrisierbare Methoden müssen nicht beachtet werden.
- Bedenken Sie, dass der auszuführende Code am Server noch nicht vorhanden ist, also vom Client zum Server übertragen und dort dynamisch geladen werden muss.
- Überlegen Sie sich, wie Sie die Unittests auf die einzelnen Server verteilen, so dass eine optimale Auslastung erzielt werden kann. Da Unittests unterschiedlich lange dauern können, wird eine simple Anzahl-basierte Aufteilung nicht die gewünschten Ergebnisse erzielen.
- Um Ihre Lösung zu testen, erstellen Sie selbst geeignete „Test-Unittests“. Ein sinnvoller Test sollte einige hundert Unittests enthalten, welche auch eine unterschiedliche Dauer haben, um die Auslastung der Server realistisch testen zu können. Ein guter Ansatz um einige realistische Tests (mit realistischen Dependencies usw.) zu bekommen könnte sein, Tests für Ihr eigenes Framework zu erstellen, also quasi Ihr Framework sich selbst testen zu lassen.
- Bei der Implementierung sind Sie auf keine speziellen Technologien beschränkt. Die einzige Anforderung ist, dass Sie .Net (C#) und Visual Studio für die Aufgabe verwenden. Verwenden Sie gerne beliebige Frameworks für Kommunikation, Unittesting oder auch für die GUI (zumindest solange diese Frameworks nicht die gerade die Verteilung und Parallelisierung von Unittests bereits selbst lösen – denn dies soll ja Ihre Aufgabe sein – im Zweifelsfall fragen Sie nach, ob Sie ein bestimmtes Framework verwenden dürfen).

Für Gruppen mit 3 Personen

- Implementieren Sie zusätzlich die Möglichkeit, dass Unittests so markiert werden können, dass Sie auf dem Server exklusiv laufen, also dass zu dieser Zeit auf diesem Server kein anderer Test laufen darf (z.B. nützlich wenn Tests bestimmte Ressourcen brauchen die mit anderen Tests nicht geteilt werden können).
- In .Net können Assemblies die einmal geladen sind nicht mehr explizit entladen werden. Um dieses Problem zu umgehen, führen Sie Unittests auf den Servern in einer eigenen AppDomain pro Testdurchlauf aus, die nach Beendigung des Testdurchlaufs wieder entladen werden kann.

Bericht

Zusätzlich zur Implementierung soll ein etwa 3-4 seitiger Bericht geschrieben werden, der zumindest folgendes beschreibt:

- Beschreiben Sie Ihre Implementierung. Welche Entscheidungen wurden getroffen und warum?
- Wie sieht ihre Lösung in Bezug auf Concurrency, Zugriffskoordination und Synchronisierung aus?

- Für welches Kommunikationsframework haben Sie sich entschieden und warum?
- Vergleichen Sie die unterschiedlichen Ausführungszeiten mit zumindest 1 und 2 Servern, und mit unterschiedlichen Parallelisierungsgraden auf den Servern.
- Auf welche Schwierigkeiten sind Sie gestoßen?
- Wieviel Zeit haben Sie wofür gebraucht?

Abgabe

Im Zip File für die Abgabe sollen folgende Dinge enthalten sein:

- Visual Studio Solution mit der Lösung
- Jegliche benötigte externe Libraries, sofern diese nicht über NuGet automatisch heruntergeladen werden können.
- Zum Testen ein bat-File, mit dem der Runner und 2 Instanzen des Servers (auf einem Rechner) gestartet werden können. Der Runner soll Serveradressen und Testassembly bereits vorkonfiguriert haben. (Sie müssen die vorkompilierte Anwendung aber nicht mit ausliefern – es reicht, wenn das bat-File nach dem kompilieren der Visual Studio Solution funktioniert.)
- Bericht im PDF Format wie oben beschrieben

1. Getroffene Entscheidungen

1.1. Unittest-Framework

Als Unittest-Frameworks standen u.a. folgende zur Auswahl:

- NUnit (<http://sourceforge.net/projects/nunit/>)
- xUnit.net (<https://github.com/xunit/xunit>)
- Ein in Visual Studio integriertes Testframework. Dieses kann durch Hinzufügen der Assembly Microsoft.VisualStudio.TestTools.UnitTesting und des Namespaces Microsoft.VisualStudio.TestTools.UnitTesting in einem Projekt verwendet werden. Getestet wurde mit Visual Studio 2013.

NUnit und xUnit besitzen einige Abhängigkeiten zu anderen Assemblies. Das Visual-Studio-basierte ist abhängig von den Assemblies, welche von Visual Studio mitgeliefert werden.

Daher haben wir uns für die Implementierung eines eigenen Unittest-Frameworks entschieden. Die Implementierung eines eigenen Unittest-Frameworks in .NET ist relativ einfach möglich. Es wurden folgende Komponenten implementiert:

- *Attribute*, welche eine Testmethode beschreiben. Es wurden die Attribute [Test] und [ExpectedException] implementiert. [Test] markiert eine Methode als Testmethode, welche von einem TestRunner ausgeführt werden soll. Wird von einer Methode erwartet, dass diese eine Exception wirft, kann die Exception mit dem Attribut [ExpectedException] angegeben werden.
- *TestRunner*: Nimmt eine Assembly entgegen und führt die darin mit [Test] markierten Methoden aus. Einem TestRunner kann auch ein einzelner Test übergeben werden.
- *Assert*: Es wurde eine statische Klasse Assert implementiert, welche eine Methode void AreEqual(object expected, object actual) zur Verfügung stellt. Schlägt der AreEqual-Vergleich fehl, wird eine AssertFailedException geworfen.
- *Exceptions*: Es wurden drei Exception-Klassen implementiert: AssertFailedException, ExpectedExceptionNotThrownException und NoExceptionThrownException. Die Details zu den Exceptions können dem Code entnommen werden.
- *Test*: Tests werden durch die Klasse Test repräsentiert. Ein Test enthält den Namen der Testmethode, eine Nachricht („Test passed.“, „Test failed“ oder der Text einer Exception), die Dauer des Tests, den Status (NONE, IN_PROGRESS, PASSED, FAILED), Infos zur Klasse der Methode und die Adresse des Servers, auf welchem der Test ausgeführt wurde.

1.2. Kommunikationsframework

Als Kommunikationsframework standen u.a. WCF und XcoAppSpace zur Verfügung.

Wir haben uns für XcoAppSpace entschieden, da es die Erstellung von asynchronen, verteilten Anwendungen ermöglicht und wir dieses Framework auf seine Verwendbarkeit prüfen wollten. WCF wurde nicht evaluiert.

1.3. GUI des Clients

Die Unittest-Runner Applikation wird in unserer Lösung als *Client* bezeichnet. Für die GUI des Clients wird WPF verwendet. Für die asynchrone Aktualisierung der GUI und für die Bindung der Daten wird das Muster MVVM (Model View ViewModel) verwendet.

2. Aufbau der Visual-Studio-Solution

Für ein besseres Verständnis des in Kapitel 3 erklärten Algorithmus, wird kurz auf den Aufbau der Visual-Studio-Solution eingegangen. Die Visual-Studio-Solution *Di stUni tTesti ng* teilt sich in folgende Projekte:

- *PCG3.Client.GUI:*
Enthält den xaml-Code zur Beschreibung der GUI und die Main-Methode zum Starten des Clients. Diesem Projekt kann der Pfad einer Assembly und eine Liste von Server-Adressen übergeben werden. Die Server-Adressen müssen in der Form `host: port` angegeben werden.
- *PCG3.Client.Logic:*
Enthält die client-seitige Logik für die Kommunikation mit den Servern in Form einer Klasse *ClientLogic*. *ClientLogic* übernimmt das Extrahieren der Testmethoden aus einer Assembly, das Übertragen einer Assembly an die Server und das Verteilen der Tests an die Server.
- *PCG3.Client.ViewModel:*
Enthält die Realisierung des MVVM-Musters in Form einer Klasse *MainVM*. *MainVM* ruft Methoden von *ClientLogic* auf. Wird der „Start“-Button gedrückt, werden in einem neuen Task das Übertragen der Assembly und das Verteilen der Tests an die Server durchgeführt.
- *PCG3.Middleware:*
Enthält den *XcoAppSpace*-Kontrakt zwischen einem Client und einem Server. Es wurden zwei Worker implementiert: *AssemblyWorker* übernimmt das Übertragen und server-seitige Laden einer Assembly. *TestWorker* übernimmt das Übermitteln und server-seitige Ausführen einer Menge von Tests.
- *PCG3.Server:*
Enthält eine Konsolenanwendung zum Starten eines Servers und die Implementierung der Worker. Die Konsolenanwendung nimmt als ersten Parameter den Parallelisierungsgrad und als zweiten Parameter den Port des Servers. Der Parallelisierungsgrad wird in unserer Lösung als *Cores* bezeichnet. Dieser ist standardmäßig bei keiner Angabe auf 4 gesetzt. Der Port des Servers ist standardmäßig 9000.
Von der Konsolenanwendung wird darüber hinaus die Erzeugung der Worker im Application Space übernommen. Dazu wird die Methode *RunWorker<...>()* von *XcoAppSpace* verwendet. Die client-seitige Verbindung zu den Workern mittels *ConnectWorker<...>()* wird in der Klasse *ClientLogic* durchgeführt.
- *PCG3.TestFramework:*
Enthält die Implementierung des eigenen Unittest-Frameworks.
- *PCG3.TestUnitTests:*
Enthält vier Klassen mit insgesamt 120 Test-Unittests.
- *PCG3.TestUnitTestsSimple:*
Enthält eine Klasse mit sechs Methoden, wobei eine Methode nicht mit `[Test]` gekennzeichnet ist.
- *PCG3.Util:*
Enthält Hilfsklassen zum Loggen auf die Konsole, zur formatierten Ausgabe der aktuellen Zeit usw.

3. Algorithmus

Das verteilte Ausführen von Unittests wird in zwei Schritte durchgeführt:

- Zu Beginn wird die Assembly mit den auszuführenden Testmethoden an alle Server verteilt und auf den Servern geladen.
- Danach werden die Tests an die Server verteilt. Die Verteilungslogik befindet sich in der Klasse *ClientLogic* des Projektes *PCG3.Client.Logic*.

In den folgenden zwei Abschnitten werden die Details erläutert.

3.1. Übertragen der Assembly zu den Servern

Der Benutzer wählt mittels Client die Assembly mit den auszuführenden Tests aus. Diese Assembly existiert auf den Servern zu diesem Zeitpunkt noch nicht. Daher muss die Assembly zu den Servern übertragen werden. Dazu wird die Assembly mittels

`File.ReadAllBytes(assemblyPath)` eingelesen und als Bytestrom übertragen. Am Server wird der Assembly-Bytestrom mittels `Assembly.Load(...)` gelesen und die Assembly dabei geladen.

Zu Beginn hatten wir überlegt, die Assembly in die auszuführenden Methoden zu zerteilen, neue Assemblies zu erzeugen und diese dann zu den Servern zu übertragen. Da die Assemblies nur einige KB groß sind und nur eine Assembly je Client zu den Servern übertragen wird, haben wir uns gegen diese Variante entschieden.

3.2. Verteilen der Tests an die Server

Beim Starten eines Servers wird diesem die Anzahl der Cores (der Parallelisierungsgrad) als Parameter mitgegeben. Die Cores sind nur dem Server und nicht dem Client bekannt. Eine Tatsache, welche uns erst bei der Verwendung von *XcoAppSpace* klar geworden ist, ist hier sehr wichtig: *XcoAppSpace* speichert keinen Zustand. Es ist daher nicht möglich beim Erzeugen eines Worker (TestWorker) mittels `RunWorker<...>()` die Cores eines Servers zu speichern. Der Client kann somit nicht direkt die Anzahl der Cores je Server abfragen. Der Client bietet den Servern stattdessen die noch verfügbaren Tests an. Stehen dem Server noch Cores zur Verfügung, reserviert (allokiert) der Server Cores für die Ausführung von Tests des Clients.

Das Verteilen der Tests passiert in einem 3-Schritte-Ansatz:

- Schritt 1 - Alloc: Der Client hat eine bestimmte Anzahl an noch verfügbaren, unverteilter Tests. Er fragt broadcast-ähnlich jeden Server, ob diese frei Cores haben zum Ausführen von einen oder mehreren Tests. Die Server antworten mit einer bestimmten Anzahl an Cores, oder mit 0 sollte der Server bereits ausgelastet sein.
- Schritt 2 - Run: Hat ein Server noch verfügbare Cores, schickt der Client diesem Server die vereinbarte Anzahl an Tests. Der Server erzeugt je Test einen neuen Task. Sobald das Ausführungsergebnis eines Tests vorliegt, antwortet der Server dem Client.
- Schritt 3 - Free: Sobald das Ausführungsergebnis eines Tests beim Client eingelangt ist, fordert der Client den Server auf, einen Core freizugeben.
Die Kontrolle geht dabei vom Client aus. Hier liegt auch ein Schwachpunkt: Sollte der Client aus irgendeinem Grund wegbrechen, kann der allokierte Core am Server nicht mehr freigegeben werden. Uns war nicht ganz klar, wie man diese Situation mit *XcoAppSpace* lösen soll.

Der beschriebene 3-Schritte-Ansatz wird solange in einer Schleife ausgeführt, bis alle Tests verteilt und ausgeführt wurden. Zu beachten ist hier, dass in unserer Lösung die Kontrolle immer vom Client ausgeht. Der Client pollt die Server und bietet ihnen verfügbare Tests an. Als Intervall wurde eine Sekunde angenommen. Ein längeres Intervall (z.B. 5 Sekunden) zwischen zwei Allocs könnte dazu führen, dass die Server unnötig lange auf neue Tests warten müssen. Ein kürzeres Intervall (z.B. 500 Millisekunden) führt möglicherweise zu unnötig

häufigen Alloc-Anfragen, speziell wenn die Server über längere Zeit mit langlaufenden Tests beschäftigt sind.

Der 3-Schritte-Ansatz wird mit XcoAppSpace so gelöst, dass je Schritt Nachrichten versendet werden. Dazu wird die Methode `Post()` eines Workers aufgerufen und mittels `Receive()` auf die Antwort gewartet. Die Kommunikation läuft dabei so ab, dass der Client eine Nachricht postet, die Nachricht am Server verarbeitet wird und die Antwort zum Client zurückgepostet wird. Wie weiter oben beschrieben, kann innerhalb eines Workers die Anzahl der verfügbaren Cores nicht gespeichert werden. Die Kommunikation muss zwischen Client und Server (der Konsolenanwendung und nicht dem serverseitigen Worker) ablaufen, um auf die verfügbare Anzahl an Cores zugreifen zu können. Dies ist mit einem reinen Post-Receive nicht möglich. Daher wurde der *Publish-Subscribe-Mechanismus* von XcoAppSpace verwendet. Der Client veröffentlicht („publish“) eine Nachricht (z.B. eine Alloc-Anfrage) und alle Server, welche sich registriert („subscribe“) haben, erhalten diese Nachricht. Das Subscribe der Server wird in der Konsolenanwendung des Servers gemacht, in welcher auch die Anzahl der verfügbaren Cores bekannt ist. Dort werden die Cores bei einem Alloc dekrementiert und bei einem Free inkrementiert. In der Subscribe-Nachricht des Clients ist der ResponsePort angegeben. Der Server schickt seine Antwort an diesen Port.

Für die Synchronisation, z.B. für die Ausgabe von Log-Meldungen auf die Konsole oder für den geteilten Zugriff auf die Anzahl der Cores eines Servers, werden locks auf Lock-Objekte gesetzt.

Post und Receive bzw. Publish-Subscribe-Nachrichten werden asynchron versendet. Daher kann es vorkommen, dass der Application Space geschlossen wird, bevor von jedem Server eine Nachricht zum Client zurückgeschickt wurde. Daher muss an bestimmten Stellen auf die Antworten der Server gewartet werden. Dazu werden CountdownEvents verwendet. Ein CountdownEvent ist zu Beginn auf die Anzahl der Tests gesetzt und wird jedes Mal um eins dekrementiert, wenn der Server einen Test ausgeführt und einen Core freigegeben hat.

4. Vergleich der Ausführungszeiten

Für die Performance-Tests wurde die Assembly des Projekts PCG3. TestUnitTests verwendet. Es enthält vier Klassen mit insgesamt 120 Test-Unittests.

Die Klassen enthalten

- 5 Basistests zum Überprüfen des Testframeworks.
- 5 Tests mit einer Dauer von 5 Sekunden jeweils.
- 5 Tests mit einer Dauer von 10 Sekunden jeweils.
- 5 Tests mit einer Dauer von 15 Sekunden jeweils.
- 50 Tests mit einer Dauer zwischen 0 und 1000 ms.
- 50 Tests mit einer Dauer von 10 ms.

Die Tests wurden ausgeführt mit

- Einem Client und einen oder zwei Servern, mit gleicher Core-Anzahl.
- Zwei Clients und einem oder zwei Servern, mit gleicher Core-Anzahl.

Die Ausführungszeiten sind in folgender Tabelle aufgelistet.

Die Ausführung von 120 Tests eines Clients auf einem Server mit einem Core benötigt ca. 4 Minuten und 20 Sekunden. Die beste Zeit wird mit zwei Servern und 8 Cores jeweils erreicht. Die Ausführungszeit liegt bei ca. 19 Sekunden. Dies ergibt einen Speedup von rund 13,7.

Client	Server		Total Execution Time
	- Server1 = localhost:9000	- Server2 = localhost:9001	
1	Server1	1 core	00:04:20.8149461
1	Server1	2 cores	00:02:12.4195201
1	Server1	1 cores	00:02:09.5824521
	Server2	1 cores	

1	Server1	2 cores	00:01:05.5216858
	Server2	2 cores	
1	Server1	4 cores	00:00:33.5054170
	Server2	4 cores	
1	Server1	8 cores	00:00:18.8364649
	Server1	8 cores	
1	Server1	16 cores	00:00:21.4001568
	Server1	16 cores	
2	Server1	1 core	00:07:27.4426314 (im Durschnitt pro Client)
2	Server1	2 cores	00:03:46.8230789
2	Server1	1 cores	00:03:36.7434720
	Server2	1 cores	
2	Server1	2 cores	00:01:50.8766487
	Server2	2 cores	
2	Server1	4 cores	00:00:56.1662903
	Server2	4 cores	
2	Server1	8 cores	00:00:29.6699607
	Server1	8 cores	
2	Server1	16 cores	00:00:20.2352762
	Server1	16 cores	

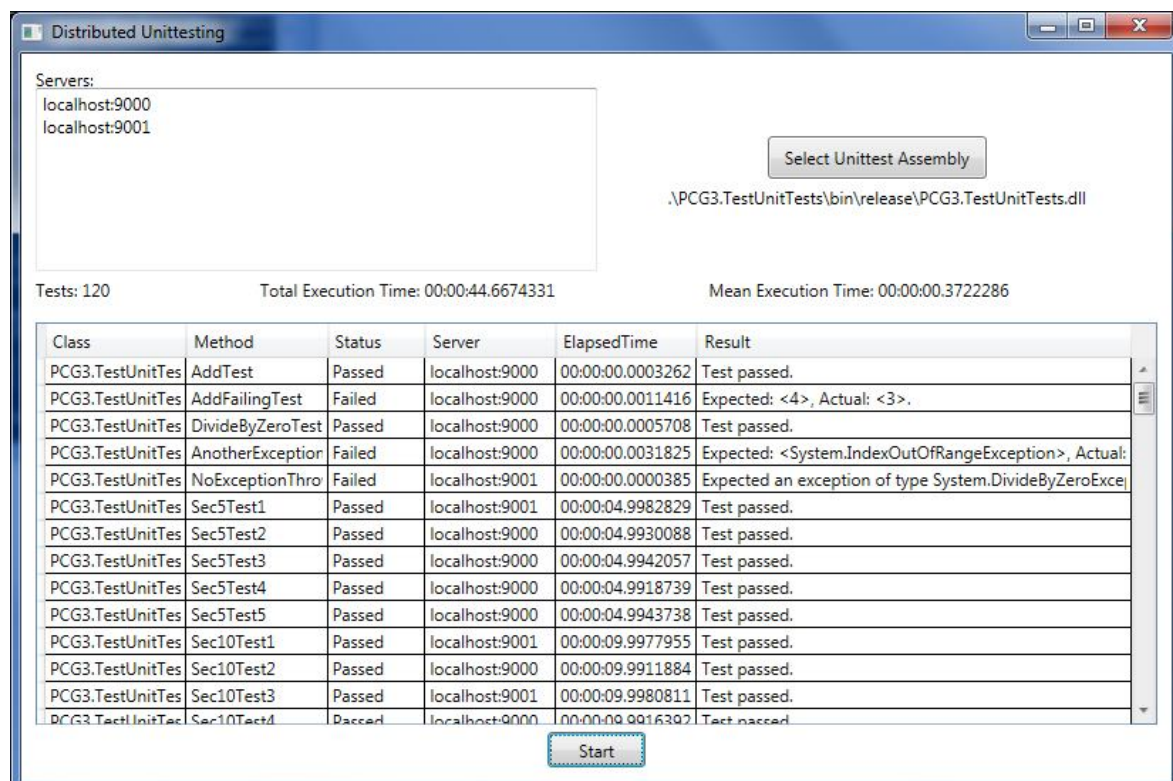
5. Ausführung des Clients und Servers mittels Skript

Für das Starten eines Clients und zweier Server gibt es ein bat-Skript run.bat im Verzeichnis DistributedUnittesting.

Im Skript kann angepasst werden, ob man die kompilierten Dateien im Unterverzeichnis Debug oder im Unterverzeichnis Release verwenden möchte.

Beim erfolgreichen Start mittels Skript sollte die GUI des Clients, ein Log-Fenster für den Client und zwei Konsolenfenster für die Server zu sehen sein.

Dies ist beispielhaft wie folgt dargestellt:




```
Server1 (localhost:9000, 4 cores)
e core request.
[20151223-20:40:57.827]: [S/Progr] [Me!sabine-think:9000] [C!sabine-think:4498]
Free core.
[20151223-20:40:57.827]: [S/TstWo] [Me!localhost:9000] [C!sabine-think:4498] Fre
e core request.
[20151223-20:40:57.827]: [S/Progr] [Me!sabine-think:9000] [C!sabine-think:4498]
Free core.
[20151223-20:40:58.777]: [S/TstWo] [Me!localhost:9000] [C!sabine-think:4498] 1 t
ests available.
[20151223-20:40:58.777]: [S/Progr] [Me!sabine-think:9000] [C!sabine-think:4498]
Requested: 1, Allocated: 1, Free: 3
[20151223-20:40:58.777]: [S/TstWo] [Me!localhost:9000] [C!sabine-think:4498] Run
tests.
[20151223-20:40:58.777]: [S/TstWo] [Me!localhost:9000] [C!sabine-think:4498] Run
test ...
[20151223-20:40:58.787]: [S/TstWo] [Me!localhost:9000] [C!sabine-think:4498] [Re
sult]
MethodName=Short10msTest50, Message=Test passed., Failed=False, Status=Passed, T
ype=PCG3.TestUnitTests.ManyShortRunningTests, Server=localhost:9000
[20151223-20:40:58.787]: [S/TstWo] [Me!localhost:9000] [C!sabine-think:4498] Fre
e core request.
[20151223-20:40:58.787]: [S/Progr] [Me!sabine-think:9000] [C!sabine-think:4498]
Free core.
```

```
Server2 (localhost:9001, 2 cores)
[20151223-20:40:57.787]: [S/TstWo] [Me!localhost:9001] [C!sabine-think:4498] Run
test ...
[20151223-20:40:57.797]: [S/TstWo] [Me!localhost:9001] [C!sabine-think:4498] [Re
sult]
MethodName=Short10msTest48, Message=Test passed., Failed=False, Status=Passed, T
ype=PCG3.TestUnitTests.ManyShortRunningTests, Server=localhost:9001
[20151223-20:40:57.797]: [S/TstWo] [Me!localhost:9001] [C!sabine-think:4498] [Re
sult]
MethodName=Short10msTest49, Message=Test passed., Failed=False, Status=Passed, T
ype=PCG3.TestUnitTests.ManyShortRunningTests, Server=localhost:9001
[20151223-20:40:57.817]: [S/TstWo] [Me!localhost:9001] [C!sabine-think:4498] Fre
e core request.
[20151223-20:40:57.817]: [S/TstWo] [Me!localhost:9001] [C!sabine-think:4498] Fre
e core request.
[20151223-20:40:57.817]: [S/Progr] [Me!sabine-think:9001] [C!sabine-think:4498]
Free core.
[20151223-20:40:57.817]: [S/Progr] [Me!sabine-think:9001] [C!sabine-think:4498]
Free core.
[20151223-20:40:58.777]: [S/TstWo] [Me!localhost:9001] [C!sabine-think:4498] 0 t
ests available.
[20151223-20:40:58.777]: [S/Progr] [Me!sabine-think:9001] [C!sabine-think:4498]
Requested: 0, Allocated: 0, Free: 2
```

```
Client
[20151223-20:40:57.827]: [C/MaiUM] Updated test MethodName=Short10msTest44, Mess
age=Test passed., Failed=False, Status=Passed, Type=PCG3.TestUnitTests.ManyShort
RunningTests, Server=localhost:9000
[20151223-20:40:57.827]: [C/Logic] [Me!sabine-think:4498] Free core.
[20151223-20:40:57.827]: [C/Logic] [Me!sabine-think:4498] 3 tests not completed
yet.
[20151223-20:40:57.837]: [C/Logic] [Me!sabine-think:4498] Free core.
[20151223-20:40:57.837]: [C/Logic] [Me!sabine-think:4498] 2 tests not completed
yet.
[20151223-20:40:58.777]: [C/Logic] [Me!sabine-think:4498] Server: localhost:9000
, Allocated: 1
[20151223-20:40:58.777]: [C/Logic] [Me!sabine-think:4498] Server: localhost:9001
, Allocated: 0
[20151223-20:40:58.787]: [C/Logic] [Me!sabine-think:4498] [Result]
MethodName=Short10msTest50, Message=Test passed., Failed=False, Status=Passed, T
ype=PCG3.TestUnitTests.ManyShortRunningTests, Server=localhost:9000
[20151223-20:40:58.787]: [C/MaiUM] Updated test MethodName=Short10msTest50, Mess
age=Test passed., Failed=False, Status=Passed, Type=PCG3.TestUnitTests.ManyShort
RunningTests, Server=localhost:9000
[20151223-20:40:58.787]: [C/Logic] [Me!sabine-think:4498] Free core.
[20151223-20:40:58.787]: [C/Logic] [Me!sabine-think:4498] 1 tests not completed
yet.
[20151223-20:40:58.787]: [C/MaiUM] Distributed tests to servers.
```

6. Aufgetretene Schwierigkeiten

Es gab folgende Schwierigkeiten:

- Im ViewModel des Clients werden das Übermitteln der Assembly und das Verteilen der Tests an die Server angestoßen. Dies wird in einem eigenen Task gemacht um das Einfrieren der GUI zu vermeiden. Soll nun aus diesem Task heraus die GUI des Client aktualisiert werden, dann ist dies nicht direkt möglich. Die GUI darf nur vom GUI-Thread aktualisiert werden. Um aus einem Task die GUI aktualisieren zu können, muss ein Dispatcher der GUI verwendet werden. Dieser Dispatcher wird beim Starten des Clients an das ViewModel übergeben. Mittels `GuiThreadDispatcher.Invoke() => { ... }` kann dann die GUI vom ViewModel aus aktualisiert werden.
- Enthält eine `XcoAppSpace`-Nachricht Komponenten, welche nicht serialisiert werden können, wird die Nachricht nicht übertragen. In unserem Fall befanden sich in der Klasse `Test Properties` vom Typ `MethodInfo` und vom Typ `Exception`. Die Ursache blieb lange Zeit unentdeckt, da keine Fehler geworfen oder in den Output geschrieben wurden.
- Wartet man auf die Antwort einer Post-Nachricht nicht und wird der Application Space zuvor geschlossen, bekommt man Exceptions vom Typ `System.ObjectDisposedException`. Dies konnten wir vermeiden, indem nun an bestimmten Stellen mittels `CountdownEvent` gewartet wird.

7. Aufgewendete Zeit

Der aufwändigste und fehlerträchtigste Teil war die Implementierung der Verteilungslogik mittels 3-Schritte-Ansatz und Publish-Subscribe.

Es wurden insgesamt 80 Stunden aufgewendet.