DAD 2021-21
Lab.3 Asynchronous Programming + Intro to G-RPC

Lab. starting at 14:10

Async Programming

Asynchronous Programming

- Asynchronous Programming allows abstract concurrent activities without Thread management.
- Async. Prog. uses the abstraction of Tasks.
- Tasks can be waited on until the asynchronous activity (e.g I/O) is done.
- Tasks can be started explicitly on different threads.

Task & Task<TResult>

- Tasks represent an asynchronous operation.
- They can be waited on.
- Task<TResult> return a TResult.

```
  // Create a task and supply a user delegate by using a
lambda expression.
  Task taskA = new Task( () => Console.WriteLine("Hello
from taskA."));
  // Start the task.
  taskA.Start();
  taskA.Wait();
```

async

- Allows running code asynchronously on a runtime managed thread pool.
- Async methods can contain await-ed operations.
- Async methods return a Task or Task<TResult>
- See: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/

await

- Can be called on anything implementing the GetAwaiter method.
- Blocks the current thread until an asynchronous result is returned.
- For example:

```
Task<int> downloading = DownloadWebpageAsync();
// do something else and then get the results
int bytesLoaded = await downloading;
```

G-RPC in C#

G-RPC Proto Buffers

- Specify the protocol between client and server: the service interfaces in a language agnostic syntax (see https://developers.google.com/protocol-buffers/docs/proto3)

```
syntax = "proto3";

service ChatServerService {

  rpc Register (ChatClientRegisterRequest) returns
(ChatClientRegisterReply);

}

message ChatClientRegisterRequest {

  string nick = 1;

  string url = 2;

}

message ChatClientRegisterReply {

        bool ok = 1;

}
```

Server (1)

- Implements Services described in Protobuf:

```
// ChatServerService is the namespace defined in the protobuf
   // ChatServerServiceBase is the generated base implementation of the service
   public class ServerService : ChatServerService.ChatServerServiceBase {
      // example of Server data structure
      Dictionary<string, string> clientMap = new Dictionary<string, string>();

      public ServerService() {
      }

      public override Task<ChatClientRegisterReply> Register(
         ChatClientRegisterRequest request, ServerCallContext context) {
         return Task.FromResult(Reg(request));
}
```

Server (2)

```
    public ChatClientRegisterReply Reg(
        ChatClientRegisterRequest request) {
        lock (this) {
            clientMap.Add(request.Nick, request.Url);
        }
        return new ChatClientRegisterReply
        {
            Ok = true
        };
    }
}
```

Server (3)

- Responds to client requests. Grpc.Core.Server is multithreaded!

```
 static void Main(string[] args) {
            Server server = new Server
               {
                  Services = { ChatServerService.BindService(new ServerService()) },
                  Ports = { new ServerPort("localhost", Port,
ServerCredentials.Insecure) }
               };
               server.Start();
              Console.ReadKey();
               server.ShutdownAsync().Wait();
            }
```

Client

- Can do calls to a server.
- Steps:
  - Disable HTTPS (optional):
    ```
    AppContext.SetSwitch("System.Net.Http.SocketsHttpHandler.Http2UnencryptedSupport", true);
    ```
  - Create Channel:

```
GrpcChannel channel = GrpcChannel.ForAddress("http://localhost:50051");
```

  - Create Client:

```
var client = new ChatServerService.ChatServerServiceClient(channel);
```

  - Do calls:

```
Client.Register(registerRequest);
```

Server Development

- Create Project
- Add code package (Tools->NuGet Package Manager):
  - Grpc.Net.Core, which contains the .NET G-RPC Core.
  - Google.Protobuf, which contains protobuf message APIs for C#.
  - Grpc.Tools, which contains C# tooling support for protobuf files.
- Add proto folder and protobuf file
- Add protobuf to project by adding following line to the G-RPC ItemGroup in the project file:
  ```
  <Protobuf Include="protos\ChatServices.proto" GrpcServices="Server" />
  ```
- Implement services
- Add server start code. Done! ;-)

Client Development

- Create Project
- Add code packages (Tools->NuGet Package Manager):
  - Grpc.Net.Client, which contains the .NET Core client.
  - Google.Protobuf, which contains protobuf message APIs for C#.
  - Grpc.Tools, which contains C# tooling support for protobuf files.
- Add protos folder and copy of server protobuf file
- Define client namespace in protobuf file:
  `option csharp_namespace = "ChatClient";`
- Add client code: create Channel, Client and server calls. Done! ;-)