gRPC communication in .Net Framework

Topical when gRPC is used as communication level between client and server when both work on .Net Framework (not .Net Core). Focused on security, interceptions and logging.

The "Code-first" approach used in the prototype **PR-30263** - Implement gRPC proxy prototypes IN PROGRESS is implemented in **protobuf-net.Grpc** library.

Security

gRPC supports security on two levels:

- *Channel-level* authentication uses a client certificate that's applied at the connection level. It can also include call-level authentication /authorization credentials to be applied to every call on the channel automatically.
    - **Insecure**
    - **Server-side TLS**
    - **Mutual TLS**
- *Call-level* authentication/authorization is usually handled through tokens that are applied in metadata when the call is made. Examples:
    - **JWT Bearer Token -easy support in .Net Core**
    - Azure Active Directory
    - IdentityServer
    - OAuth 2.0
    - OpenID Connect
    - WS-Federation

It's possible to use either or both of these mechanisms to help secure the service service. The call authentication methods are all based on tokens. The only real difference is how the tokens are generated and the libraries that are used to validate the tokens.
The prototype is using Jwt-token via interceptors.

Insecure

All data transferred between client and server is not encrypted.

Insecure - Server

```
// Server insecure channel
var server = new Grpc.Core.Server{new ServerPort(host, port, ServerCredentials.Insecure)};
```

Insecure - Client

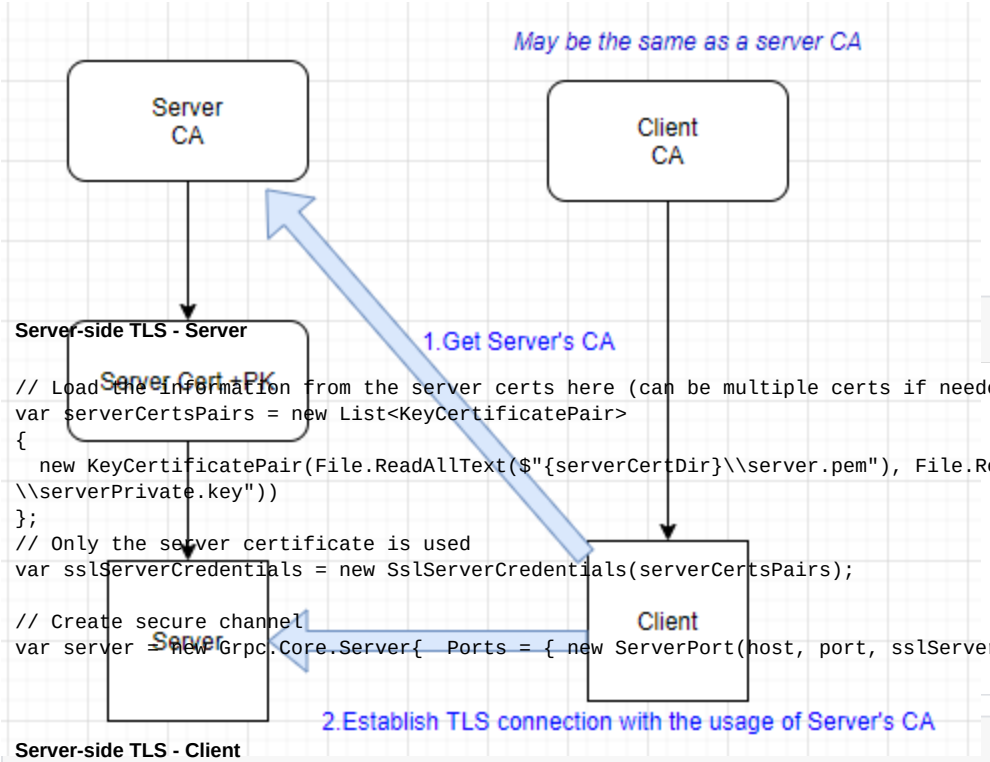
```
// Client insecure channel
var channel = new Channel(host, port, ChannelCredentials.Insecure);
```

Server-side TLS (the case for Pitram internal communications)

All the data is encrypted, but only the server needs to provide its TLS certificate or **its CA certificate** to the client. Can be used if the server doesn't care which client is calling its API.

I would recommend to read up on the following topics first:

- Why do I need a certificate to establish a secure gRPC connection as a client?
- Follow-Up: Why don't I need a certificate to establish a secure connection from a browser?



Server-side TLS - Server

```
// Load the information from the server certs here (can be multiple certs if needed)
var serverCertsPairs = new List<KeyCertificatePair>
{
  new KeyCertificatePair(File.ReadAllText($"{serverCertDir}\\server.pem"), File.ReadAllText($"{serverCertDir}
\\serverPrivate.key"))
};
// Only the server certificate is used
var sslServerCredentials = new SslServerCredentials(serverCertsPairs);

// Create secure channel
var server = new Grpc.Core.Server{  Ports = { new ServerPort(host, port, sslServerCredentials ) }};
```

Server-side TLS - Client

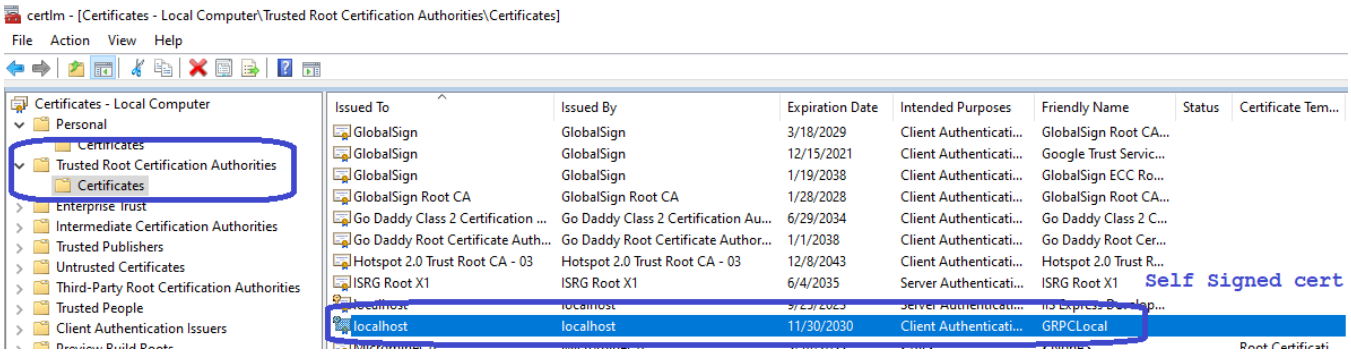
```
// Client needs to have server CA cert or server cert (just PEM without no server's private key)
var channelCredentials = new SslCredentials(server_CA_Cert);

// Create secure channel
var channel = new Channel(host, port, channelCredentials);
```

Thoughts: To get the client connected, you need to give it to the server.crt (or server.pem) public key. In normal operation, this key can be fetched from a certificate authority (CA), but since we're doing internal RPC, the public key must be shipped with the application.
It is an open question about how to manage certificates in a larger system, but potentially an internal certificate authority resolves these problems.

Peter Chapman proposes to use the very first call over an insecure channel to get the server's cert data. Later, after receiving, to construct the secure server-side channel and use it for further communication.

IMO, in a production environment, it's better to store certs in some trusted storage and load certs (by thumbprints, for example) if needed from there. See *CertUtils* class in prototype for more details.

The prototype uses local self-signed server certificate. See the script from Peter Chapman comment on how to generate certificate or just use OpenSSL utility 🙂



Mutual

Used when the server also needs to verify who's calling its services. So in this case, both client and server must provide their TLS certificates to the other.

The example with a client certificate can be found in the prototype (commented):

Mutual TLS - Server

```
// Server
// Load the information from the server certs here (can be multiple certs if needed)
var serverCertsPairs = new List<KeyCertificatePair>
{
  new KeyCertificatePair(File.ReadAllText($"{serverCertDir}\\server.pem"), File.ReadAllText($"{serverCertDir}
\\serverPrivate.key"))
};
// Use the client\clientRoot cert to mutually secure connection (parameter: clientRootPem)
var sslServerCredentials = new SslServerCredentials(serverCertsPairs, client_CA_Cert,
SslClientCertificateRequestType.RequestAndVerify);

// Create secure channel
var server = new Grpc.Core.Server{  Ports = { new ServerPort(host, port, sslServerCredentials ) }};
```

Mutual TLS - Client

```
// Client
// Load the information from the client cert here
var clientCertPair = new KeyCertificatePair(clientCert, clientKey);
// Use the client cert to mutually secure connection
var channelCredentials = new SslCredentials(server_CA_Cert, clientCertPair, verifyPeerCallback => true);

// Create secure channel
var channel = new Channel(host, port, channelCredentials);
```

Call-level security (Jwt)

For clients, authentication can be specified via CallOptions from Grpc.Core. In addition, **protobuf-net.Grpc** unifies the CallOptions and ServerCallContext types into a single value-type **CallContext**. It is common to include an optional CallContext parameter on your methods for this purpose. The client can now provide this additional detail by passing in a CallContext/CallOptions that describe the need:

Mutual TLS - Client

```
// Method definition
Task<MultiplyResult> MultiplyAsync(MultiplyRequest request, CallContext context = default);

// Example of usage (see the prototypes)
CallOptions options = new CallOptions(new Metadata {{ "SomeHeader", "SomeHeaderValue" }}, // Add some http-
header
          null, // Deadline - how long client is willing to wait for a reply from the server.
          new CancellationTokenSource(TimeSpan.FromMinutes(1)).Token, // cancellation token if needed
          null,//new WriteOptions(WriteFlags.BufferHint | WriteFlags.NoCompress), // BufferHint allows grpc to
accumulate data into big chunks before sending
          null, // propogation token - another lesson to learn yet
          CallCredentials.FromInterceptor(AccessTokenInterceptor(accessToken)) // allows to inject the token to
the call context - authorization stuff
);
// Call with custom context
MultiplyResult result = await calculator.MultiplyAsync(new MultiplyRequest { X = 15, Y = 3 }, options );

// Example of token injection
public static AsyncAuthInterceptor AccessTokenInterceptor(string accessToken)
{
  return new AsyncAuthInterceptor((context, metadata) =>
  {
    metadata.Add("Authorization", "Bearer " + accessToken);
    return Task.CompletedTask;
  });
}
```

Interceptors

Part of native Grpc.Core library. There available client's and server's interceptors out-of-the-box. They can be useful for tracing/logging.

The server interceptors are applied to the service on the server. Multiple interceptors can be defined for every service on the server. In the opposite, on the client side the interceptors are applied on the whole channel.

For details see the prototype where the client's interceptor does JSON-formatting of the call it received.

Logging

Out-of-the-box Grpc.Core logging

Console logger

```
private static Grpc.Core.Logging.ILogger s_logger = new Grpc.Core.Logging.ConsoleLogger();
...
s_logger.Warning("Added header via Server interceptor");
```

By default, the library uses Console as a default output for internal messages. To enable an output for grpc-internal messages system Environment variables should be used:

gRPC environment variables

```
// native grpc cc-lib logging - uses console as a default output
Environment.SetEnvironmentVariable("GRPC_TRACE", "api");
Environment.SetEnvironmentVariable("GRPC_VERBOSITY", "debug");
```

See gRPC environment variables for more details.

The output can also be redirected to any object which implements `Grpc.Core.Logging.ILogger` interface. The output will contain grpc-internal messages too, if enabled.

In the following example a text file is used as an output:

Logging to a text file

```
private static ILogger s_logger= new Grpc.Core.Logging.TextWriterLogger(new StreamWriter("d:\\out.txt"));
...
// Register the logger in Grpc
Grpc.Core.GrpcEnvironment.SetLogger(s_logger);
...
s_logger.Warning("Added header via Server interceptor");
```

Notes (from MS-docs, however for ASP.Net Core)

1. gRPC leaves secure networking to the underlying HTTP/2 protocol, which you can secure by using TLS certificates. Web browsers insist on using TLS connections for HTTP/2, but most programmatic clients, including .NET's `HttpClient`, can use HTTP/2 over unencrypted connections. `HttpClient` does require encryption by default, but you can override this by using an AppContext switch. More details
2. When you're using gRPC over a TLS-encrypted HTTP/2 connection, all traffic between clients and servers is encrypted, even if you don't use channel-level authentication. More details

References

- gRPC-prototype with "code-first" approach **PR-30263** - Implement gRPC proxy prototypes IN PROGRESS