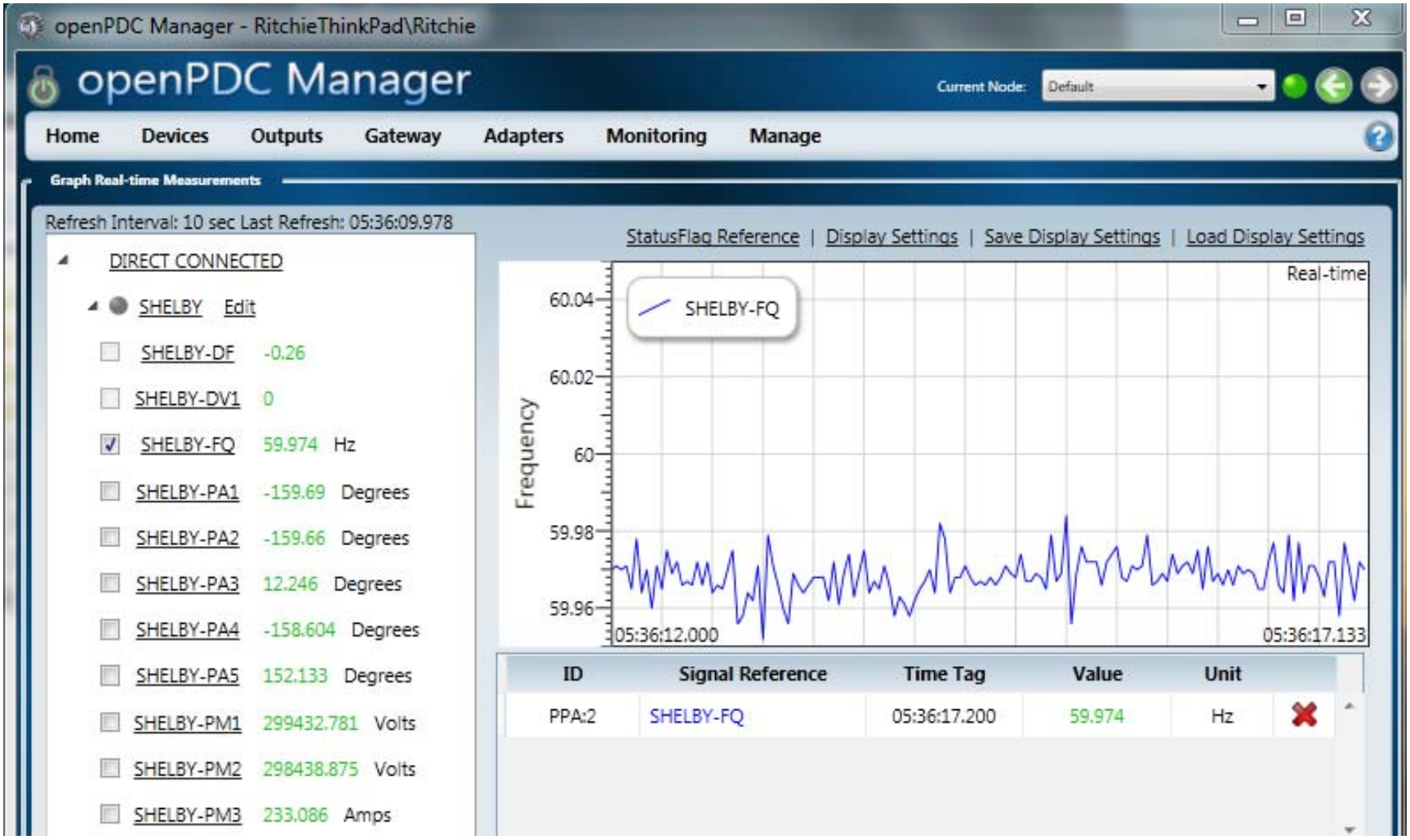
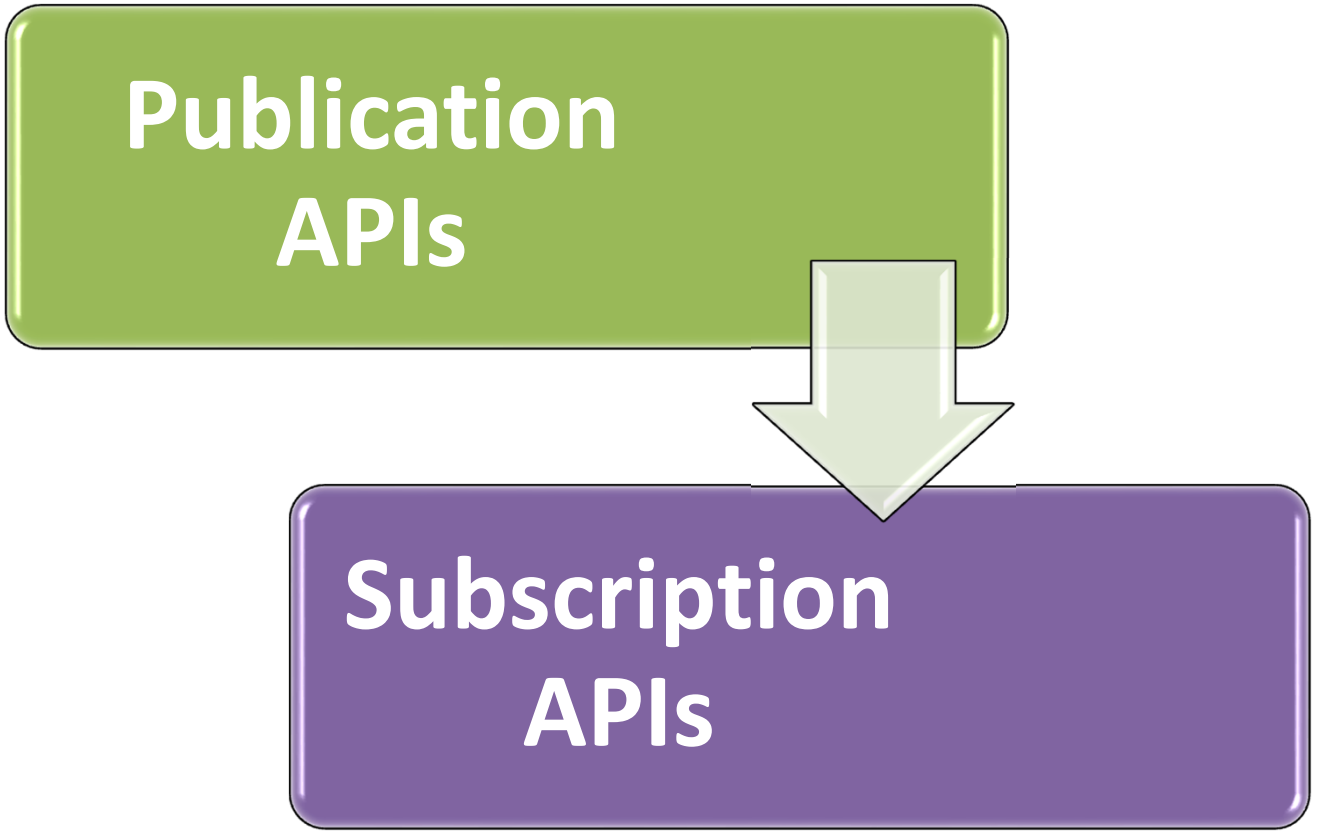
Using the Gateway Exchange Protocol

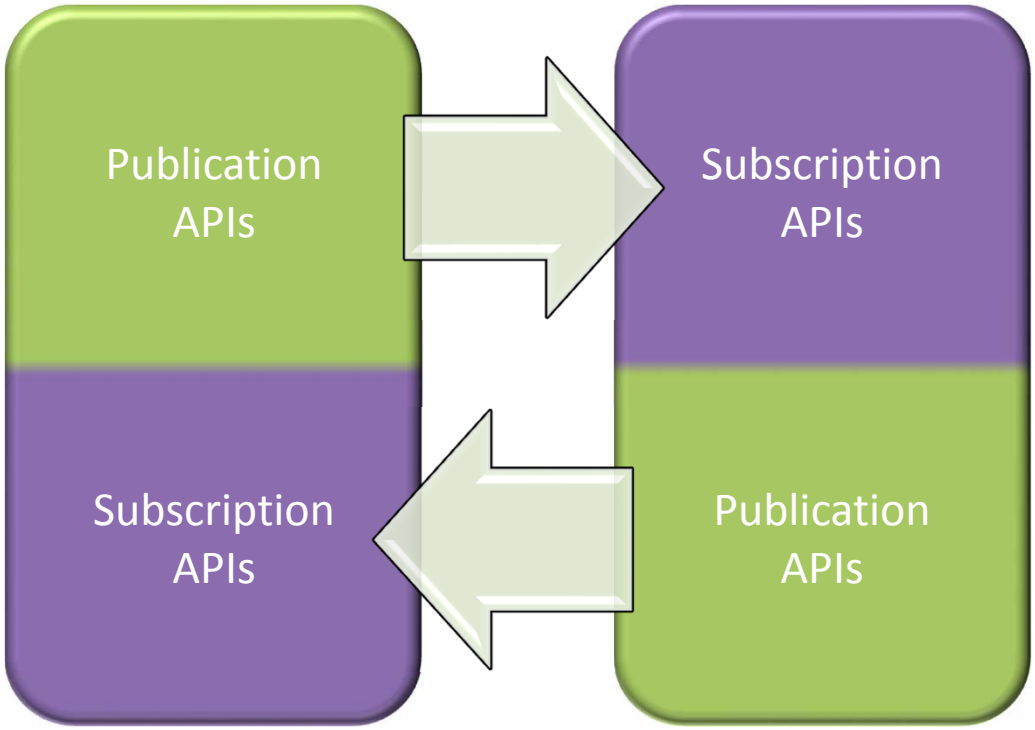
Subscribing to Measurements



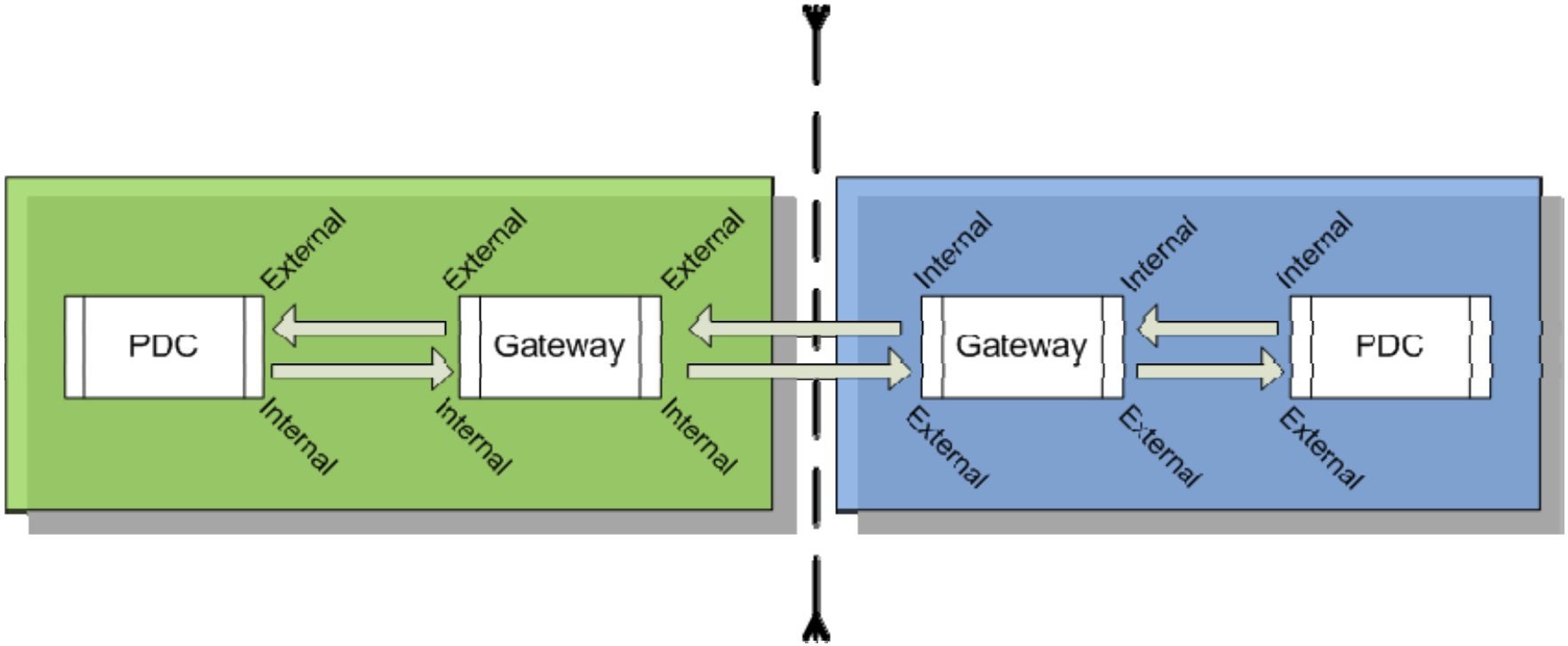
Primary Data Flow



Primary Data Flow (cont.)



Internal/External



Gateway Exchange Protocol (GEP)

- GEP is an extremely simple, small and fast wire format than can be used to exchange data points without a fixed predefined configuration – that is:
  - Points arriving in one data packet can be different than those arriving in another data packet. This can be due to each point having a different delivery schedule – or a dynamic schedule (e.g., alarms).
- GEP is a signal level publish/subscribe protocol with two available channels:
  - ***Command Channel (TCP)***
  - ***Data Channel (UDP or TCP)***

Synchrophasor Data Protocol Comparisons

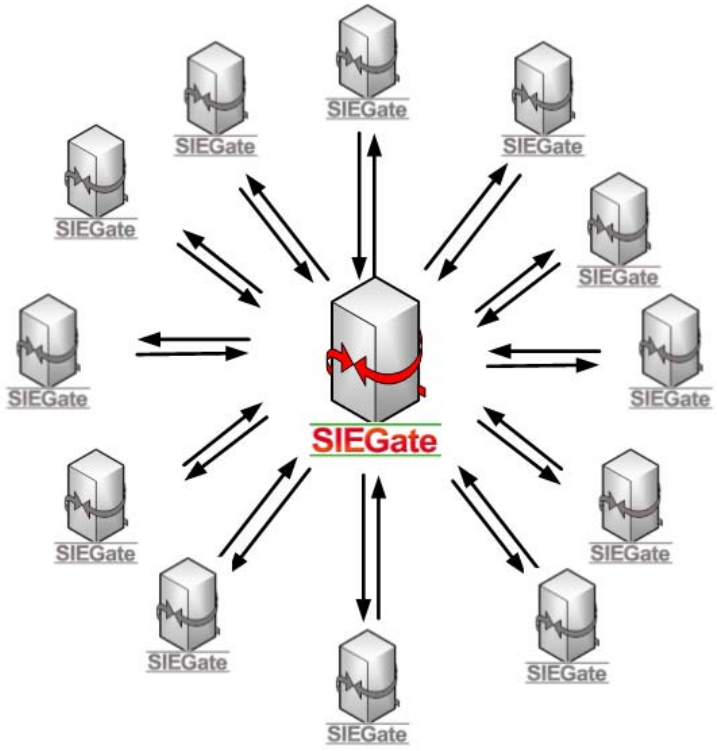
	IEEE C37.118	IEC 61850	GEP
Deployment Zones *Today*	Substation Control Center Inter-company	Substation Control Center	Control Center Inter-company
Preconfigured Data Packet Format	Yes	Yes – but client definable	No
Security Options	No	Yes	Yes
Signal Level Publish / Subscribe	No	Yes – but not dynamic	Yes

Example Interoperability Layers

Utility Layer	Example	Challenges
Inter-Reliability Coordinator	GEP	• High Volume at Low Latency • Dynamic Configuration
Inter-Operating Center	GEP IEEE C37.118	• Configuration Management
Control Center	GEP IEEE C37.118	• System Integration
Device / Substation	IEEE C37.118 IEC 61850	• Device interoperability • Device performance

What are the requirements?

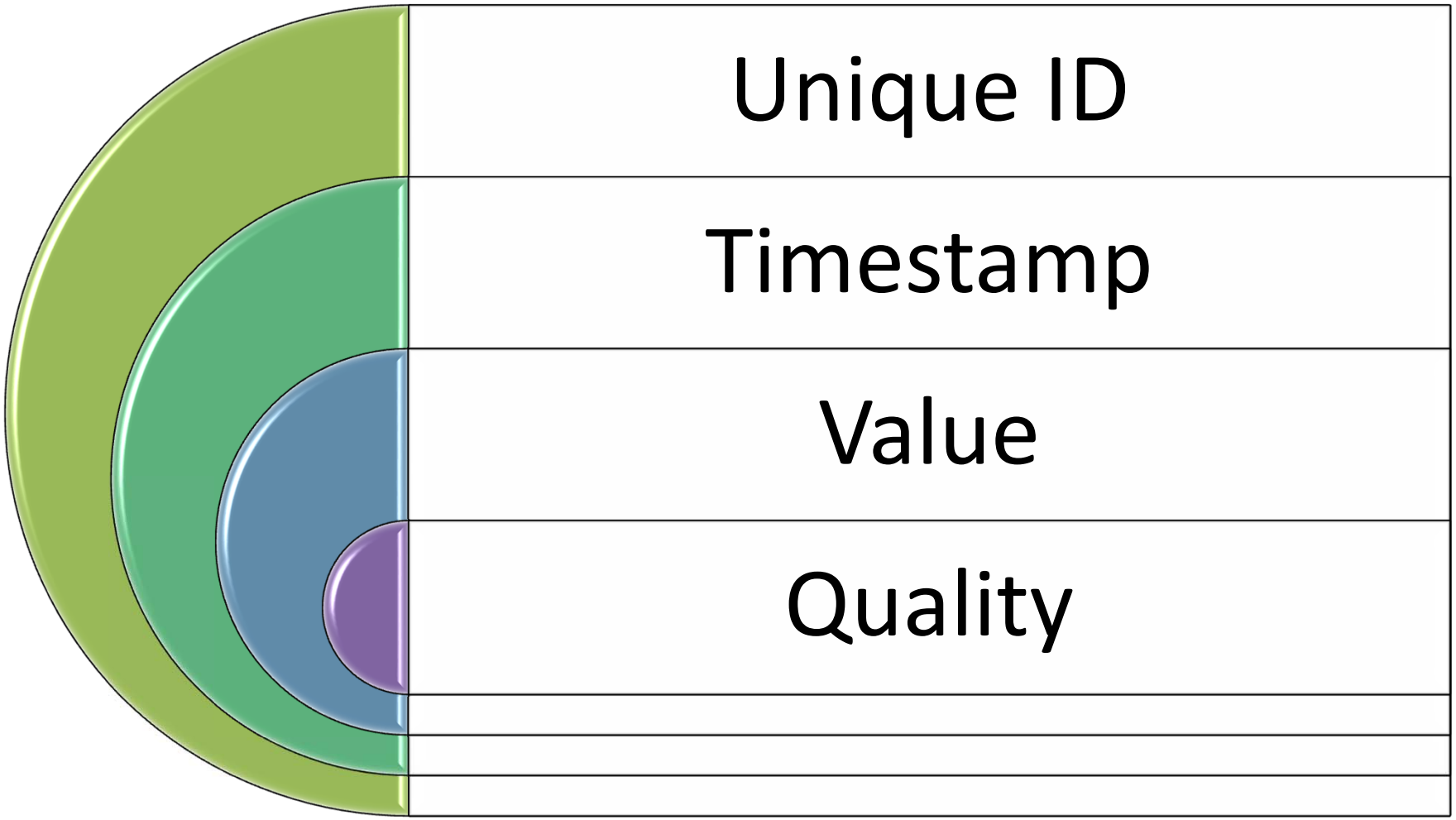
- GEP must move a *continually variable* set of points at low latency – to be successful, around 1 million points per second.



- 1 million assumes 12 associations and 100 PMUs (in and out)
  = ~ 0.5 M points in / sec
  ~ 0.5 M points out / sec

- As of SIEGate 1.0 and openPDC 2.0, over 3,350,000 measurements per second can be accommodated.

Moving Measurement Data



Simple Optimizable Structure

- Measurement data is well structured and can be safely condensed into a simple data structure (per signal):
  - 16-bit ID (established at connection)
  - Time (condensed where possible)
  - Value (32-bit real number)
  - Flags
- A highly effective lossless data compression is optionally enabled for the time-series data:
  - Implements an Xor based back-tracking compression algorithm to remove repeating bytes

Buffer Block

- Buffer block measurements define a block of data, rather than a simple measurement value
- GEP can transmit buffer blocks to transfer serialized data in chunks
- SIEGate uses buffer blocks for file-based transfers through GEP

Options for Connecting with GEP

- To get data “into” an application you can use GEP using a variety of API options:
  - C++
  - Java
  - .NET
  - Mono.NET
  - Unity 3D

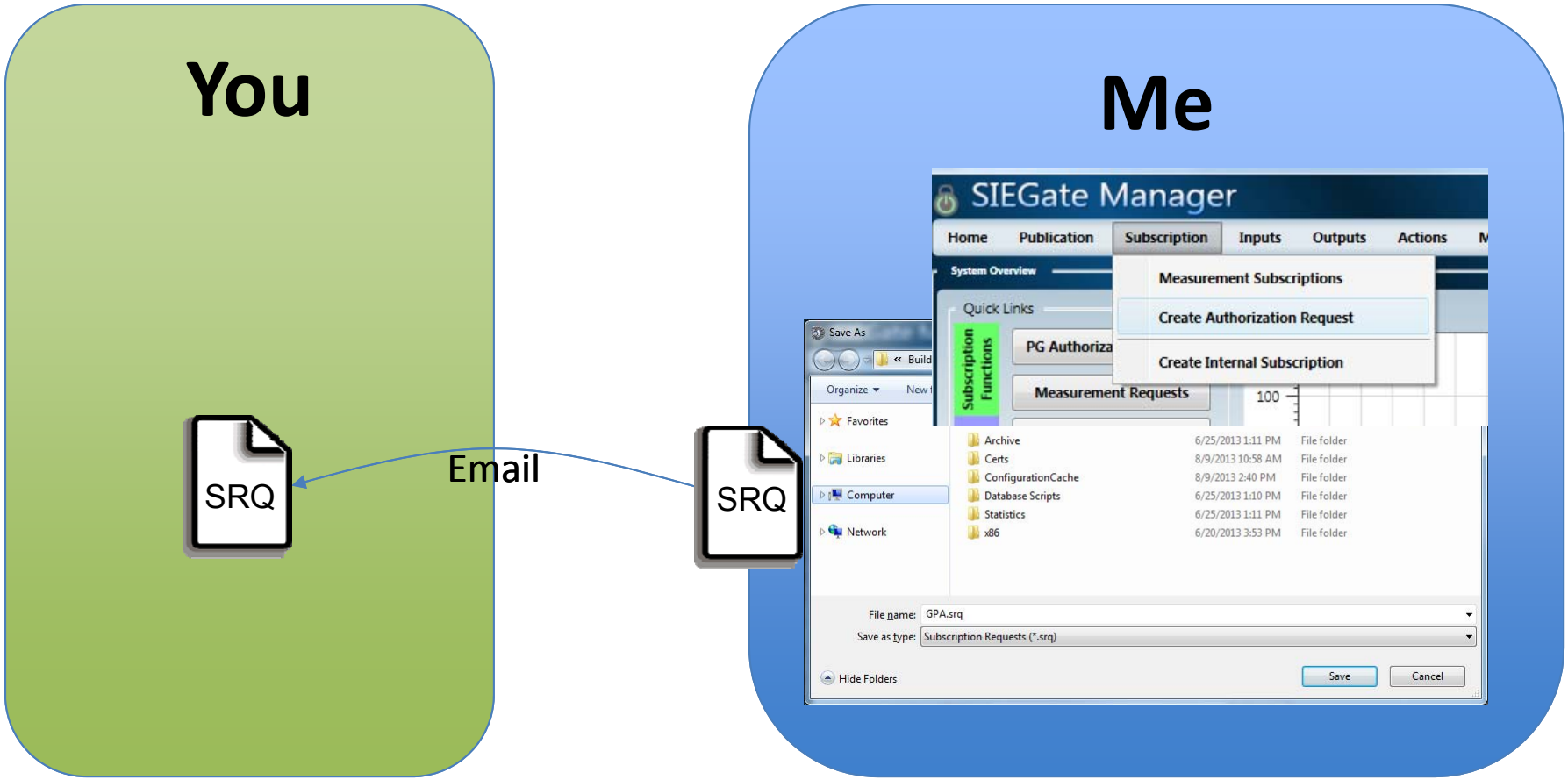
GEP Security Modes

- Transport Layer Security Mode
  - TCP command channel is secured using TLS – certificates exchanged out of band
  - Optional UDP data channel is secured using rotating keys exchanged over TLS command channel
  - Measurement access restricted on a per subscriber basis
- Gateway-to-Gateway Security Mode
  - TCP command channel is secured using symmetric AES encryption – keys exchanged out of band
  - Optional UDP data channel is secured using rotating keys exchanged over encrypted command channel
  - Measurement access restricted on a per subscriber basis
- Internal Access Mode (No Encryption)
  - Data transferred openly (ideal for internal connections or VPN transfers when connection is already encrypted)
  - Measurement access is unrestricted

Steps to Exchange Data

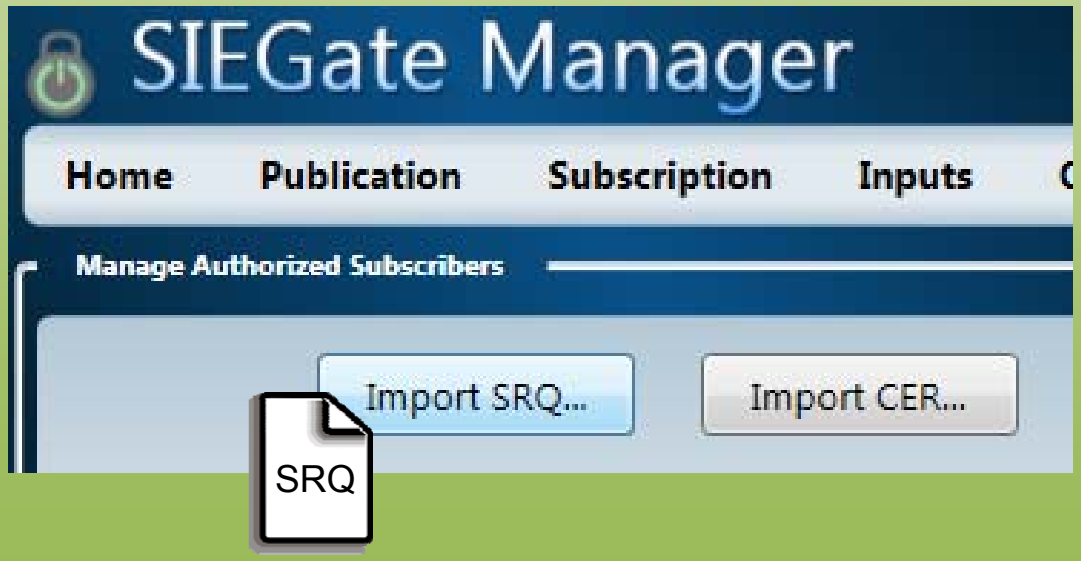
1. **Subscriber creates an authorization request**
   - Generates an SRQ file
   - Send the SRQ file out-of-band (email, thumb drive, CD, etc.)
2. **Publisher imports SRQ file**
   - Authorizes subscriber to connect, but still cannot subscribe
3. **Publisher authorizes subscriber to subscribe to measurements**
   - Publisher can control which measurements that subscriber can see
4. **Subscriber subscribes to measurements**
   - Subscriber can control which measurements that subscriber needs to see

Subscriber Creates an Authorization Request



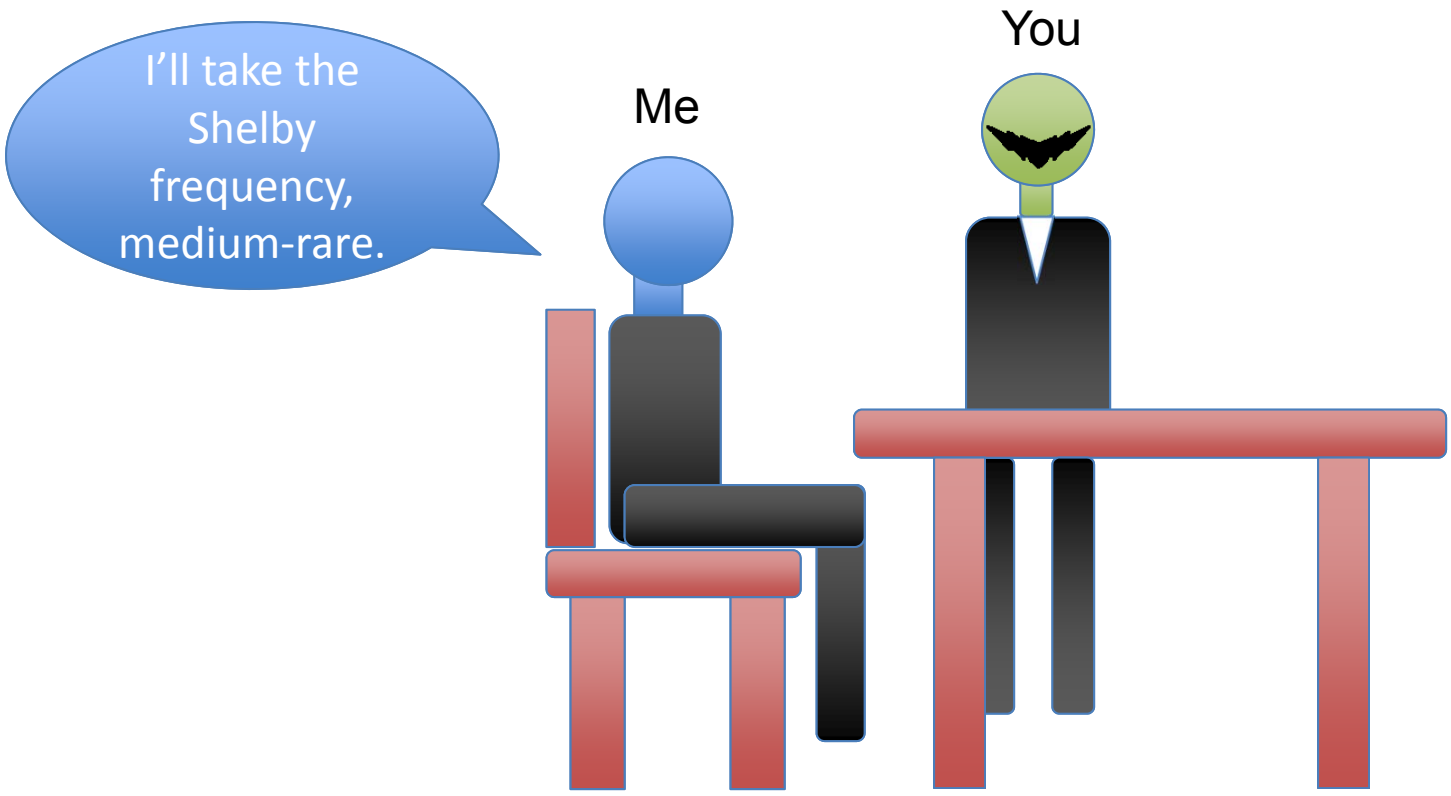
Publisher Imports SRQ File

You



Publisher Authorizes Subscriber to Subscribe to Measurements

Subscriber Subscribes to Measurements



DataPublisher API Usage

Purpose:
SEND

- **Attach to publisher events**
- **Initialize publisher**
- **Start publisher**
- **Queue new measurements for processing**

Example DataPublisher API Code

```
namespace DataPublisherTest
{
    class Program
    {
        static DataPublisher publisher = new DataPublisher();
        static Ticks lastDisplayTime;
        static object displayLock = new object();

        static void Main(string[] args)
        {
            // Attach to publisher events
            publisher.StatusMessage += publisher_StatusMessage;
            publisher.ProcessException += publisher_ProcessException;
            publisher.ClientConnected += publisher_ClientConnected;

            // Initialize publisher
            publisher.Name = "dataPublisher";
            publisher.UseBaseTimeOffsets = true;
            publisher.Initialize();

            // Start publisher
            publisher.Start();

            ThreadPool.QueueUserWorkItem(ProcessMeasurements);
```

DataSubscriber API Usage

Purpose:
Receive

- **Attach to subscriber events**
- **Set up subscription info objects**
- **Initialize subscriber**
- **Start subscriber connection cycle**
- **Handle new measurement data**

Example DataSubscriber API Code

```
static void Main(string[] args)
{
    if (args.Length < 2)
    {
        Console.Error.WriteLine("Error: requires two command line arguments");
        Console.Error.WriteLine("       1. hostname of publisher");
        Console.Error.WriteLine("       2. port used to initiate connection");
        return;
    }

    // Set up subscription info object
    unsynchronizedInfo.FilterExpression = "FILTER ActiveMeasurements WHERE SignalID LIKE '%'";

    // Attach to subscriber events
    subscriber.StatusMessage += subscriber_StatusMessage;
    subscriber.ProcessException += subscriber_ProcessException;
    subscriber.ConnectionEstablished += subscriber_ConnectionEstablished;
    subscriber.ConnectionTerminated += subscriber_ConnectionTerminated;
    subscriber.NewMeasurements += subscriber_NewMeasurements;

    // Initialize subscriber
    subscriber.OperationalModes |= OperationalModes.UseCommonSerializationFormat |
        OperationalModes.CompressMetadata |
        OperationalModes.CompressSignalIndexCache |
        OperationalModes.CompressPayloadData;

    subscriber.ConnectionString = string.Format("server={0}:{1}", args[0], args[1]);
    subscriber.Initialize();

    // Start subscriber connection cycle
    subscriber.Start();
```

Live Demos

- *Subscribing from a .NET C# application*
- *Subscribing from a C++ application (Linux)*
- *Subscribing from a Java application*
- *Subscribing from the Unity platform*