Why ZeroMQ?
About ZeroMQ

• The “zero” in ZeroMQ
  ▪ Zero Broker
  ▪ Zero Latency (Low Latency)
  ▪ Zero Administration
  ▪ Zero Cost – Cross Platform & Open Source

• Allows complex messaging exchange patterns with minimal effort

• Scalable for distributed or concurrent applications
ZeroMQ Benefits

- Numerous language and platform integration points – all integrated and compatible
- Small and light-weight with the performance to support high-volume phasor data flows
- Larger variety of message patterns with a range of loss/reliability characteristics – pub/sub; client/server; brokered.
- Content of the message flexible and easily accommodates phasor measurement pattern – ID, Timestamp, value, flags
- In practice, ZeroMQ is used to manage the socket layer on behalf of the application
- Ability to scale well is inherent in architecture -- scales easily from intra-application communication, to inter-application communication to wide-area communication
ZeroMQ vs. DDS

• DDS
  ▪ Pros: Mature "middle-ware" layer supporting mission critical apps, extensive number of options
  ▪ Cons: Heavy-weight, slower, steep learning curve, no open source standards based security yet

• ZeroMQ:
  ▪ Pros: Many messaging patterns, extensive language implementations, fully open source with security, light-weight, faster
  ▪ Cons: Lower level API, not as many features as DDS for options like discovery, delivery deadlines and QoS
Summary of CERN* Evaluated Middleware

<table>
<thead>
<tr>
<th></th>
<th>patterns</th>
<th>QoS</th>
<th>resources</th>
<th>performance</th>
<th>user friendly</th>
<th>community</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA</td>
<td>x</td>
<td>v</td>
<td>x</td>
<td>v</td>
<td>x</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>Ice</td>
<td>v</td>
<td>v</td>
<td>x</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>5</td>
</tr>
<tr>
<td>Thrift</td>
<td>x</td>
<td>x</td>
<td>v</td>
<td>v</td>
<td>x</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>ZeroMQ</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>6</td>
</tr>
<tr>
<td>YAMI4</td>
<td>v</td>
<td>v</td>
<td>x</td>
<td>v</td>
<td>v</td>
<td>x</td>
<td>4</td>
</tr>
<tr>
<td>RTI</td>
<td>x</td>
<td>v</td>
<td>x</td>
<td>v</td>
<td>x</td>
<td>v</td>
<td>3</td>
</tr>
<tr>
<td>Qpid</td>
<td>x</td>
<td>v</td>
<td>x</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 3: Summary of evaluated middleware products.

Figure 2: Test 3, pub-sub to a C++ server.

Message Patterns

- pub/sub
- queue
- pipeline
- async
- gateway

PUB / SUB

ROUTER / DEALER
The Coding Pattern

```
import zmq
import random

context = zmq.Context()
socket = context.socket(zmq.PUB)
socket.bind("tcp://*:5556")

while True:
    zipcode = random.randrange(1,100000)
    temperature = random.randrange(1,215) - 80
    relhumidity = random.randrange(1,150) + 10
    socket.send("%d %d %d %d %d" % (zipcode, temperature, relhumidity))
```

```
import sys
import zmq

# Socket to talk to server
context = zmq.Context()
socket = context.socket(zmq.SUB)
socket.setsockopt(zmq.SUBSCRIBE, b")")

print "Collecting updates from weather server...",
socket.connect("tcp://localhost:5556")

while True:
    string = socket.recv()  # Subscribe in zipcode, default is NYC, 10001
    zipcode, temperature, relhumidity = string.split()
    print "Average temperature for zipcode '%s' was %s F" %
```
GSF ZeroMQ Implementation

- Implemented the ROUTER – DEALER ZeroMQ pattern as a standard client / server streaming data transfer implementation.
- Allows for all support ZeroMQ transport protocols
  - TCP
  - In-Process (e.g., named pipes)
  - Pragmatic General Multicast (PGM)
  - Encapsulated PGM
ZeroMQ Example Code (from GSF)

Setup ZeroMQ:

```java
m_zeroMQServer = new ZSocket(ZContext.Create(), ZSocketType ROUTER);
m_zeroMQServer.Identity = ServerID.ToArray();
m_zeroMQServer.SendHighWatermark = m_maxSendQueueSize;
m_zeroMQServer.ReceiveHighWatermark = m_maxReceiveQueueSize;
m_zeroMQServer.Immediate = true;
m_zeroMQServer.SetOption(ZSocketOption.LINGER, 0);
m_zeroMQServer.SetOption(ZSocketOption.SNDTIMEO, 1000);
m_zeroMQServer.SetOption(ZSocketOption.RCVTIMEO, -1);
m_zeroMQServer.SetOption(ZSocketOption.RECONNECT_IVL, -1);
m_zeroMQServer.IPv6 = (Transport.GetDefaultIPStack() == IPStack.IPv6);
m_zeroMQServer.Bind(m_configData["server"]);
```

Route data to client:

```java
// Lookup client info, adding it if it doesn't exist
TransportProvider<dateTime> clientInfo = GetClient(clientID);

// Router socket should provide identity, delimiter and data payload frames
using (ZMessage message = new ZMessage())
{
    // Add identity, delimiter and data payload frames
    message.Add(new ZFrame(clientID.ToArray()));
    message.Add(new ZFrame());
    message.Add(new ZFrame(data, offset, length));

    // ZeroMQ send is asynchronous, but API call is not thread-safe
    lock (m_sendLock)
    {
        m_zeroMQServer.Send(message);
    }
}
ZeroMQ Observations

- Excellent for distribution of simple messages
- Also supports stateful-style message protocol operations, but can require extra work
- Patterns and classes exist to make multi-threading with ZeroMQ simple, but API itself is not thread-safe
- Security, called CURVE, is now baked-in and is easy to “turn on”
- Low-level library is surprisingly fast and can have performance benefits over other socket implementations on IoT style hardware.
ZeroMQ Threading Patterns

• Calls into the ZeroMQ API library are not inherently thread-safe
• ZeroMQ uses patterns for simplifying multithreading

ØMQ for Multithreading

Don't use locks, semaphores, mutexes
Design app as message-driven tasks
Each task reads from 1..n sockets
Tasks can talk over inproc://
Tasks can be split into processes over tcp://
No wait states, no locks, full CPU use
Scalable to any number of cores
ZeroMQ Demo

- Demonstrate GEP operation over ZeroMQ transport using high-speed, high-bandwidth message distribution.
- Demo will use 2-channel WAV based inputs distributed to multiple clients.