Design of a Stream Based Software Radar Architecture

Karl Cronburg (Bucknell University), Mentors: Frank Lind & Robert Schaefer (MIT Haystack)

Goals:

- Stream data directly from instruments to signal processing and analysis systems
- Dynamic scaling of computation and storage resources
- Event-based control of processing and analysis systems
- Plug-and-play message distribution
- Automatic system configuration for various tasks
- Robust data / object formats for:
  - Radar frequency voltages
  - Metadata
  - Status, logging, and debugging events

Tools:

- Redhat Enterprise Linux Operating Environment
- Python Implementation
- GNU C Based Instruments
- ZeroMQ Sockets Library (python binding)
- Gevent – deterministic python event-based threading
- YAML (Yet Another Markup Language)
- MessagePack Object Serialization
- PyCUDA GPU Acceleration
- Numpy & Scipy Scientific Analysis Tools
- Matplotlib Plotting
- HDF5 Data Storage

This Project:

The primary objective of this project is to collect, analyze, and process Radar data with live streaming and viewing capabilities using existing and future processing and analysis components.

The work I undertook towards this goal involved setting up the architecture for defining data objects apart from Radar processing implementations, and further serializing these objects for transport over networks.

A similar architecture being designed is **Ocean Observatories Initiative (OOI)** Cyberinfrastructure, for dynamically collecting and distributing Ocean-related data.

YAML Object Definition:

This project primarily focused on the design and implementation of a YAML-based Interface Definition Language (IDL) used for defining composite objects with named attributes.

YAML object definition provides…

- A human readable data format
  - Easily listen to / debug data streams
  - Non-programmer understandable object composition
  - Implementation agnostic object definitions
  - Use any YAML-aware programming language
- A highly capable yet robust object format
  - Capable of default attribute values
  - Handles all basic data types (strings, floats, ints, etc)
  - Numpy aware
  - Automatic construction of composite data types

```yaml
RFVoltageData:
    rf_signal_id: !!str ''
    rf_frequency: !dtype.float128 0.0
    start_time: !timestamp_picosecond
    stop_time: !timestamp_picosecond
    sample_period: !dtype.int64 0
    iq_data: !dtype.int16 {default: 0, len:[]}

RFSource:
    site: !!str ''
    site_tag: !!str ''
    latitude: !dtype.float64 0.0
    longitude: !dtype.float64 0.0
    altitude: !dtype.float64 0.0
    antenna: !!str ''
    antenna_tag: !!str ''
    antenna_type: !!str ''
    antenna_azimuth: !dtype.float64 0.0
    antenna_elevation: !dtype.float64 0.0
    antenna_polarization: !!str ''

RFSignal:
    source: !ref RFSource
    voltage_data: !ref RFVoltageData
```

Figure 1 – Example object definition for RF voltage data and corresponding metadata.

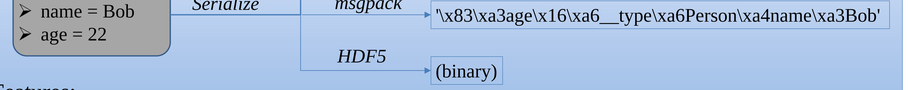
```python
# Load YAML definition & create RFSignal instance:
import numpy as np
from DynamicObject import load_defn
RFVoltageData, RFSource, RFSignal = load_defn('RFSignal.yml')
voltage_data = RFVoltageData('signal_id', np.float128(100), ...)
my_source = RFSource('Millstone', ...)
my_signal = RFSignal(my_source, voltage_data)

# Serialize using various formats & compression:
from Serializer import YAML, MsgPack, HDF5
yaml = YAML().dumps(my_signal)
msgpack = MsgPack().dumps(my_signal)
hdf5 = HDF5().dumps(my_signal, 'gzip')

# Transmit an RFSignal instance:
from Transport import Sender
sender = Sender('push', 5000, 'myStream',
                'md5', 'msgpack', 'gzip')
sender.send(my_signal)

# Receive RFSignal instance with on-the-fly deserialization:
from Transport import Receiver
recvr = Receiver('pull', 'localhost', 5000, 'myStream')
my_signal = recvr.recv()
```

Figure 2 – Example usage of the dynamic object instantiation, serialization, and streaming capabilities of various python modules written for this project.

Object Serialization:

Formats:

- YAML – slow but human-readable serialization for message stream debugging
- MessagePack – fast & compact serialization for on-the-wire data transmission
- HDF5 – object storage in a consistent scientific data format



Features:

- On-the-fly deserialization – dynamic loading of object definitions from revision control
- 'Pickle' like syntax for loading and dumping objects
- Optional gzip compression

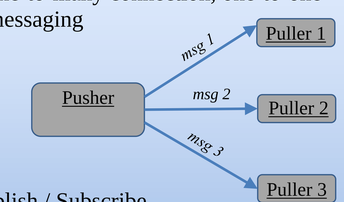
Example serialization usage shown in Fig. 2.

ZeroMQ Object Transport:

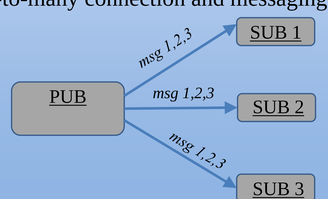
The ZeroMQ sockets library provides elastic / scalable plug-and-play style messaging.

Messaging Patterns:

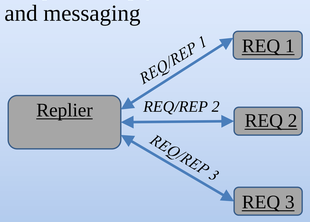
- Push / Pull
  - one-to-many connection, one-to-one messaging



- Publish / Subscribe
  - one-to-many connection and messaging



- Request / Reply – one-to-one connection and messaging



Features Implemented:

- Automatic object (de)serialization
- Debugging / re-routing message filters
- Optional hash-based integrity verification
- Clean messaging interface (see Fig. 2)

Runtime Results:

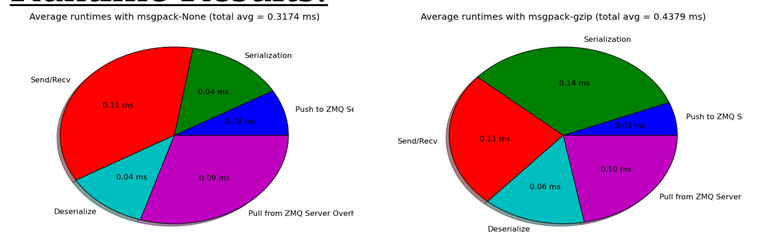
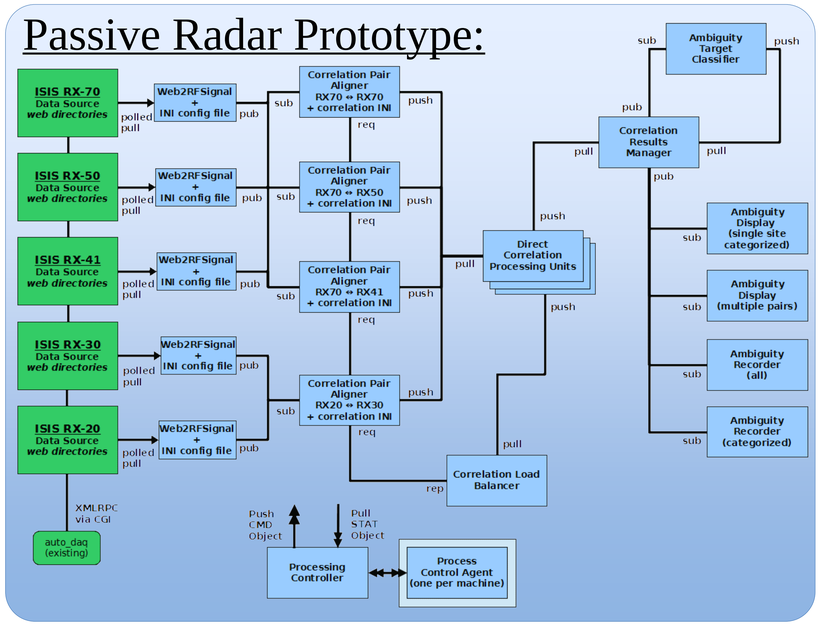


Figure 3 – Average message rate using MessagePack serialization (with and without compression), broken up by task. Small (126 – 512 byte messages) were used.

Passive Radar Prototype:



Thanks to

- Frank Lind for giving clear specifications and assistance
- Robert Schaefer for implementation guidance
- Phil Erickson, Vincent Fish, and KT Paul for organizing the REU
- The MIT Haystack staff for support throughout the summer