xFOCE Data Lexicon

A **Stream** is a set of **Records** that are logically related (by some combination of time, producer, type, value, purpose etc.)

A **Record** is a single unit of **Data** and some relevant **Metadata.** A record may contain **composite** or **simple** Data. A record may contain **original** or **derivative** Data.

A **Timestamp** is metadata the indicates the time at which its associated data originated.

Data is information observed or archived for a specific purpose. Data may be **composite** (containing multiple units of information) or **simple** (containing a single unit of information). The smallest unit of Data is also referred to as an **Observation**.

Original Data: Data that retains the representation and/or purpose established by the DataProducer.

Derivative Data: A modified version of Data, for example a subset or transformation.

Metadata is contextual Data with the primary purpose of manipulating and/or interpreting other Data

A **DataType** is a unique representation and purpose that serves to distinguish observations.

A **DataProducer** is the entity at which an Observation originates.

A **DataConsumer** is an entity that requests or uses one or more Observations.

An **Archive** is a persistent representation of a Stream. The primary key of an archive is assumed to be timestamp.

An Archive may be:
heterogenous: contains multiple DataTypes
homogeneous: contains a single DataType
hybrid: archived in domain related bins (science, diagnostics)
monolithic: uses a single file
segmented: uses multiple files

Archive Organization

What is best way to organize and access the primary archive?

Assumptions:
Archives are organized primarily by producer/device

Archives are indexed chronologically

Additionally, Archives may be:
heterogenous: contains multiple DataTypes
homogeneous: contains a single DataType
monolithic: uses a single file
segmented: uses multiple files

Archive Views
Archives are accessed via Streams, which are typically produced by applying one or more filters to the (chronological) records in an archive:

- Timestamp range
- Data type(s)
- Data value range (<,<>,>, ==, <=,>=)

Heterogeneous

Advantages
Represents order of events
Single data stream per producer

Disadvantages
More complicated to parse
Additional metadata needed to identify data type
mixes domain and diagnostic data

Homogeneous

Advantages
Single data type in one file is easy to understand and parse

Disadvantages
More difficult to present chronology

Hybrid

Advantages
Helps separate domain data from diagnostic data (e.g.)

Disadvantages
More difficult to present chronology

Monolithic

Advantages
All data in one place
Fewer files to manage

Disadvantages
Loss or corruption of file affects entire archive

Segmented (by size or time)

Advantages
Single file failure doesn't result in archive failure.

Easier to transfer files over low bandwidth links

Faster r/w access for smaller files

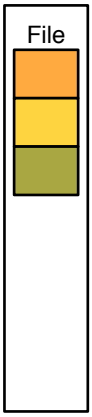
Segmenting can help make it easier to find a specific piece of data

Monolithic is special case (segment size unlimited); can implement in phases.

Disadvantages
Access tools must span multiple files.

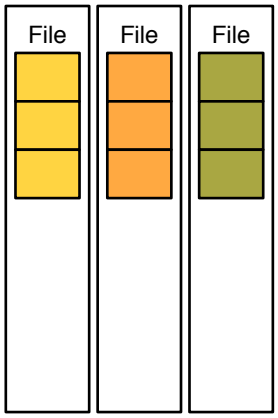
More files to manage

Heterogeneous, Monolithic

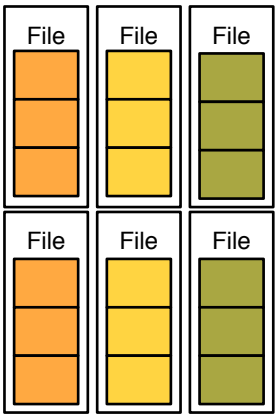


Heterogeneous, Segmented

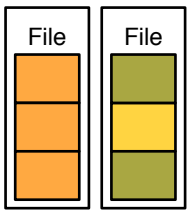
Homogeneous, Monolithic



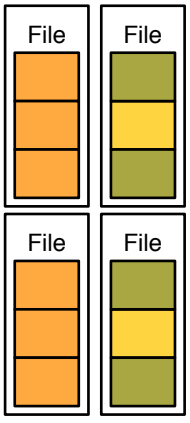
Homogeneous, Segmented



Hybrid, Monlithic



Hybrid, Segmented



Archive Structure

Master Index File

Indexes span of records in archive files

Master Index File

- headerSize[uint16]
- platformID[uint16]
- sourceID[uint16]
- checksum[uint32]
- minKey[uint64]
- maxKey[uint64]
- recordIndex[byte[256]]
- delimiter[byte]

Minimize archive volume
- Record only original data bytes
- Derivative (parsed, modified) data streams are generated downstream)
- where possible, do not duplicate metadata, use compact references, (parsing schemas, etc)

Science User Comments:

- Useful to keep data in files by producer/device (b/c easier to apply calibrations, etc. before concatenation in work flow)
- Useful to include human-sensible key(s) and maps that enable sorting streams (e.g. "node1", "pH1") w/o remembering numeric IDs
- Useful to be able to access a small slice of an archive, and be able to request/transfer small data subsets over network.
- Segmented data files are not an issue and would help with transfer and searching

Record Index File

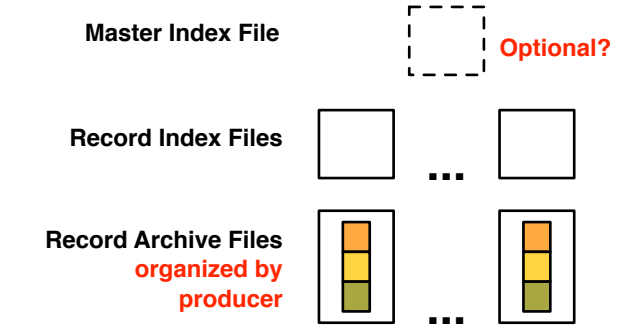
Indexes location of records in archive file

Record Index File

- headerSize[uint16]
- platformID[uint16]
- streamID[uint16]
- recordArchive[byte[256]]
- minKey[uint64]
- maxKey[uint64]
- checksum[uint32]
- key[uint64]
- offset[uint64]
- size[uint64]
- delimiter[byte]

invalid, missing...

- flags[byte]



Archive Record Format

Record fields

Record Archive File

- platformID[uint16]
- streamID[uint16]
- timestamp[uint64]
- sequenceNumber[uint64]
- sourceID[uint16]
- typeID[uint16]
- size[uint64]
- contents[byte[size]]
- delimiter[byte]

Could use Serialization Formats to describe Records (or index file formats) to clients in a platform-neutral way... include in header?

Serialization Format Schema

- headerSize[uint16]
- fieldCount[uint16]
- endianess[byte]
- checksum[uint32]
- fieldType[uint16]
- fieldSize[uint16]
- fieldNameSize[uint16]
- fieldName[fieldNameSize]

Legend

- fixed length (header)
- fixed length (recurring)
- variable length

Message Pattern Use Cases

Message Passing

Message passing enables data streaming and coordination of services, e.g. between device services and experiement control.

A message passing interface between Gateway nodes and external systems, makes it possible to implement clients in different languages.

Core Messaging Patterns

There are several messaging patterns needed:

Request-reply (R/R)
Connects a set of clients to a set of services. This is a remote procedure call and task distribution pattern.

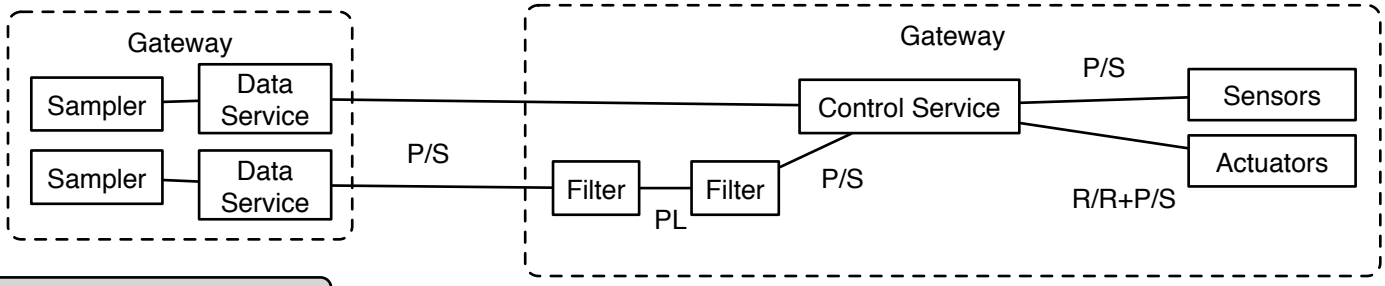
Publish-subscribe (P/S)
Connects a set of publishers to a set of subscribers. This is a data distribution pattern.

Pipeline (PL)
Connects nodes in a fan-out / fan-in pattern that can have multiple steps, and loops. This is a parallel task distribution and collection pattern. (uses Push/Pull - P/P)

Exclusive Pair (EP)
Connects a peer to precisely one other peer. This pattern is used for inter-thread communication across the inproc transport.

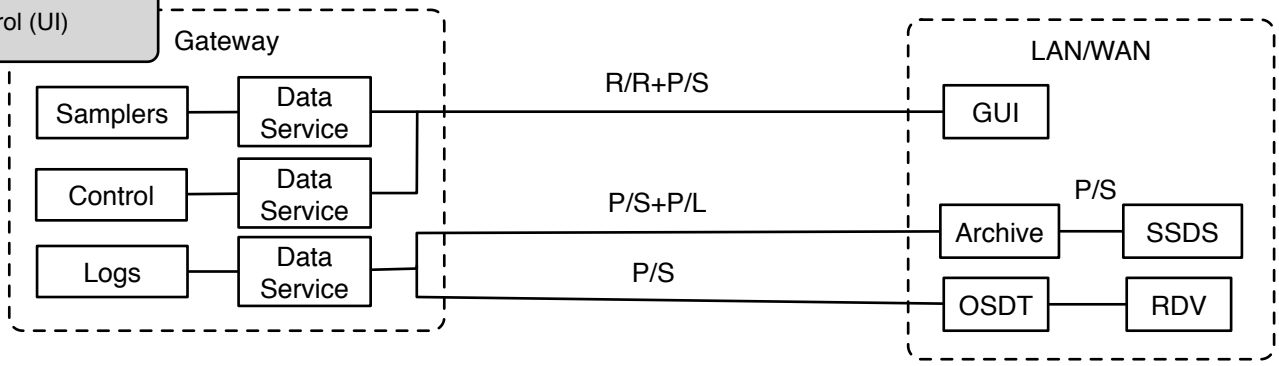
Messaging Use Cases

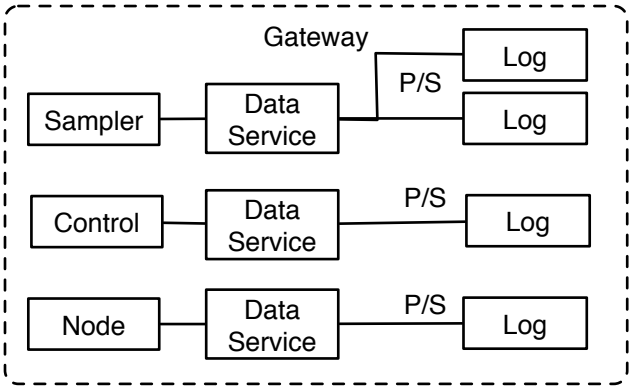
Control System data input
Service Coordination
Control system filtering (pipeline)



Messaging Use Cases

Archive Access/Data Transfer
Monitoring (UI)
Command/Control (UI)





Messaging Use Cases

Logging
(or better to log w/o using messaging)

Messaging Use Cases

Device Access
(serial port/device as service)?...

ZeroMQ Message Passing

ZeroMQ is a set of blocks that enable simple(r) construction of messaging patterns

ZeroMQ documents and provides examples of many messaging constructs and patterns

- Multi-part messages
- Message envelopes
- Intermediaries/Proxies/Broker/Bridge
- Multithreading
- Synchronization
- Relay, Extended Reply Envelope (Router/Dealer)
- Node Coordination (Pub/Rep,Sub/Req)

In essence, ZeroMQ replaces traditional point-to-point TCP/IP sockets and pipes with several flavors of N:N sockets that are optimized for high throughput and reliable transport.

ZeroMQ Socket Combinations

REQ/REP
PUB/SUB
DEALER/ROUTER
PUSH/PULL
PAIR
XPUB/XSUB

HIgh Level Patterns for reliability

The Lazy Pirate pattern: reliable request reply from the client side
The Simple Pirate pattern: reliable request-reply using load-balancing
The Paranoid Pirate pattern: reliable request-reply with heartbeating
The Majordomo pattern: service-oriented reliable queuing
The Titanic pattern: disk-based / disconnected reliable queuing
The Binary Star pattern: primary-backup server fail-over
The Freelance pattern: brokerless reliable request-reply

ZeroMQ...

Is brokerless
Does not manage persistence of messages
Lighter weight than AMQP
Heavier weight (more capable) than LCM
Examples are easy to use
Has many language bindings (including our favorites)
Supports many transports
Data format neutral

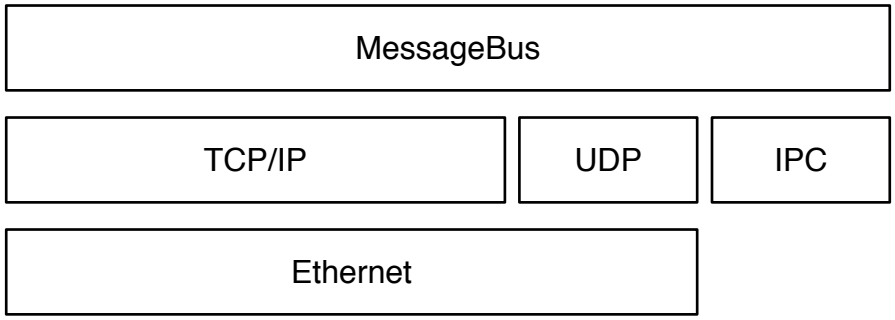
Other candidates

AMQP
- uses a broker
- persistence support
- heavy weight

LCM
- Primarily UDP
- Less flexible
- More difficult to build higher level patterns

Potential data serialization formats

MessagePack (JSON compatible, more efficient)
JSON (does not support comments)
BSON (superset of JSON)
Google protocol buffers (Java/C++ only)



Discovering Data Streams

Data streams use well known names on well known ports

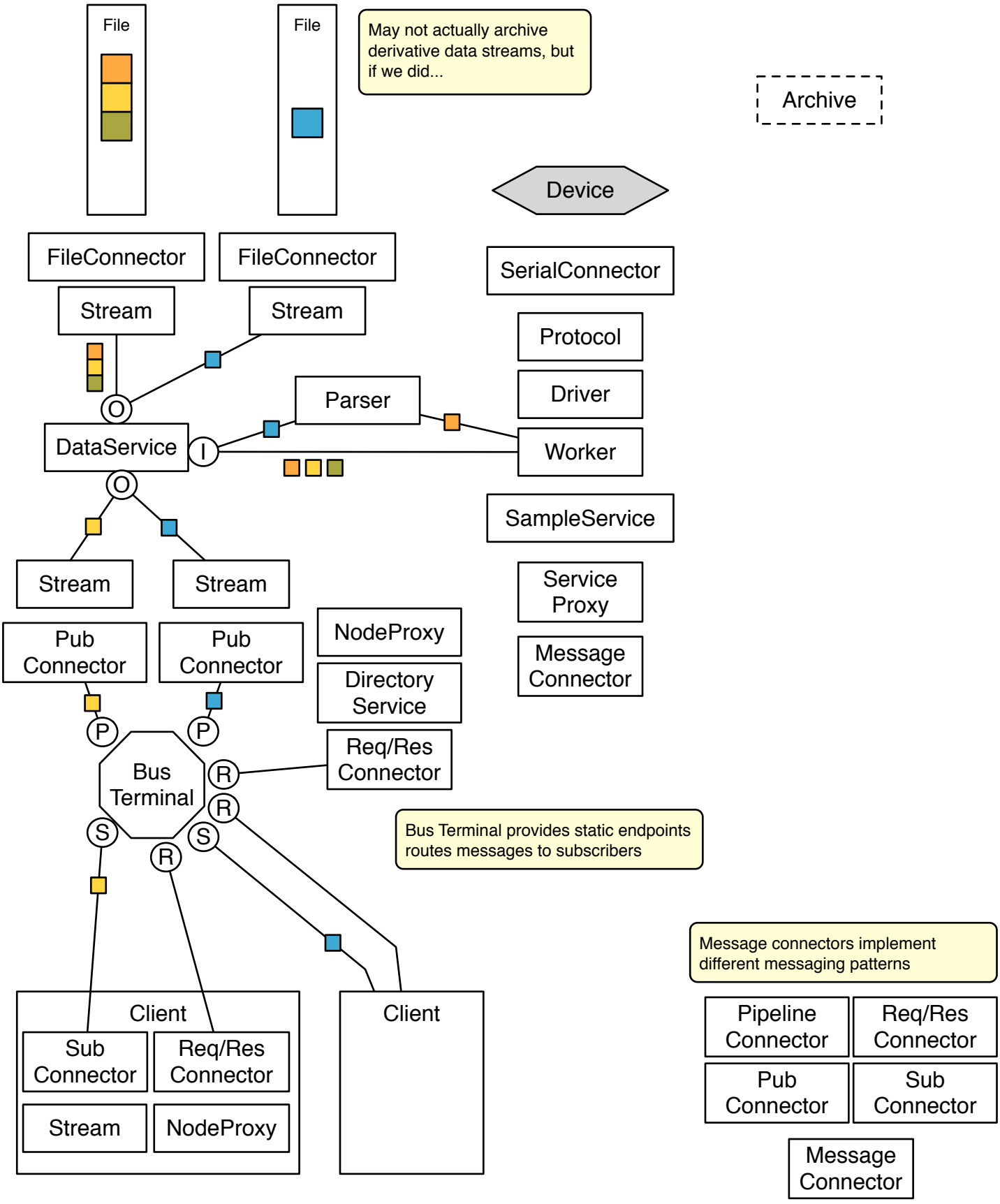
Discovering Data Streams

Data streams use well known names
Directory Service on well known port helps clients find streams

ZeroConf?

Discovering Data Streams

Dynamic using ZeroConf?


File
File
May not actually archive derivative data streams, but if we did...
Archive
Device
FileConnector
FileConnector
SerialConnector
Stream
Stream
Protocol
Parser
Driver
O
DataService
I
Worker
O
SampleService
Stream
Stream
Service Proxy
Pub Connector
Pub Connector
NodeProxy
Message Connector
Directory Service
P
P
Req/Res Connector
Bus Terminal
R
R
S
S
R
Bus Terminal provides static endpoints routes messages to subscribers
Message connectors implement different messaging patterns
Client
Client
Pipeline Connector
Req/Res Connector
Sub Connector
Req/Res Connector
Pub Connector
Sub Connector
Stream
NodeProxy
Message Connector

Stream Semantics

Stream Sources

- fresh: accessed as it is produced
- canned: accessed from persistent storage
- real time
- synchronous
- asynchronous
- sequential
- random access

Stream Access

- pub/sub
- push/pull
- broadcast
- multicast

Stream Message Types

Data
Notification
Acknowledgment
Request

AsynchronousData
SynchronousData
SubscriberData

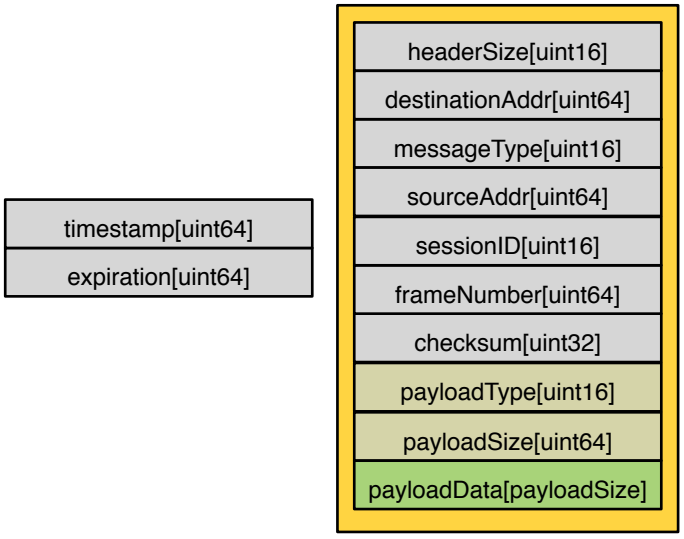
Warning
Info
Error

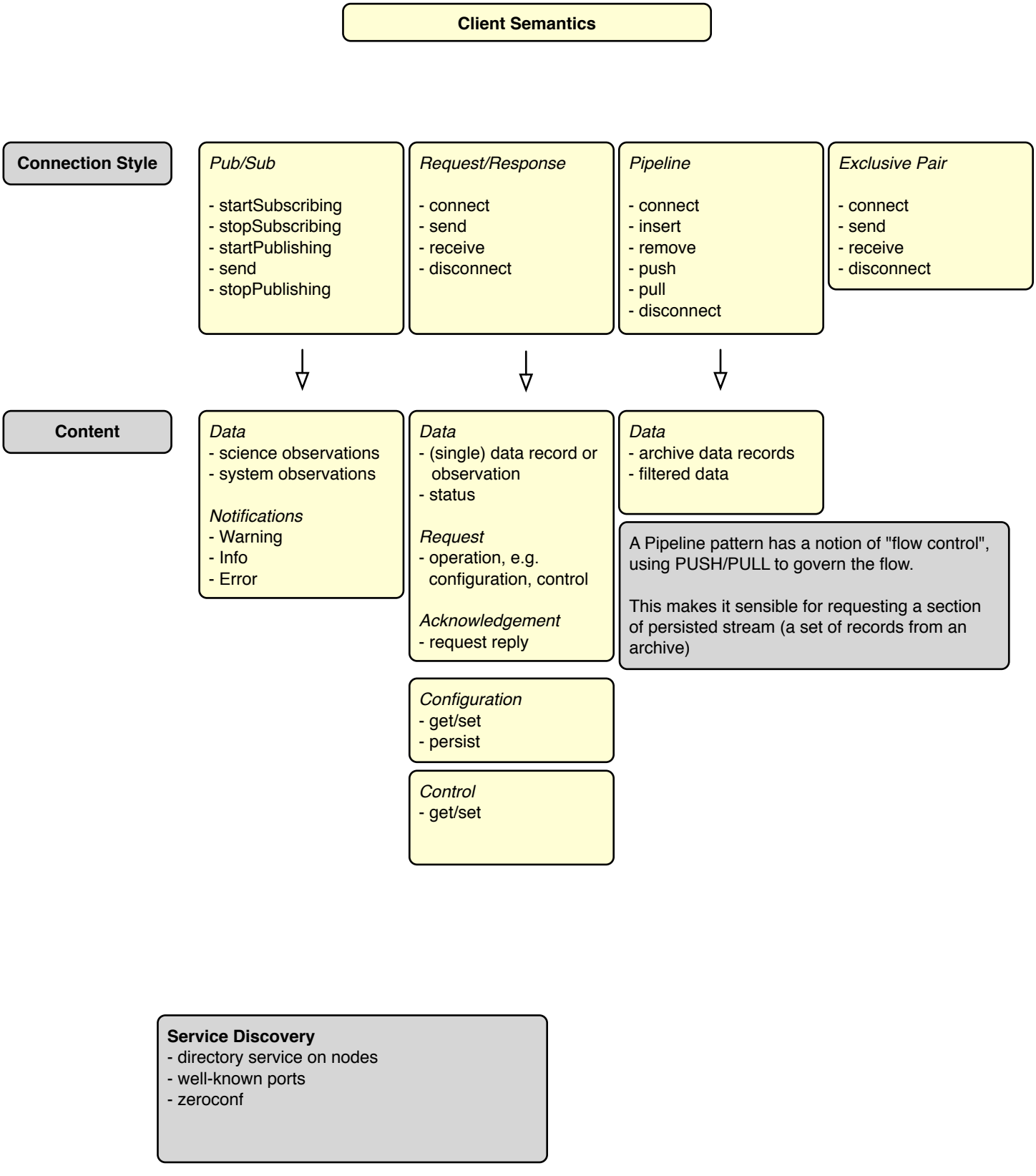
DataRequest
OperationRequest

Stream Operations
[uni/bi-directional?]
Create
Configure
SetCallback
Connect
Disconnect
Start
Stop
Available
Peek
Jump
Skip
Step
ApplyFilters
RemoveFilters
Get
Put
[Merge?]

Stream Payload Types
Record
struct...

Stream Frame




Client Semantics
Connection Style
Pub/Sub
- startSubscribing
- stopSubscribing
- startPublishing
- send
- stopPublishing
Request/Response
- connect
- send
- receive
- disconnect
Pipeline
- connect
- insert
- remove
- push
- pull
- disconnect
Exclusive Pair
- connect
- send
- receive
- disconnect
Content
Data
- science observations
- system observations
Notifications
- Warning
- Info
- Error
Data
- (single) data record or observation
- status
Request
- operation, e.g. configuration, control
Acknowledgement
- request reply
Configuration
- get/set
- persist
Control
- get/set
Data
- archive data records
- filtered data
A Pipeline pattern has a notion of "flow control", using PUSH/PULL to govern the flow.
This makes it sensible for requesting a section of persisted stream (a set of records from an archive)
Service Discovery
- directory service on nodes
- well-known ports
- zeroconf

Metadata Handling

What's important to capture?

Changes to the system configuration that affect the interpretation of data

Metadata use case:
System start
Service restart

Metadata use case:
Changing (replaciing) an instrument

Metadata use case:
Changing a service configuration parameter
variant: temporary
variant: persist across restart

Metadata use case:
Changing an instrument configuration via console

Measurement context are metadata parameters associated with a measurement that define the state affecting the measurement.
Changes to the measurement context should be captured in the data stream.

Measurement Context

- parent ID — Platform, gateway
- stream ID — functional index (e.g. interior pH)
- device ID — instrument, service instance
- type ID — record type
- format ID — data format
- active properties — current service configuration state
- parsing schema — data format definition
- deviceSettings — self-reported device configuration
- sequenceNumber — absolute sequence number
- timestamp

data

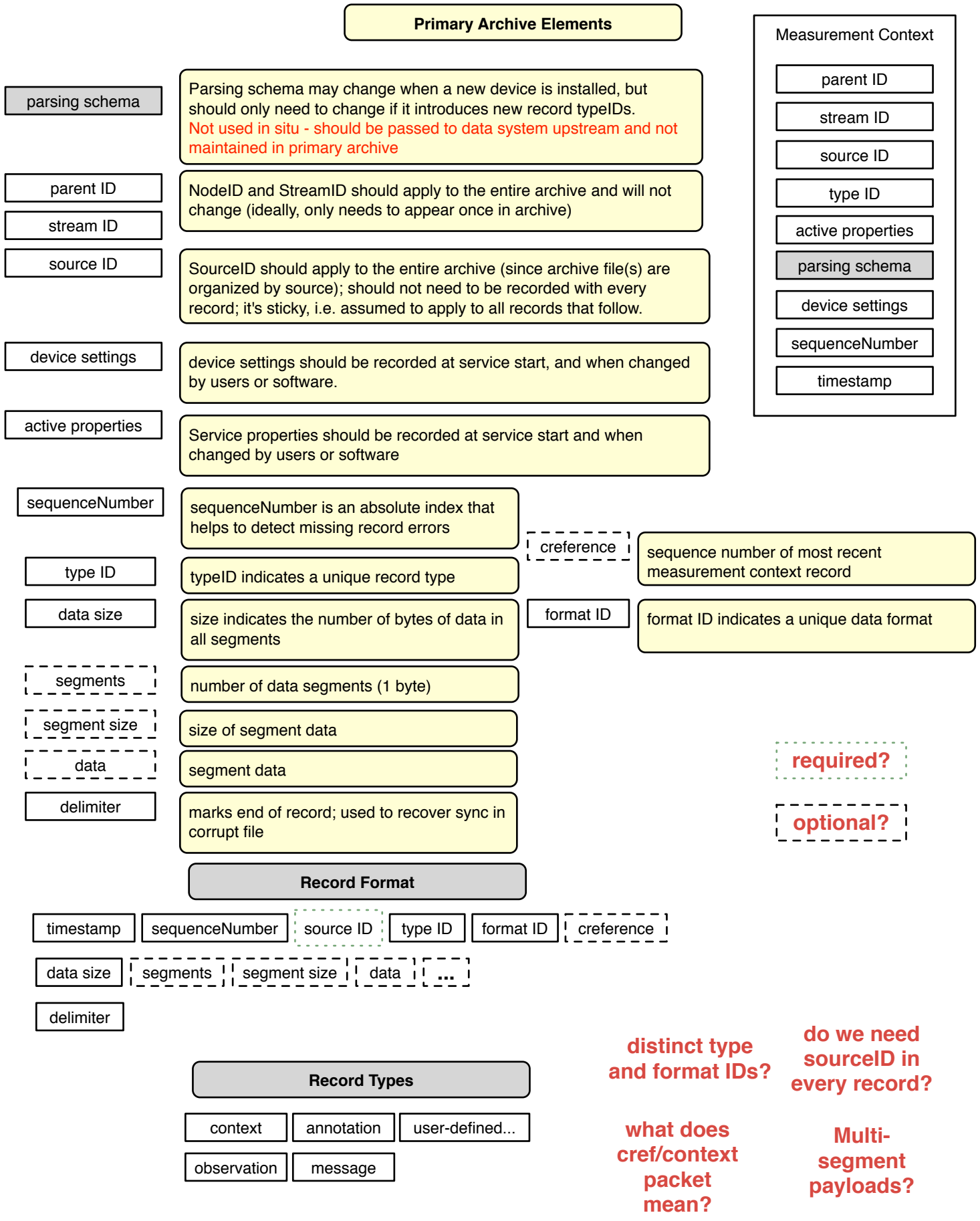
Metadata Payload

- InitialDefaultProperties
- DataStreams xml

read configuration
update configuration
write configuration

- DeviceSettings
- DefaultProperties
- ActiveProperties
- DataStreams xml

Not used in situ...should not maintain in archive


Primary Archive Elements
Measurement Context
parent ID
stream ID
source ID
type ID
active properties
parsing schema
device settings
sequenceNumber
timestamp
parsing schema
Parsing schema may change when a new device is installed, but should only need to change if it introduces new record typeIDs.
Not used in situ - should be passed to data system upstream and not maintained in primary archive
parent ID
stream ID
NodeID and StreamID should apply to the entire archive and will not change (ideally, only needs to appear once in archive)
source ID
SourceID should apply to the entire archive (since archive file(s) are organized by source); should not need to be recorded with every record; it's sticky, i.e. assumed to apply to all records that follow.
device settings
device settings should be recorded at service start, and when changed by users or software.
active properties
Service properties should be recorded at service start and when changed by users or software
sequenceNumber
sequenceNumber is an absolute index that helps to detect missing record errors
creference
sequence number of most recent measurement context record
type ID
typeID indicates a unique record type
data size
size indicates the number of bytes of data in all segments
format ID
format ID indicates a unique data format
segments
number of data segments (1 byte)
segment size
size of segment data
data
segment data
delimiter
marks end of record; used to recover sync in corrupt file
required?
optional?
Record Format
timestamp
sequenceNumber
source ID
type ID
format ID
creference
data size
segments
segment size
data
...
delimiter
Record Types
context
annotation
user-defined...
observation
message
distinct type and format IDs?
do we need sourceID in every record?
what does cref/context packet mean?
Multi-segment payloads?

Configuration

Configuration File Features

- Human-readable
- Machine-readable
- Persistence/serialization: support read/update/write
- Dynamic refresh (reload while running)

Configuration files persist a configuration data structure

Technology candidates:
JSON, BSON, MessagePack, libConf, libConfuse...

Configuration file contents

System Properties

General

name

proxies[name:port...]

Data

produces[streams:logs]

publishes[streams]

subscribes[streams]

[Service-Specific]

properties[]
name:value
array
struct
map/hash

paths[]
hardware addresses
power management
policies
telemetry
timeouts
service cfg [driver, ports, power, id, PUCK policy, modes]
ID map

Service Properties

General

name

proxies[name:port...]

Data

produces[streams:logs]

publishes[streams]

subscribes[streams]

[Device]

sampling[period, timeouts...]

CommunicationPowerPolicy

DevicePowerPolicy

Protocols[versions]

[Service-Specific]

e.g. modes, strings, constants, policies...

properties[]
name:value
array
struct
map/hash

gateway.config

service.config

service-defaults.config

service-streams.xml

...

Things that generate data streams need IDs:
- devices (instruments, system devices)
- virtual devices (control system)
- gateway nodes

Plug and Work

PUCK Support

- Optional for xFOCE
- Preferred for swFOCE

Payload:
- service-defaults.config
- service-streams.xml?

Service properties used by platform

service XML pass through data or reference - not used on platform. Should/should not log?

service XML reference are/are not shared

Gateway

gateway.config

service.config

service-defaults.config

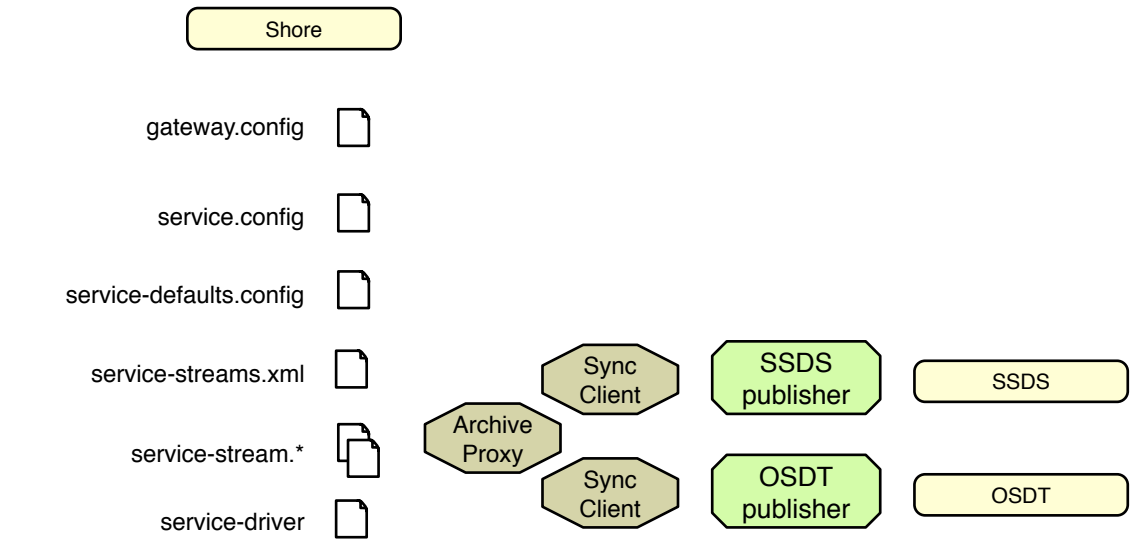
service-streams.xml

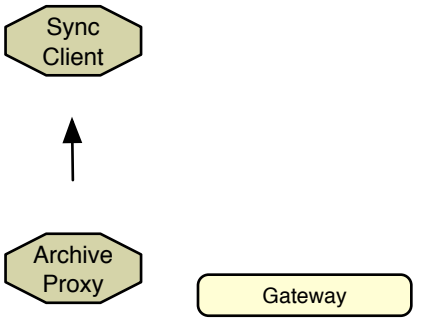
service-streams.xml

Data Distribution

Data sync use cases:
synchronous (push/pull)
asynchronous (pub/sub)
on demand (also synchronous)
restart shore side
restart gateway

Archive State (good enough):
parent ID
device ID
files[]
{
  Start key — index match
  End key — record match
  Missing keys[]
  Invalid keys[]
}



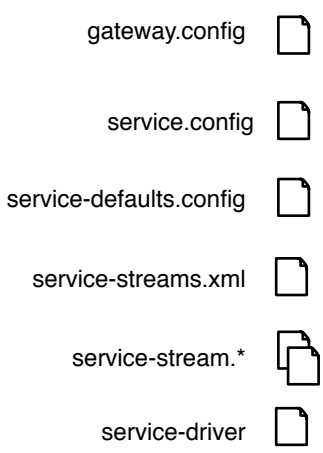


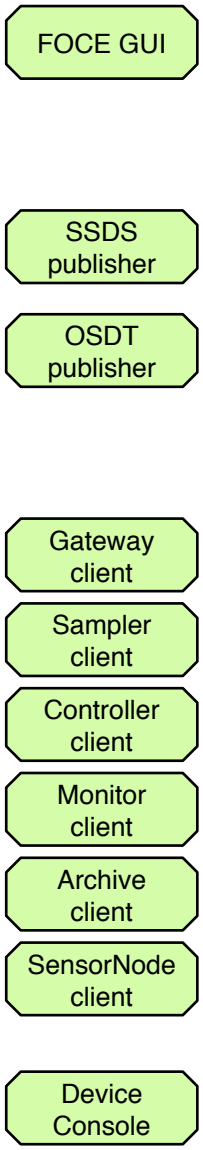
Binary Sync
Periodically and/or on demand
- stop sampling
- close files
- rsync* (...wget, scp, flippernet...)
- [start subscribers]
- resume sampling

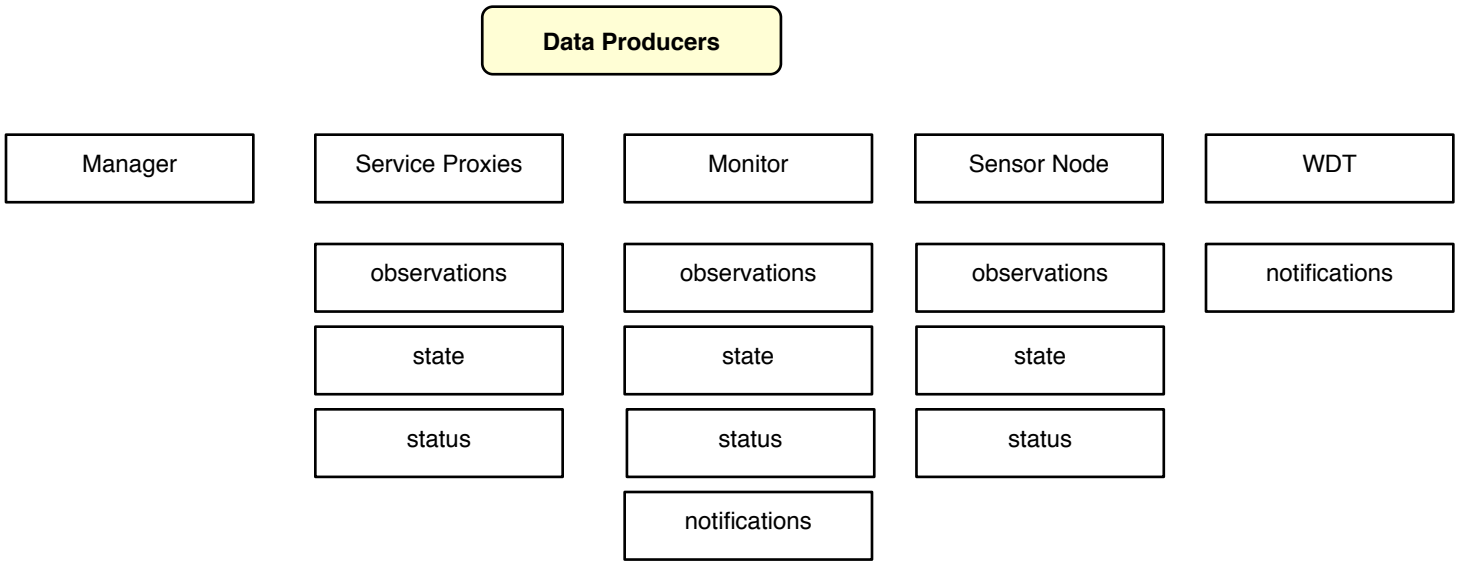
*** rsync excludes open/busy files**

Incremental Sync (client pull)
- DataService tracks archive state
- clients track local archive state
- clients sync at startup
  - query primary archive state
  - applies policies, filters (type, time, staleness)
  - pull data as needed
  - configurable
- clients subscribe to primary streams

Best Effort
- Primary archive DataService publishes all/most records
- Secondary archive (clients) subscribes whenever running
- Use binary sync as needed




FOCE GUI
SSDS publisher
OSDT publisher
Gateway client
Sampler client
Controller client
Monitor client
Archive client
SensorNode client
Device Console


Data Producers
Manager
Service Proxies
Monitor
Sensor Node
WDT
observations
state
status
observations
state
status
notifications
observations
state
status
notifications