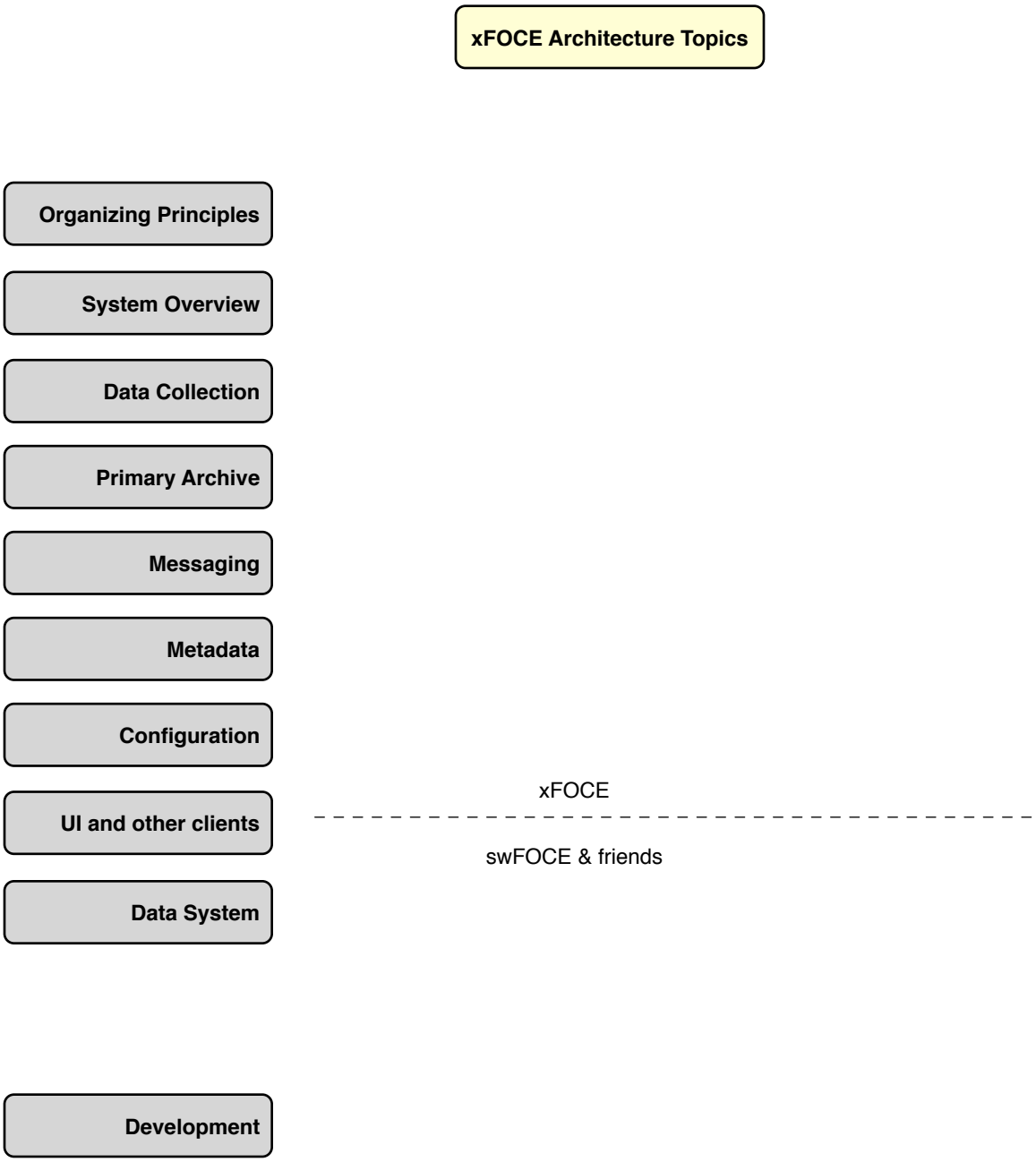

xFOCE Architecture Topics
Organizing Principles
System Overview
Data Collection
Primary Archive
Messaging
Metadata
Configuration
UI and other clients
Data System
Development
xFOCE
swFOCE & friends

xFOCE Architecture
Organizing Principles

Organizing Principles

Drivers

- data acquisition: reliable, simple, comprehensive, extensible, accessible
- Material cost
- Easy to build, use with minimal technical resources

- Delivering timestamped observations
- Managing change and risk
- COTS, Open source
- SensorNode - Gateway serial expansion
- Distributed

Simple by design

- Use (a few) simple patterns and repetition
- Separation of concerns
- No unnecessary features
- Design for tweakability: straight lines, minimal layers, plan for cut and paste

- Do as little as possible in a general, expandable way
- Define component interfaces
- Decouple client architectures
- Message passing, Service oriented

xFOCE Software Architecture Strategies

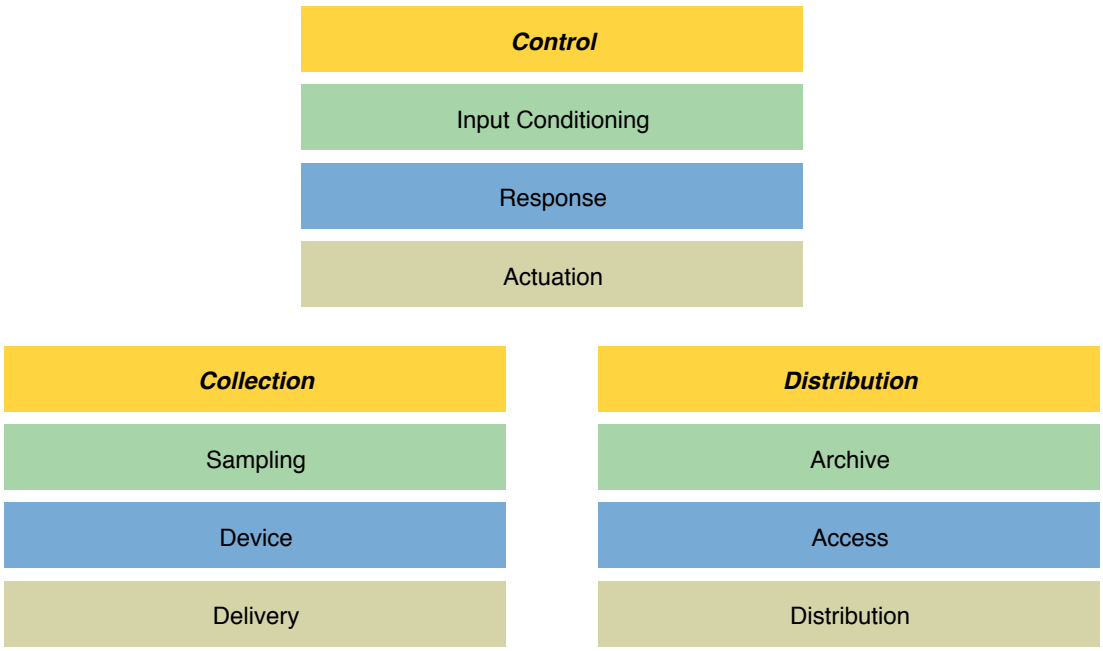
- Do a few things, simply and well: Volkswagen, In and Out Burger, Legos...
- Architecture should be tweakable by design
- Reduce barriers to using other hardware and reusing/modifying software
  - architectural pattern
  - software component choices

xFOCE has two main concerns:

controlling pH in a box
enable users to measure the effect on something

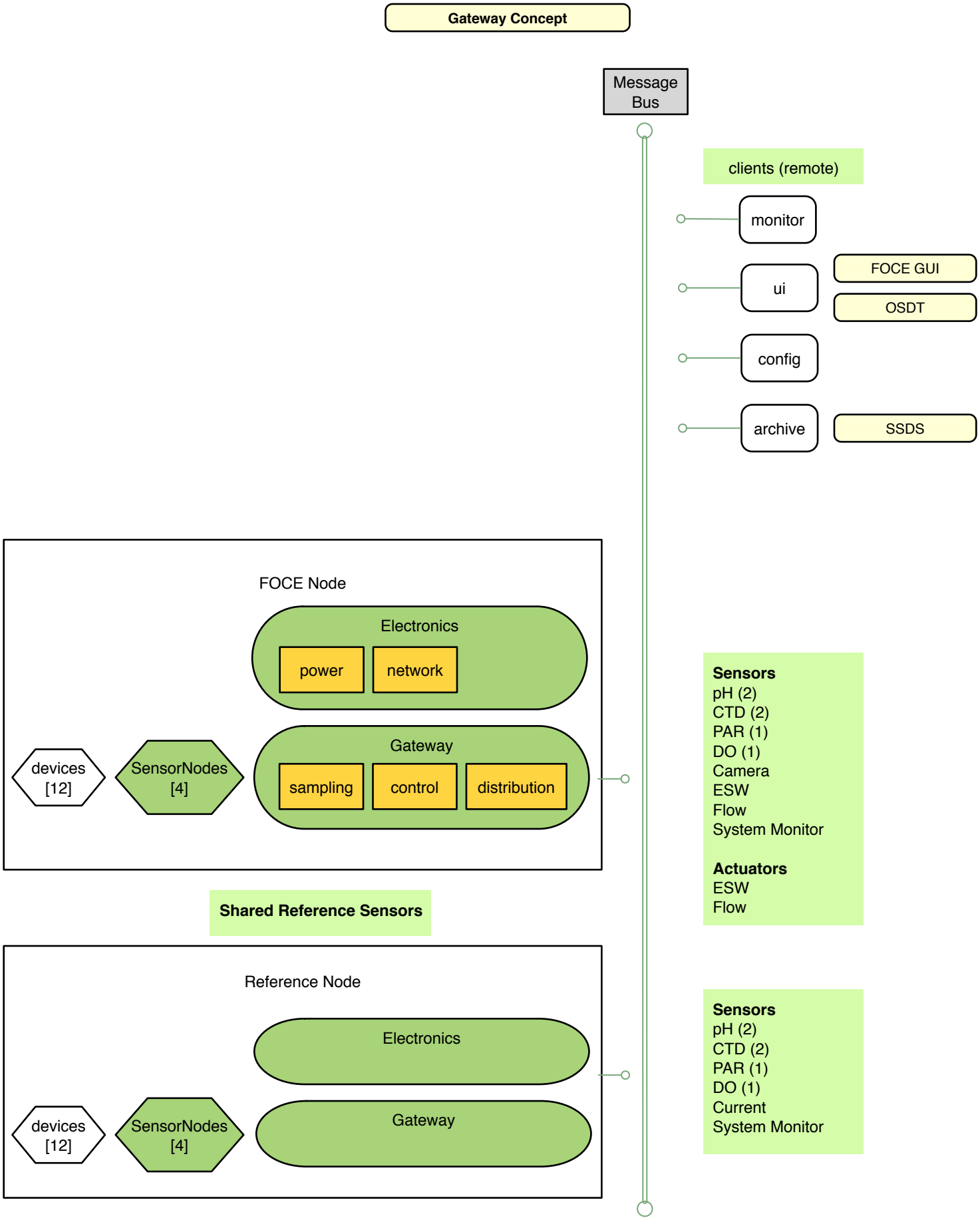
that are supported by two related concerns:

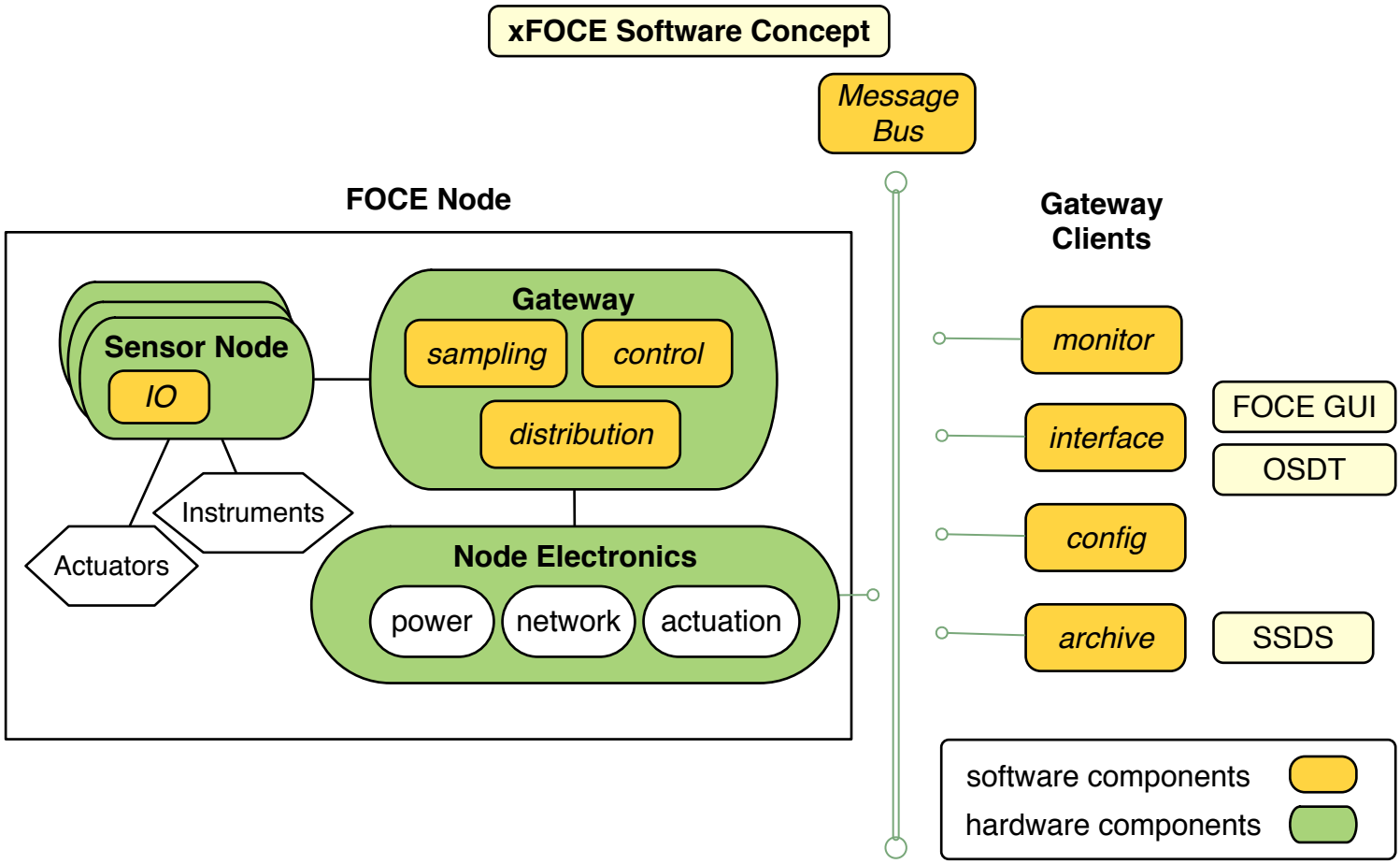
collecting data (recording measurements and context)
distributing data (internally and externally)



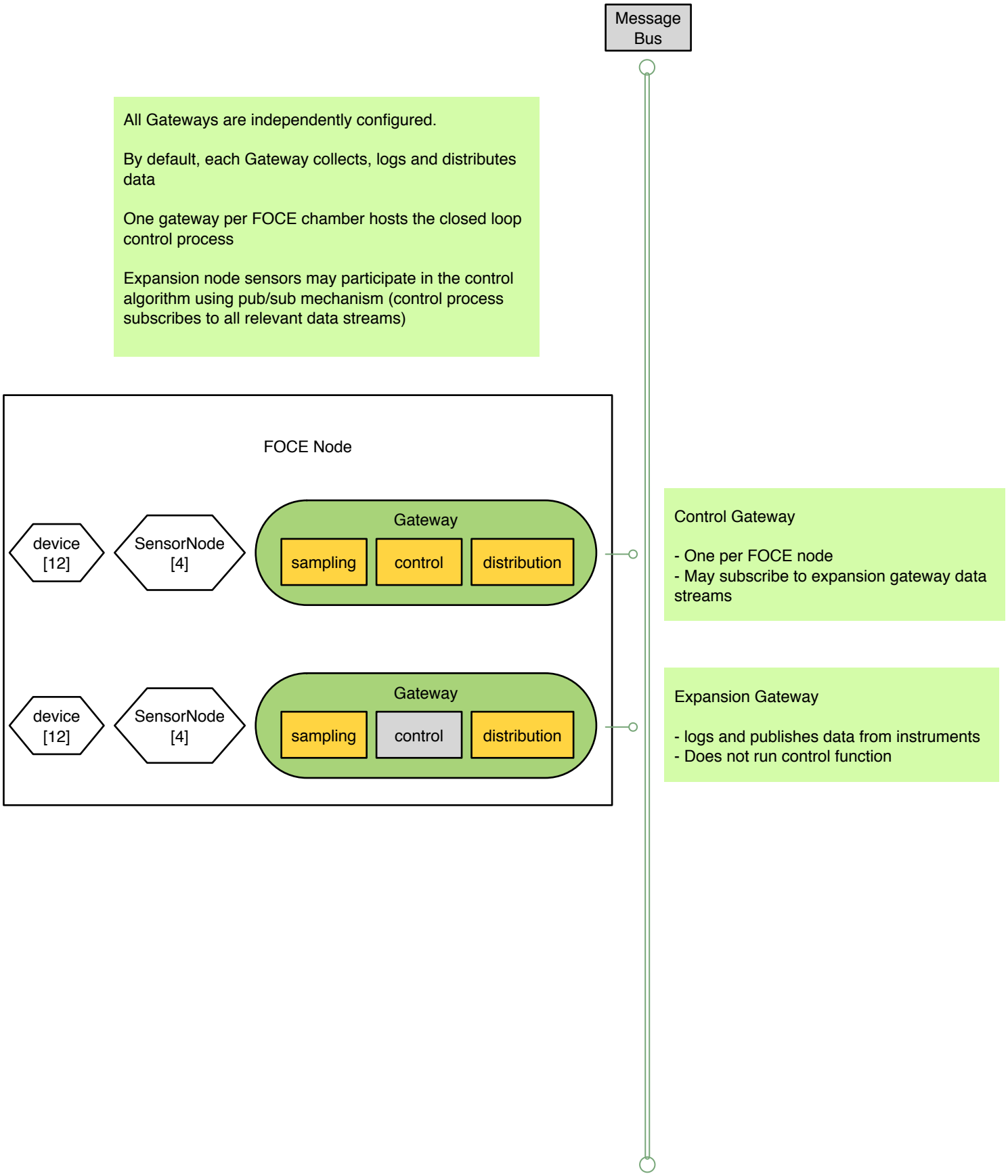
Consistency, symmetry and repetition make it easier to understand software.

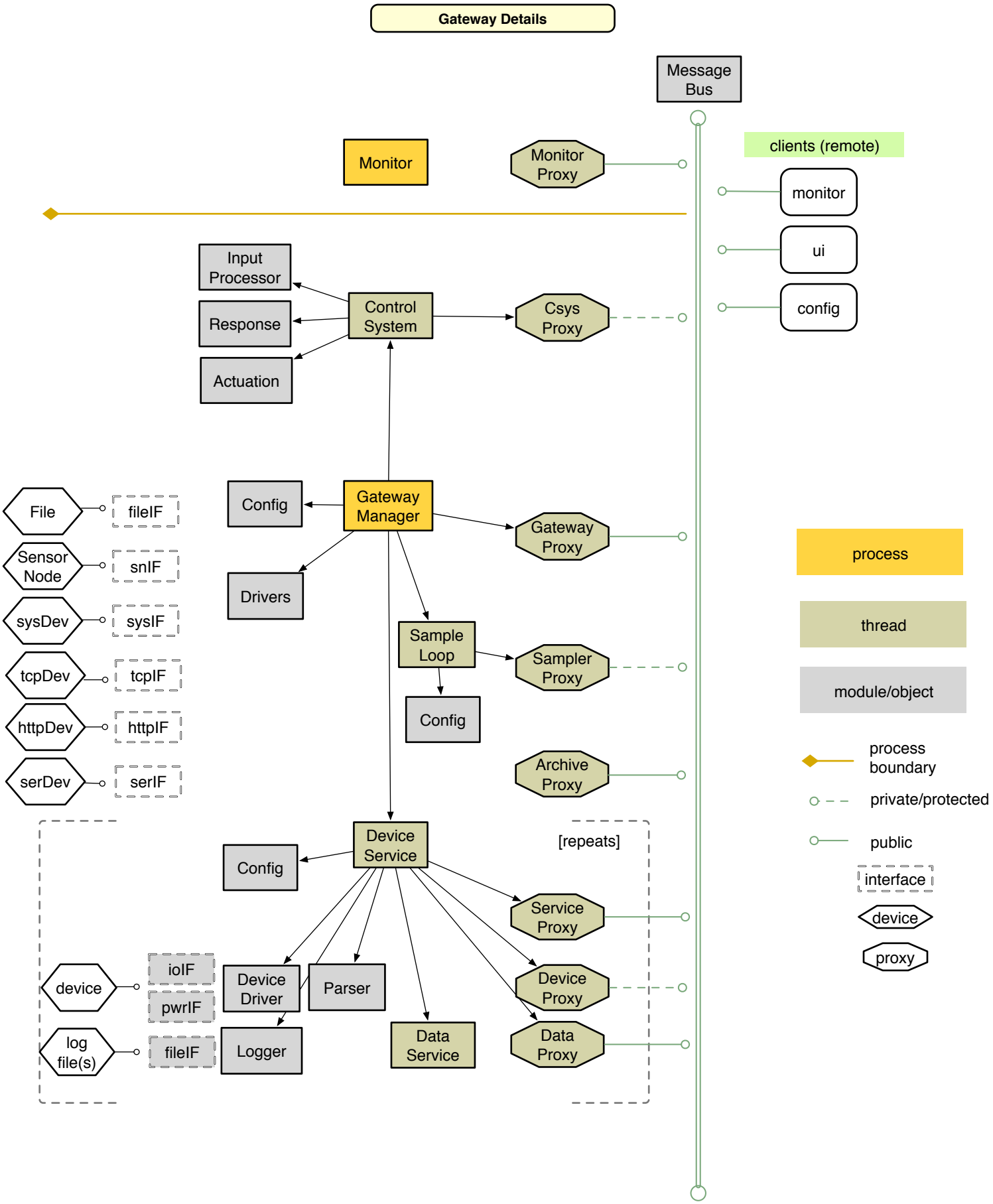
- Separate concerns to keep components as simple as possible
- Minimize component count and arrange into simple patterns
- Reuse the patterns as much as possible
- Be consistent when applying/using patterns
- If something doesn't fit into an existing pattern
  - re-factor patterns (especially if several similar things don't quite fit)
  - create a new one


Gateway Concept
Message Bus
clients (remote)
monitor
ui
FOCE GUI
OSDT
config
archive
SSDS
FOCE Node
Electronics
power
network
Gateway
sampling
control
distribution
devices [12]
SensorNodes [4]
Sensors
pH (2)
CTD (2)
PAR (1)
DO (1)
Camera
ESW
Flow
System Monitor
Actuators
ESW
Flow
Shared Reference Sensors
Reference Node
Electronics
Gateway
devices [12]
SensorNodes [4]
Sensors
pH (2)
CTD (2)
PAR (1)
DO (1)
Current
System Monitor


xFOCE Software Concept
Message Bus
FOCE Node
Gateway Clients
Sensor Node
IO
Gateway
sampling
control
distribution
Instruments
Actuators
Node Electronics
power
network
actuation
monitor
interface
config
archive
FOCE GUI
OSDT
SSDS
software components
hardware components

Gateway Expansion




Gateway Details
Message Bus
Monitor
Monitor Proxy
clients (remote)
monitor
ui
config
Input Processor
Response
Actuation
Control System
Csys Proxy
File
fileIF
Sensor Node
snIF
sysDev
sysIF
tcpDev
tcpIF
httpDev
httpIF
serDev
serIF
Config
Gateway Manager
Gateway Proxy
Drivers
Sample Loop
Sampler Proxy
Config
Archive Proxy
[repeats]
Device Service
Config
Service Proxy
Device Proxy
Data Proxy
device
ioIF
pwrIF
log file(s)
fileIF
Device Driver
Parser
Logger
Data Service
process
thread
module/object
process boundary
private/protected
public
interface
device
proxy

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

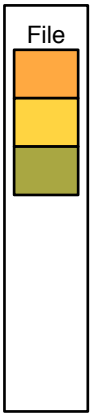
Segmenting can help make it easier to find a specific piece of data

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

Disadvantages
Access tools must span multiple files.

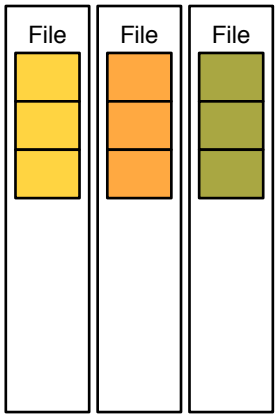
More files to manage

Heterogeneous, Monolithic

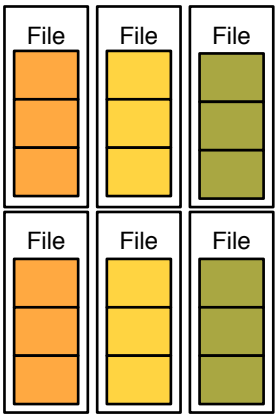


Heterogeneous, Segmented

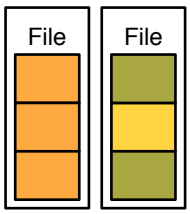
Homogeneous, Monolithic



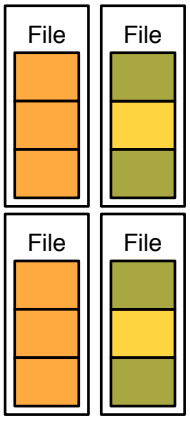
Homogeneous, Segmented



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:

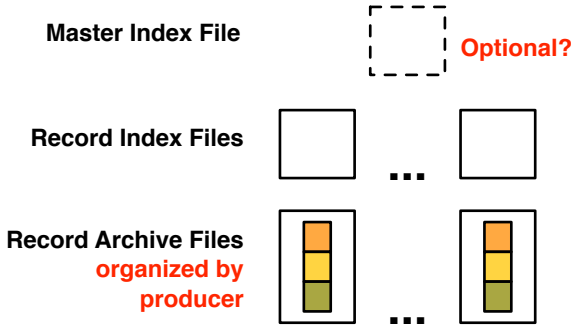
- Useful to keep data in files by producer/device (b/c easier to apply calibrations, etc. before concatenation in work flow)
- Useful to include human-sensible key(s) and maps that enable sorting streams (e.g. "node1", "pH1") w/o remembering numeric IDs
- Useful to be able to access a small slice of 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]



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.

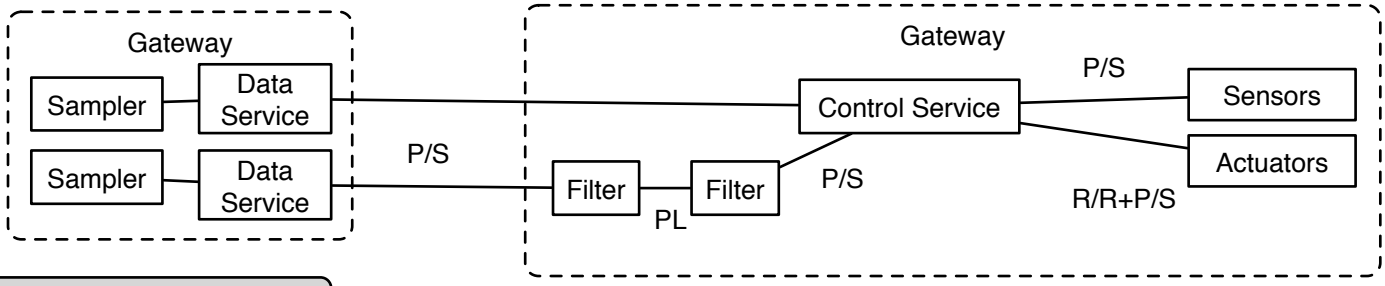
Publish-subscribe (P/S)
Connects a set of publishers to a set of subscribers. This is a data distribution pattern.

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.

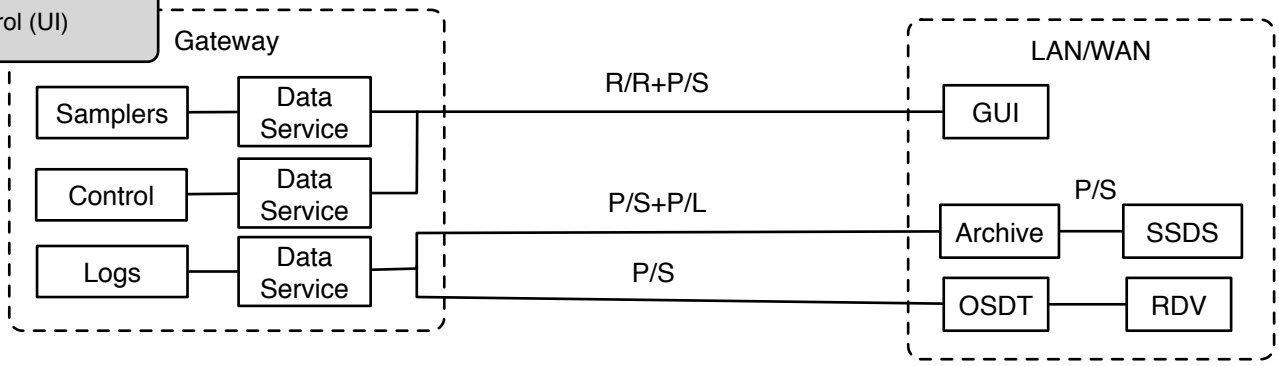
Messaging Use Cases

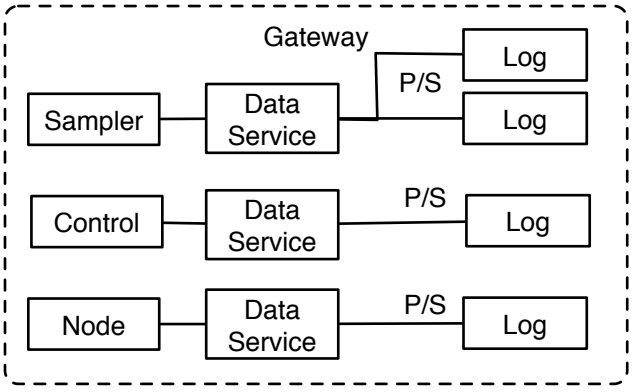
Control System data input
Service Coordination
Control system filtering (pipeline)



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

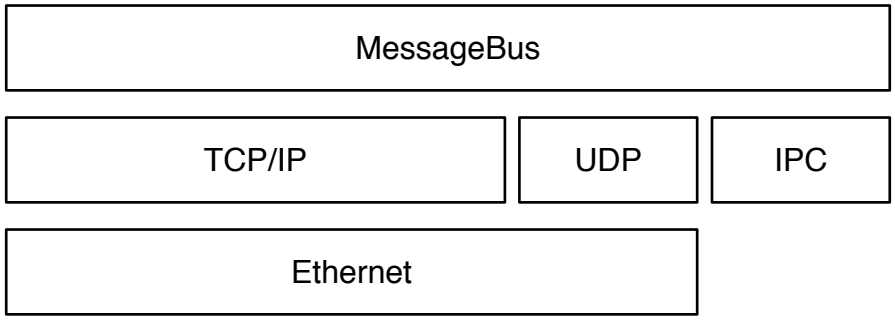
Other candidates

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, from database)
Google protocol buffers (Java/C++ only)



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.

Metadata Payload

- InitialDefaultProperties
- DataStreams xml

read configuration
update configuration
write configuration

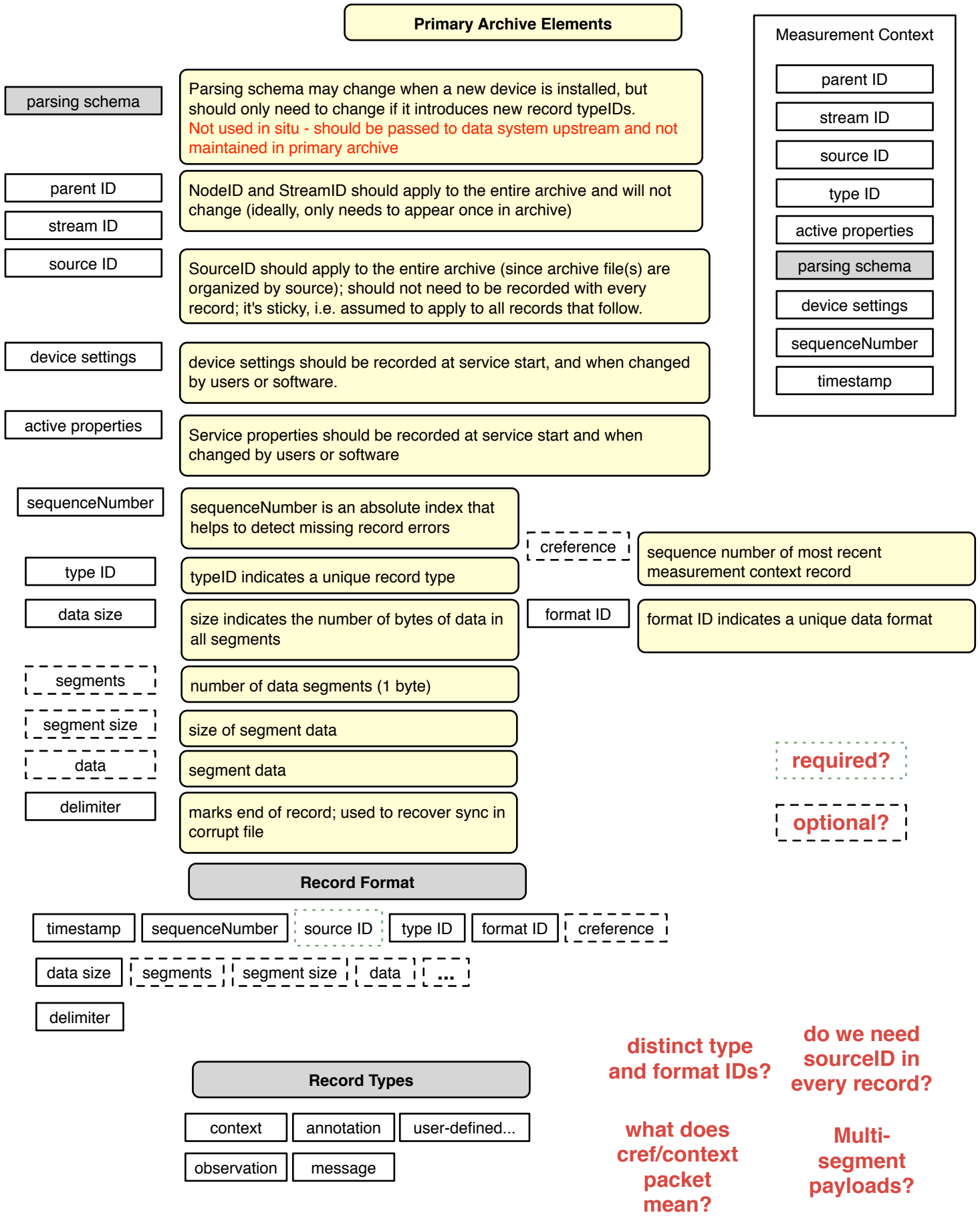
- DeviceSettings
- DefaultProperties
- ActiveProperties
- DataStreams xml

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

Measurement Context

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

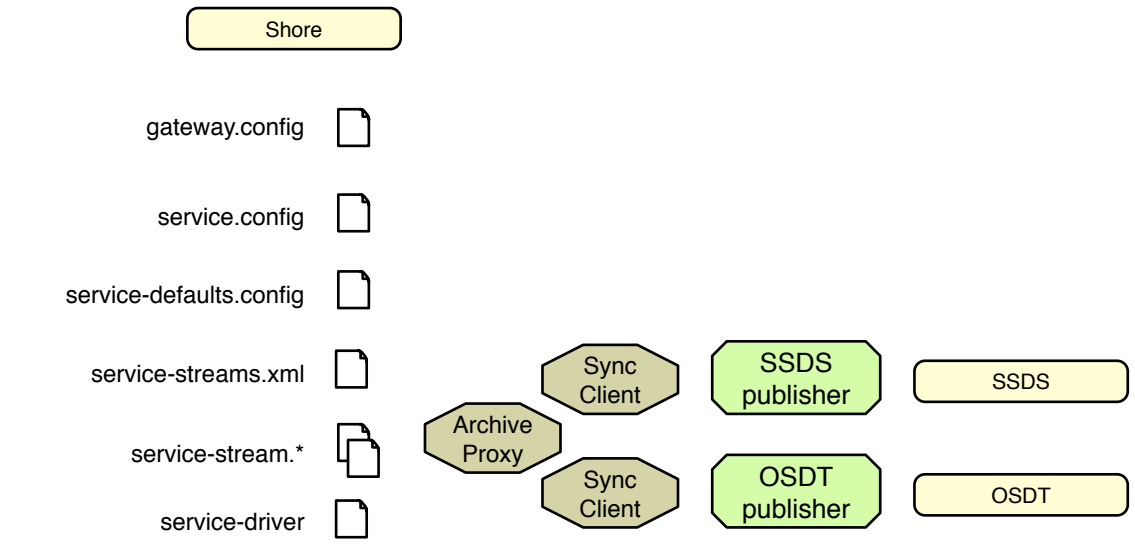
data

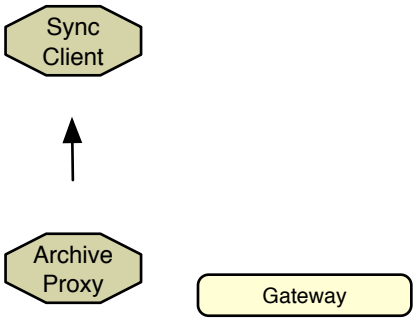

Primary Archive Elements
Measurement Context
parent ID
stream ID
source ID
type ID
active properties
parsing schema
device settings
sequenceNumber
timestamp
parsing schema
Parsing schema may change when a new device is installed, but should only need to change if it introduces new record typeIDs.
Not 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?

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[]
}



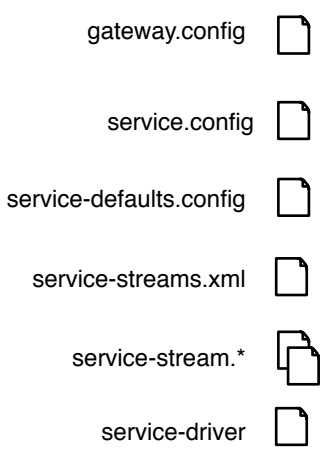


Binary Sync
Periodically and/or on demand
- stop sampling
- close files
- rsync* (...wget, scp, flippernet...)
- [start subscribers]
- resume sampling

*** 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



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

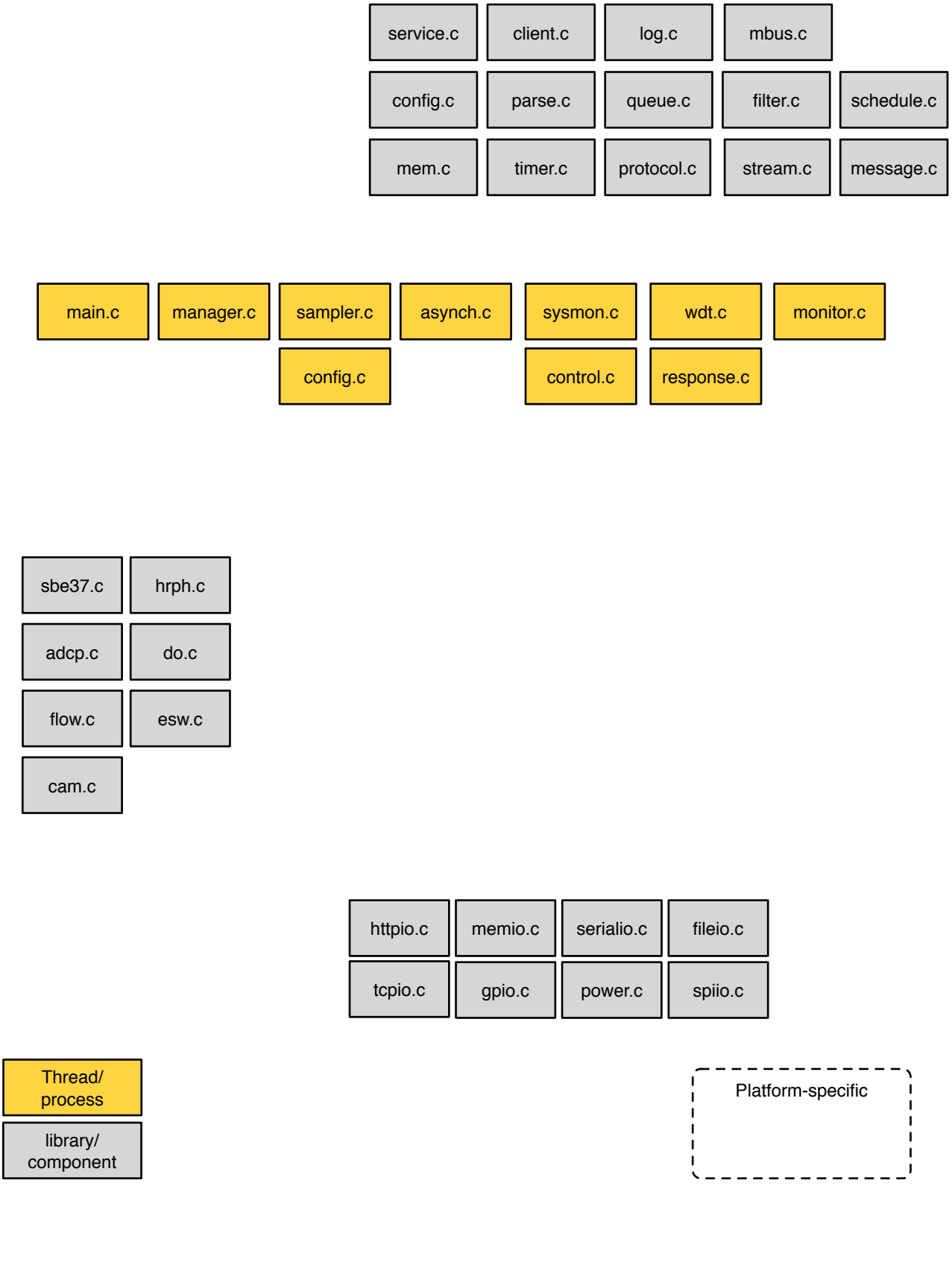
service.config

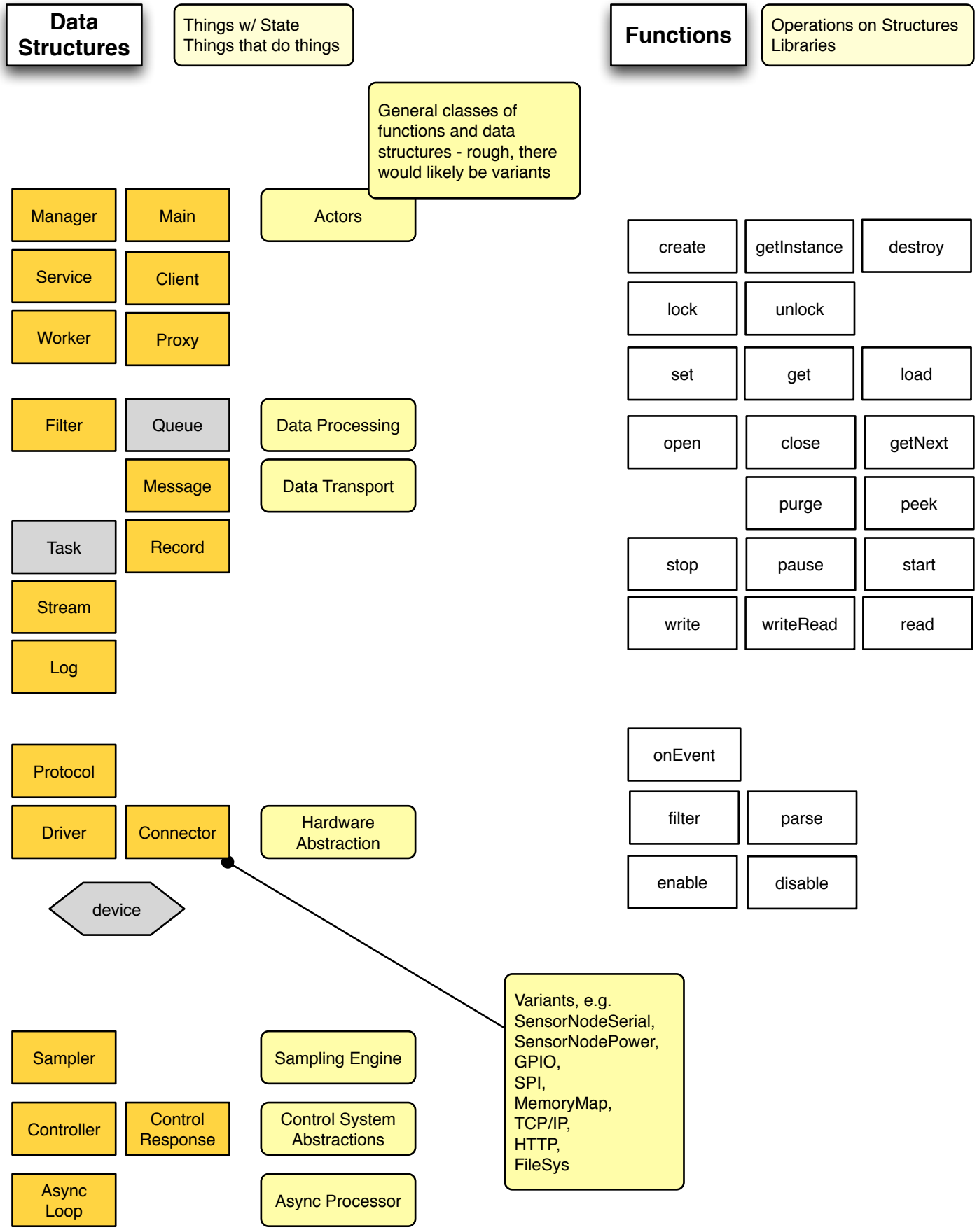
service-defaults.config

service-streams.xml

...

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


service.c
client.c
log.c
mbus.c
config.c
parse.c
queue.c
filter.c
schedule.c
mem.c
timer.c
protocol.c
stream.c
message.c
main.c
manager.c
sampler.c
asynch.c
sysmon.c
wdt.c
monitor.c
config.c
control.c
response.c
sbe37.c
hrph.c
adcp.c
do.c
flow.c
esw.c
cam.c
httpio.c
memio.c
serialio.c
fileio.c
tcpio.c
gpio.c
power.c
spiio.c
Thread/ process
library/ component
Platform-specific


Data Structures
Things w/ State
Things that do things
Functions
Operations on Structures
Libraries
General classes of functions and data structures - rough, there would likely be variants
Manager
Main
Actors
Service
Client
Worker
Proxy
Filter
Queue
Data Processing
Message
Data Transport
Task
Record
Stream
Log
Protocol
Driver
Connector
Hardware Abstraction
device
Sampler
Sampling Engine
Controller
Control Response
Control System Abstractions
Async Loop
Async Processor
Variants, e.g.
SensorNodeSerial,
SensorNodePower,
GPIO,
SPI,
MemoryMap,
TCP/IP,
HTTP,
FileSys
create
getInstance
destroy
lock
unlock
set
get
load
open
close
getNext
purge
peek
stop
pause
start
write
writeRead
read
onEvent
filter
parse
enable
disable

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

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

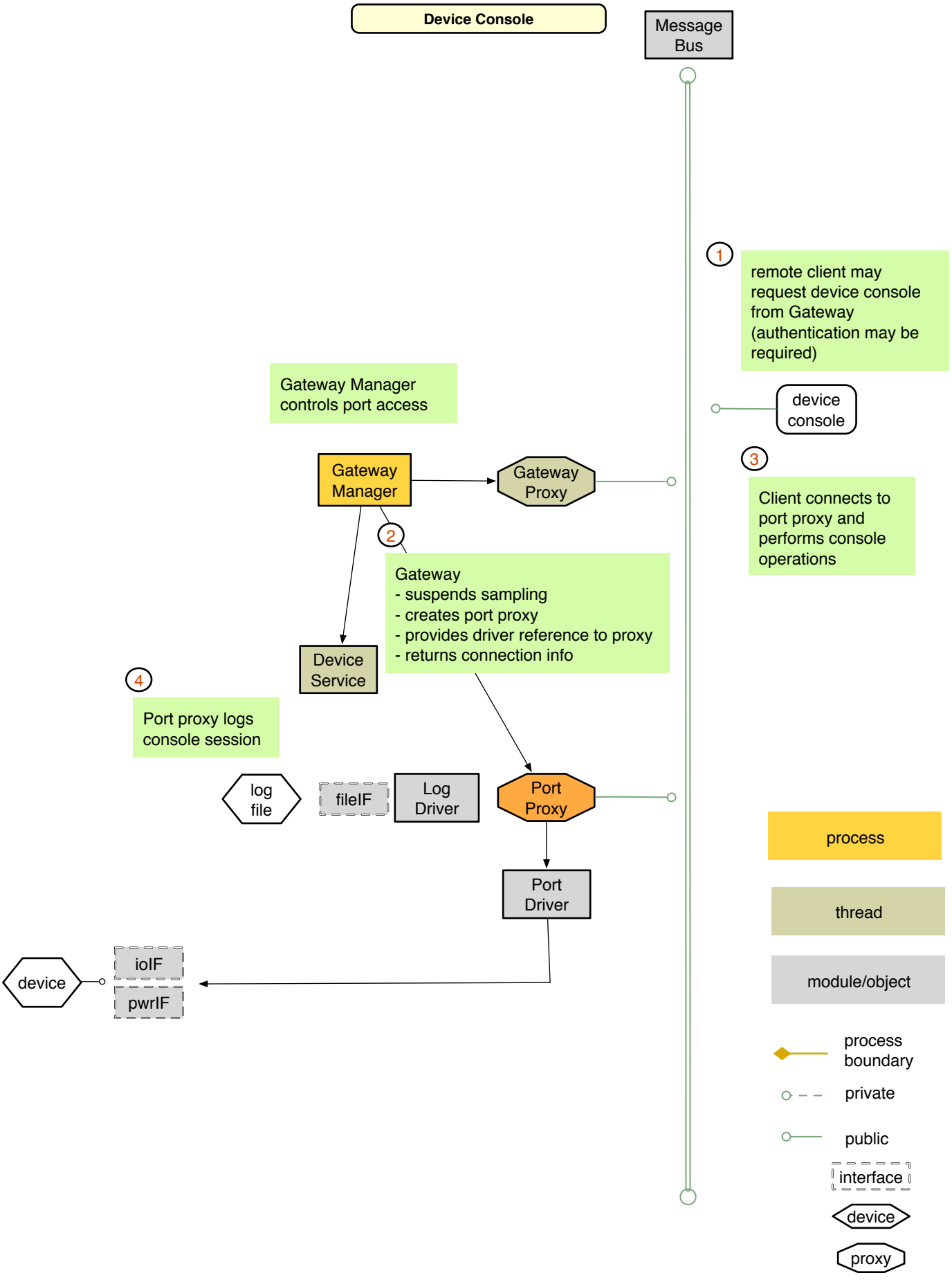
Acknowledgement
- request reply

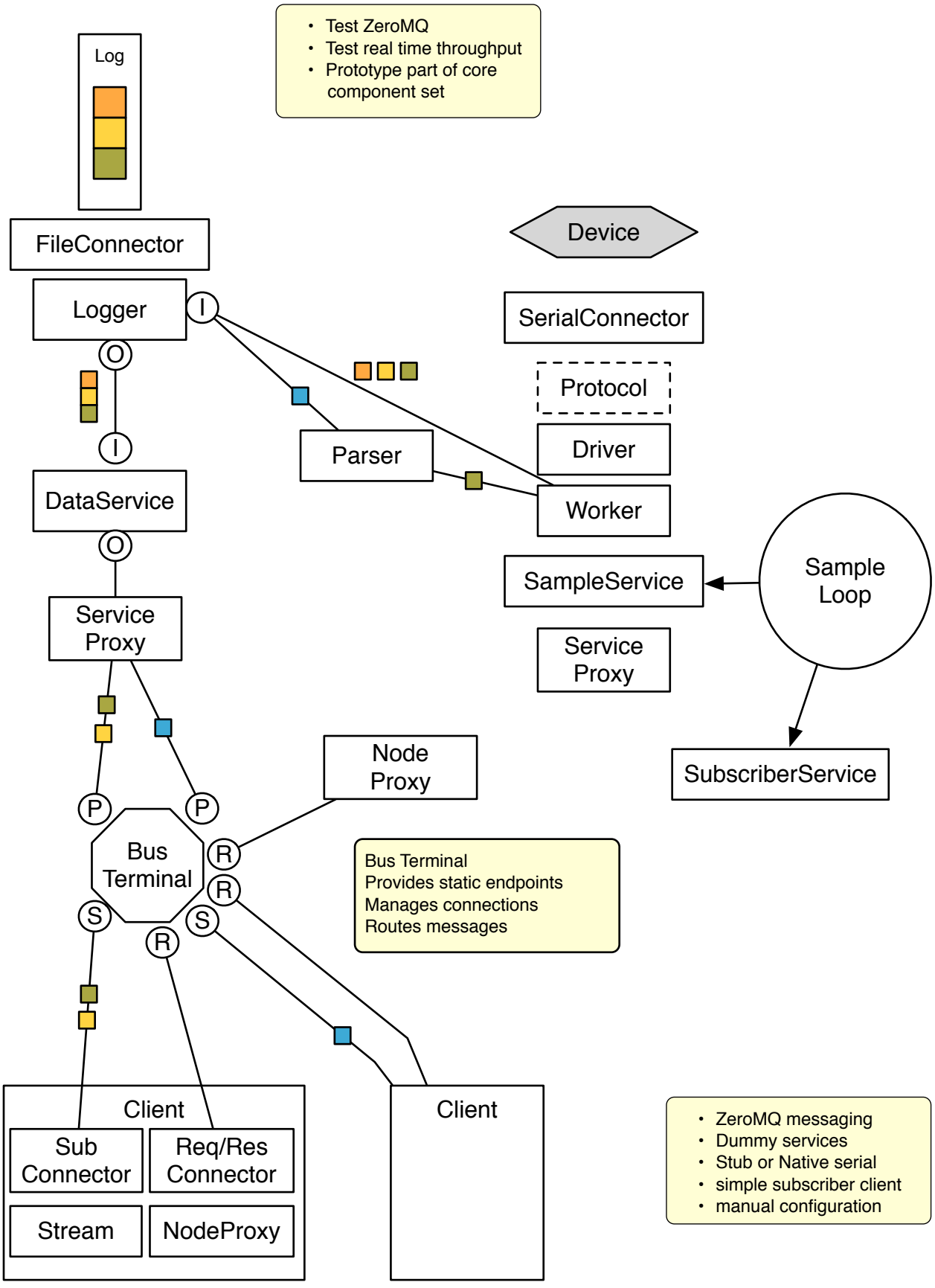
Data
- archive data records
- filtered data

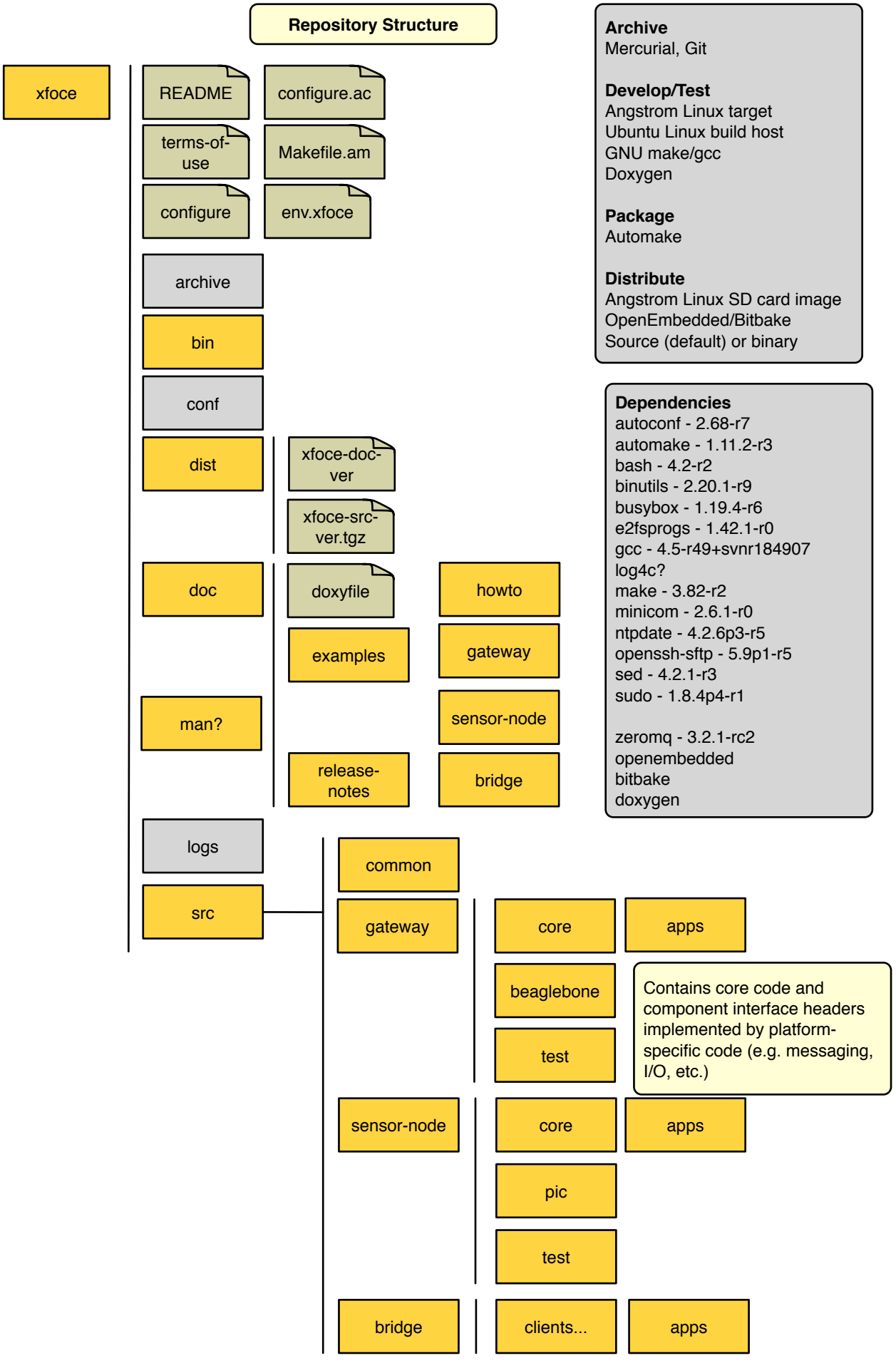
A Pipeline pattern has a notion of "flow control", using PUSH/PULL to govern the flow.

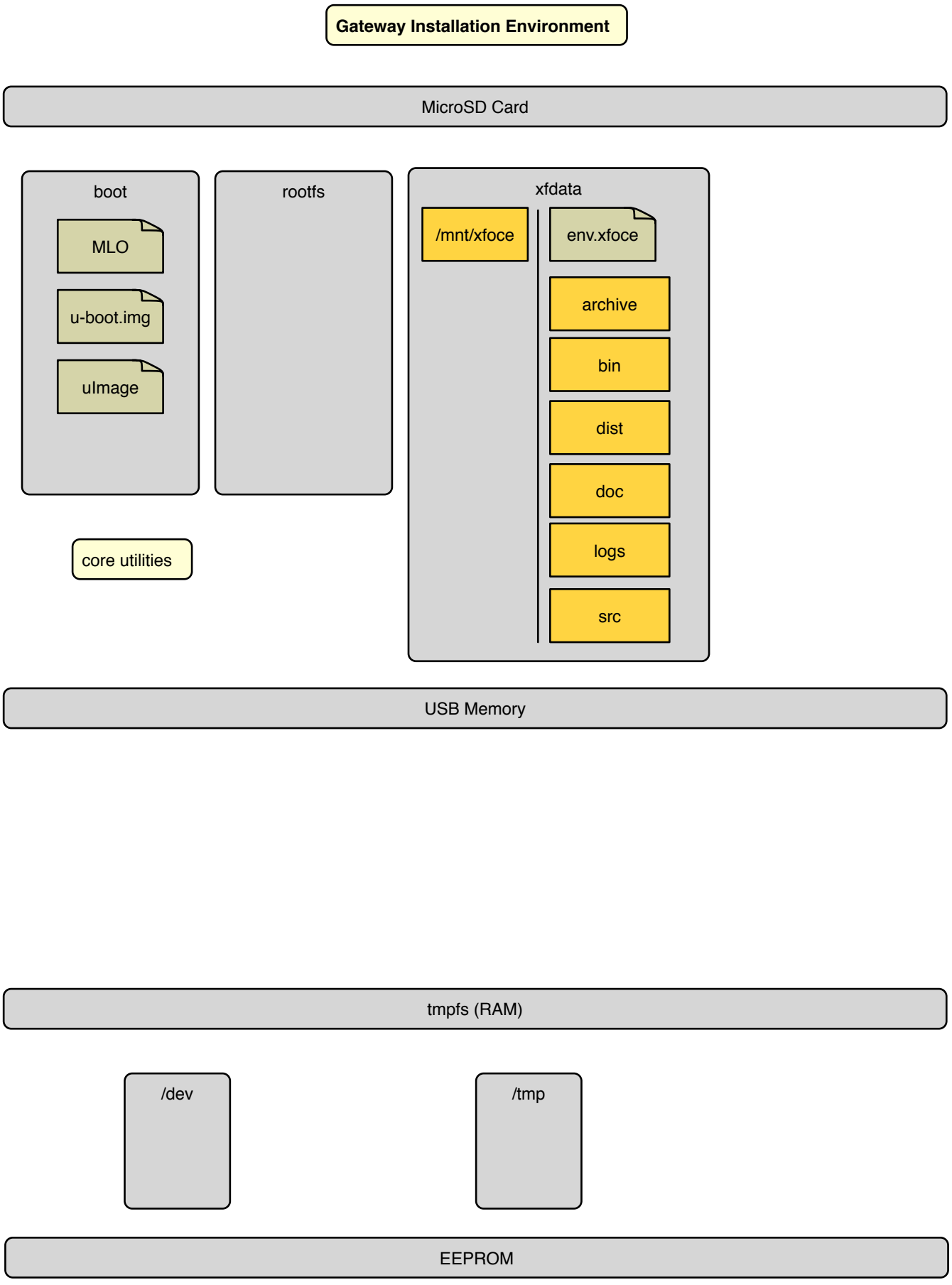
This makes it sensible for requesting a section of persisted stream (a set of records from an archive)

Service Discovery
Clients use directory service on nodes


Device Console
Message Bus
1
remote client may request device console from Gateway (authentication may be required)
device console
3
Client connects to port proxy and performs console operations
Gateway Manager controls port access
Gateway Manager
Gateway Proxy
2
Gateway
- suspends sampling
- creates port proxy
- provides driver reference to proxy
- returns connection info
Device Service
4
Port proxy logs console session
log file
fileIF
Log Driver
Port Proxy
Port Driver
device
ioIF
pwrIF
process
thread
module/object
process boundary
private
public
interface
device
proxy

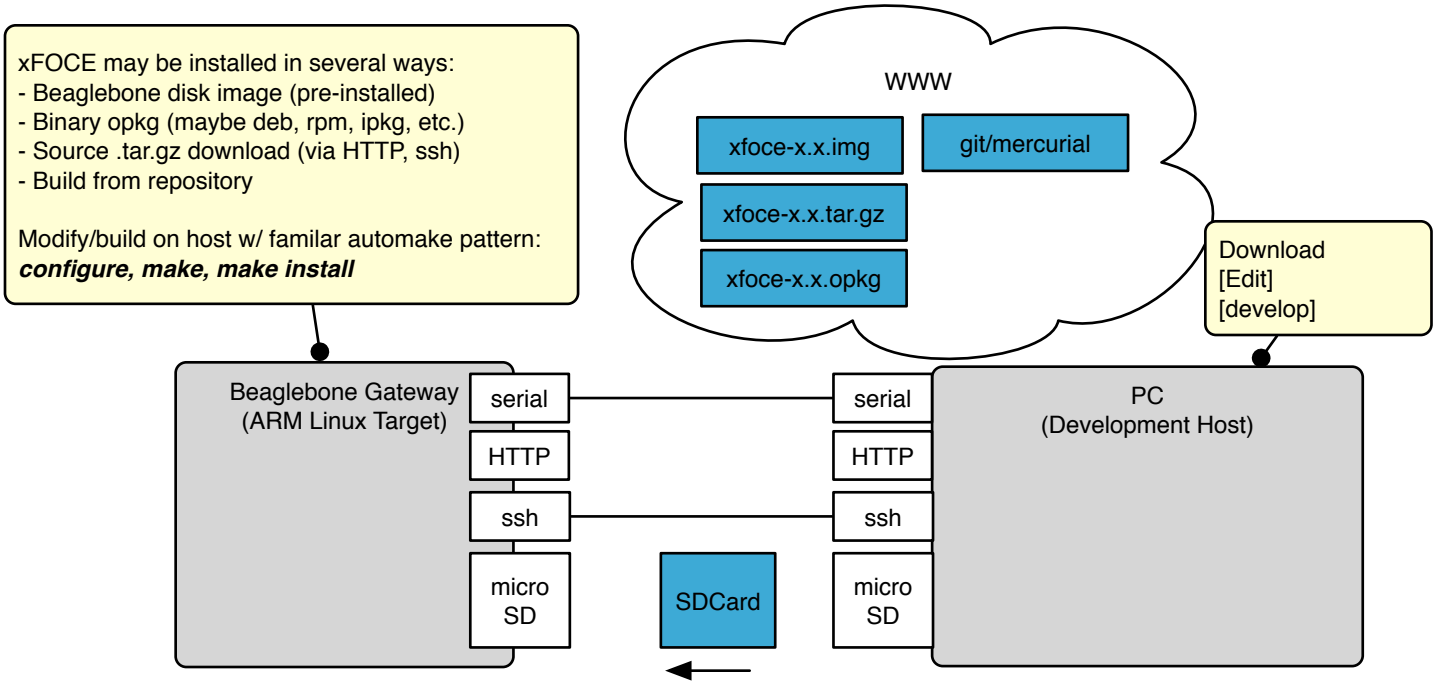

• Test ZeroMQ
• Test real time throughput
• Prototype part of core component set
Log
FileConnector
Logger
I
O
I
DataService
O
Service Proxy
Device
SerialConnector
Protocol
Driver
Parser
Worker
SampleService
Service Proxy
Sample Loop
SubscriberService
Node Proxy
P
P
Bus Terminal
R
R
S
S
R
Bus Terminal
Provides static endpoints
Manages connections
Routes messages
Client
Sub Connector
Req/Res Connector
Stream
NodeProxy
Client
• ZeroMQ messaging
• Dummy services
• Stub or Native serial
• simple subscriber client
• manual configuration


Repository Structure
xfoce
README
configure.ac
terms-of-use
Makefile.am
configure
env.xfoce
archive
bin
conf
dist
xfoce-doc-ver
xfoce-src-ver.tgz
doc
doxyfile
howto
examples
gateway
man?
sensor-node
release-notes
bridge
logs
src
common
gateway
core
apps
beaglebone
test
Contains core code and component interface headers implemented by platform-specific code (e.g. messaging, I/O, etc.)
sensor-node
core
apps
pic
test
bridge
clients...
apps
Archive
Mercurial, Git
Develop/Test
Angstrom Linux target
Ubuntu Linux build host
GNU make/gcc
Doxygen
Package
Automake
Distribute
Angstrom Linux SD card image
OpenEmbedded/Bitbake
Source (default) or binary
Dependencies
autoconf - 2.68-r7
automake - 1.11.2-r3
bash - 4.2-r2
binutils - 2.20.1-r9
busybox - 1.19.4-r6
e2fsprogs - 1.42.1-r0
gcc - 4.5-r49+svnr184907
log4c?
make - 3.82-r2
minicom - 2.6.1-r0
ntpdate - 4.2.6p3-r5
openssh-sftp - 5.9p1-r5
sed - 4.2.1-r3
sudo - 1.8.4p4-r1
zeromq - 3.2.1-rc2
openembedded
bitbake
doxygen

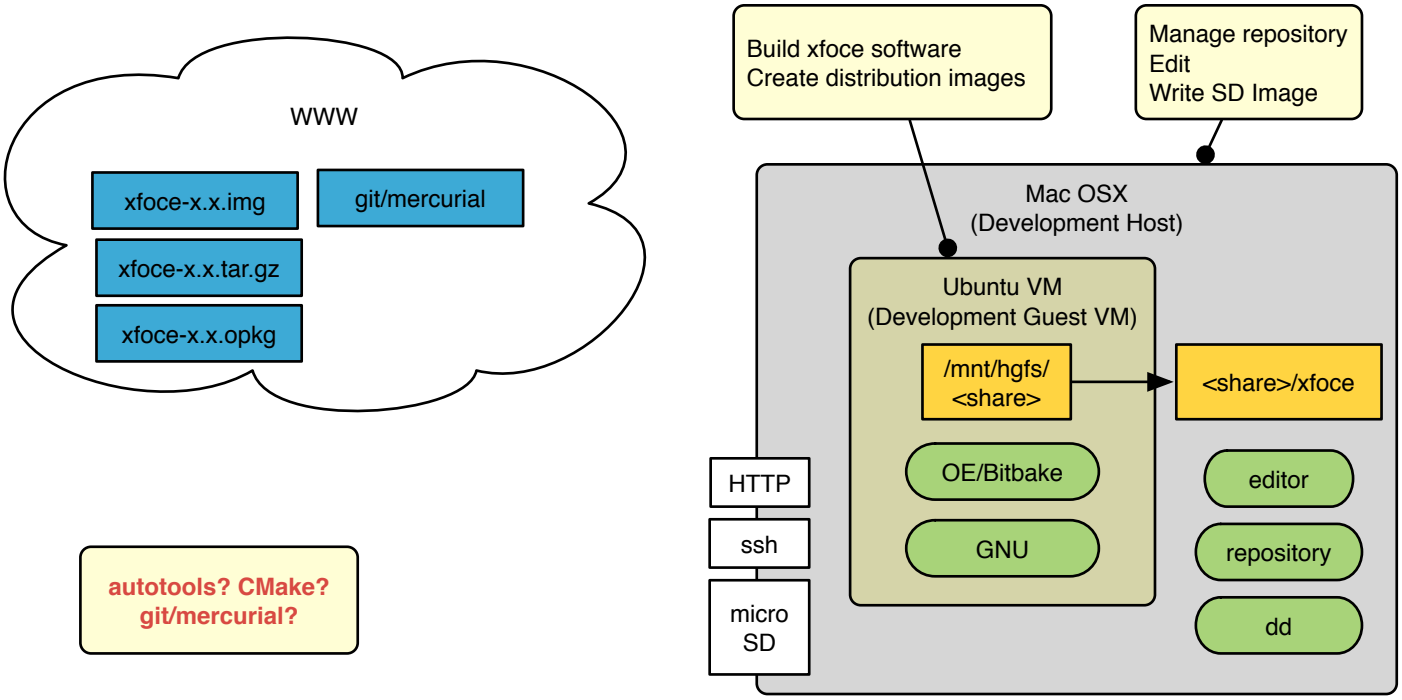

Gateway Installation Environment
MicroSD Card
boot
MLO
u-boot.img
uImage
rootfs
xfdata
/mnt/xfoce
env.xfoce
archive
bin
dist
doc
logs
src
core utilities
USB Memory
tmpfs (RAM)
/dev
/tmp
EEPROM

Building xFOCE Gateway Distributions

Users



Developers



Building xFOCE Gateway Distributions

Users

xFOCE Gateway Binary Distribution
May be distributed as a binary micro SD card image:
Requirements:
Beaglebone, SD Card, web browser

- Download image
- Transfer image to micro SD card (Mac, Win, Linux...)
- Put the SD Card in a Beaglebone

xFOCE Gateway Source Distribution
May be distributed via repository or tar.gz
Requirements:
Build on Beaglbone:

- Download image / get repository
- Install to micro SD card
- Put the SD Card in a Beaglebone

Build on Ubuntu Cross Host:

Modify, Build, Test, Deploy, Update

Build on Target (no host setup required)
Build on Host (faster compilation)
Test (use sandbox)
Deploy
Update Repository

xFOCE may be installed in several ways:
- Beaglebone disk image (pre-installed)
- Binary opkg (maybe deb, rpm, ipkg, etc.)
- Source .tar.gz download (via HTTP, ssh)

Modify/build on host w/ familar automake pattern:
configure, make, make install

Developers

xFOCE Gateway Development Tools (xfoce.dev)
May be distributed as a VMWare virtual disk:
Requirements: VMWare, bitbake recipies

Set up Ubuntu VM
Install tools and packages (OpenEmbedded, Bitbake, gcc cross, etc.)

Building Beaglebone Binary Distribution Image
Requirements:
VMWare xfoce.dev image
VMWare SDCard image

Building Beaglebone Source Distribution
Requirements: VMWare xfoce.dev image