

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)

Control

Input Conditioning

Response

Actuation

Collection

Sampling

Device

Delivery

Distribution

Archive

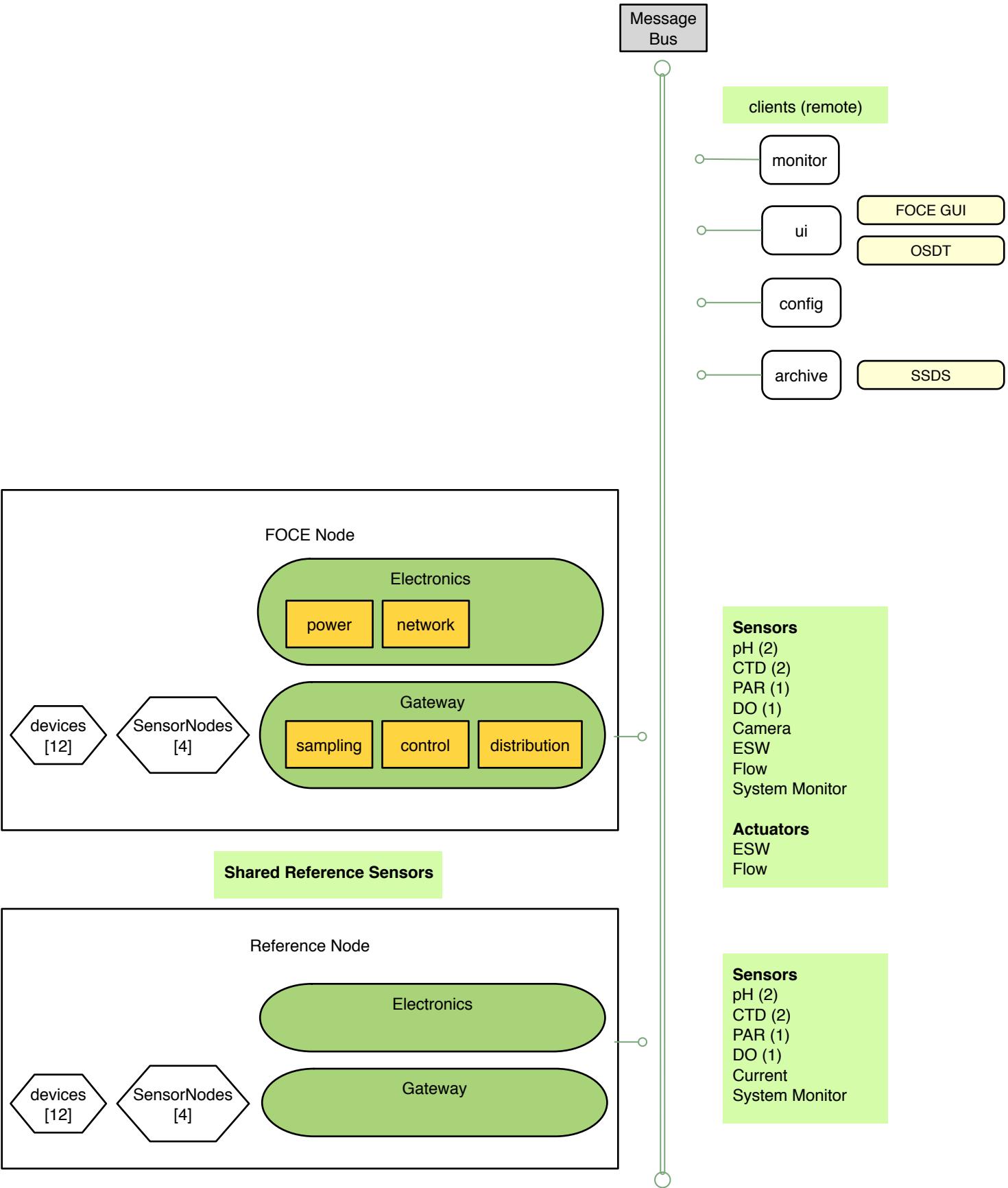
Access

Distribution

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



Gateway Expansion

Message
Bus

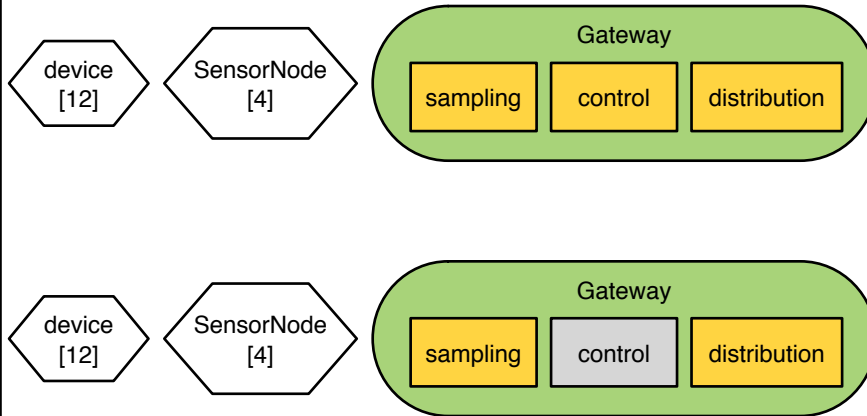
All Gateways are independently configured.

By default, each Gateway collects, logs and distributes data

One gateway per FOCE chamber hosts the closed loop control process

Expansion node sensors may participate in the control algorithm using pub/sub mechanism (control process subscribes to all relevant data streams)

FOCE Node

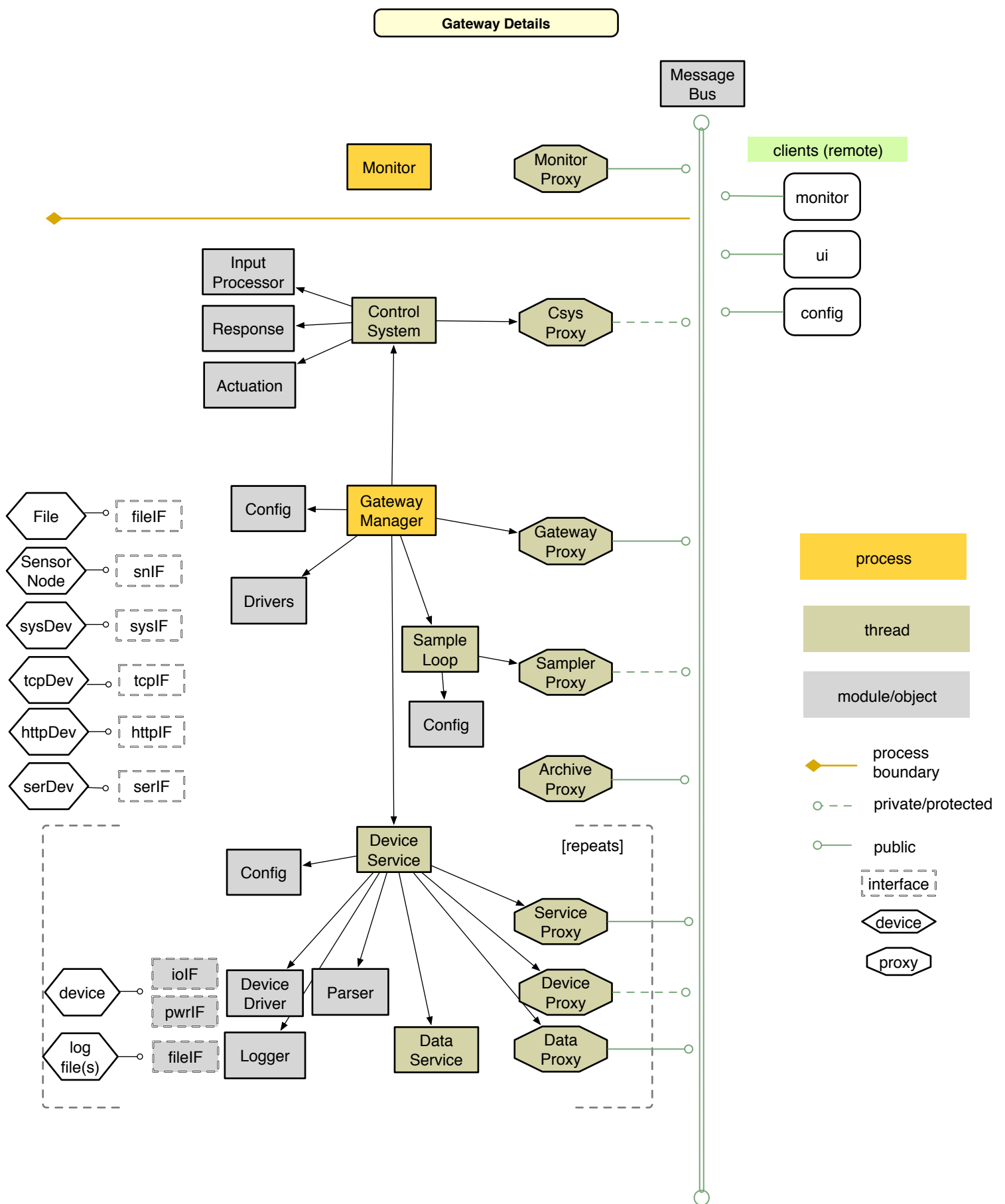


Control Gateway

- One per FOCE node
- May subscribe to expansion gateway data streams

Expansion Gateway

- logs and publishes data from instruments
- Does not run control function



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:

heterogeneous: 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, Monolithic



Heterogeneous, Segmented



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

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

Homogeneous

Advantages

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

Disadvantages

More difficult to present chronology

Hybrid

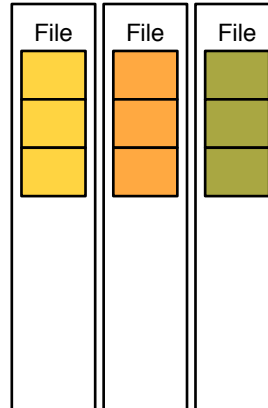
Advantages

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

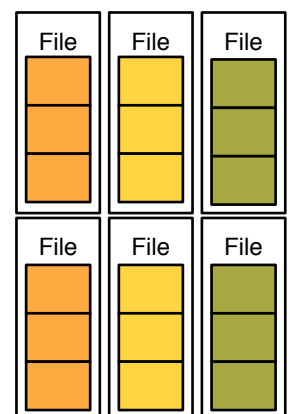
Disadvantages

More difficult to present chronology

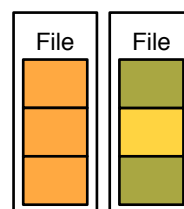
Homogeneous, Monolithic



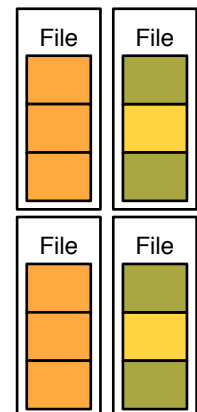
Homogeneous, Segmented



Hybrid, Monolithic



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]

Master Index File



Optional?

Record Index Files



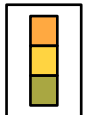
...



Record Archive Files
organized by
producer



...



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?

Legend

fixed length (header)

fixed length (recurring)

variable length

Serialization Format Schema

headerSize[uint16]

fieldCount[uint16]

endianess[byte]

checksum[uint32]

fieldType[uint16]

fieldSize[uint16]

fieldNameSize[uint16]

fieldName[fieldNameSize]

Message Pattern Use Cases

Message Passing

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

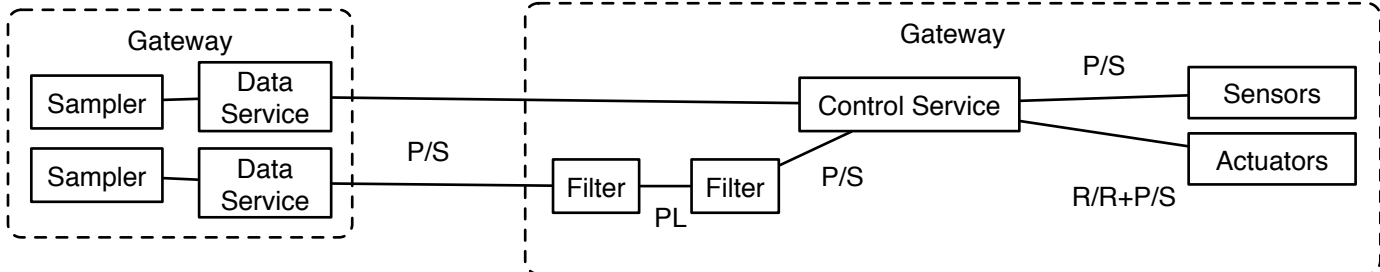
Publish-subscribe (P/S)
Connects a set of publishers to a set of subscribers. This is a data distribution pattern.

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.

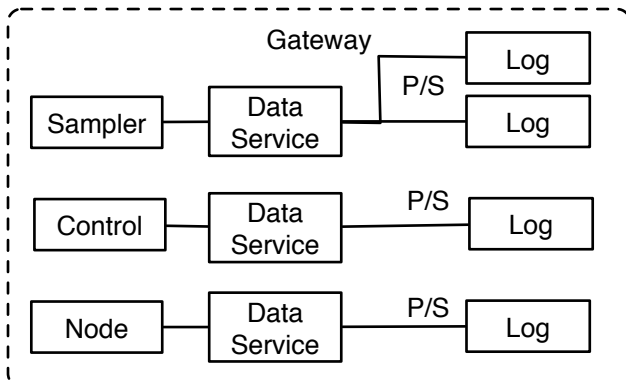
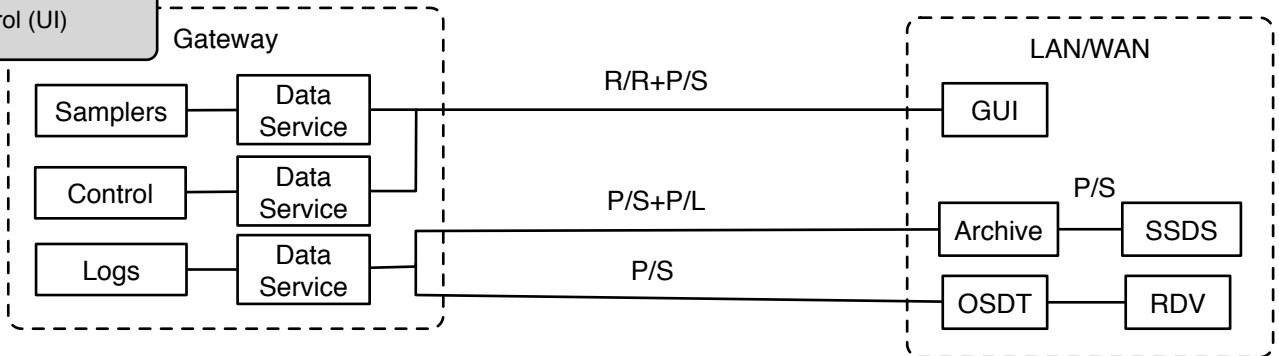
Messaging Use Cases

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)

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

Other candidates

AMQP

- uses a broker
- persistence support
- heavy weight

LCM

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

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

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

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)

MessageBus

TCP/IP

UDP

IPC

Ethernet

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

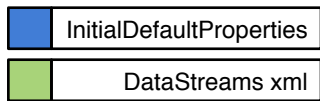
Metadata use case:
Changing (replacing) an instrument

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

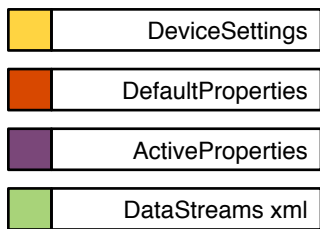
Metadata use case:
Changing an instrument configuration via
console

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

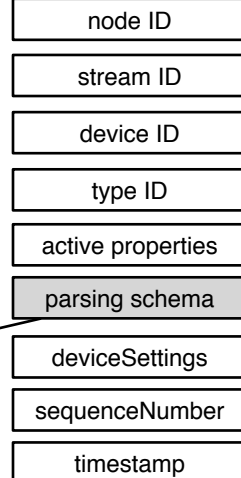


read configuration
update configuration
write configuration



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

Measurement Context



Platform, gateway

functional index (e.g. interior pH)

instrument, service instance

record type (data format)

current service configuration state

data format definition

self-reported device configuration

absolute sequence number

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

source ID

NodeID and StreamID should apply to the entire archive and will not change (ideally, only needs to appear once in archive)

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

reference

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

reference

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[]
}
```

Shore

gateway.config 

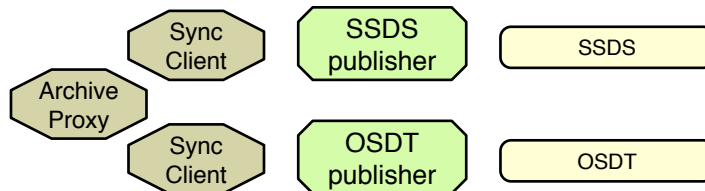
service.config 

service-defaults.config 

service-streams.xml 

service-stream.* 

service-driver 



Gateway

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

gateway.config 

service.config 

service-defaults.config 

service-streams.xml 

service-stream.* 

service-driver 

Best Effort

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

Configuration

Configuration File Features

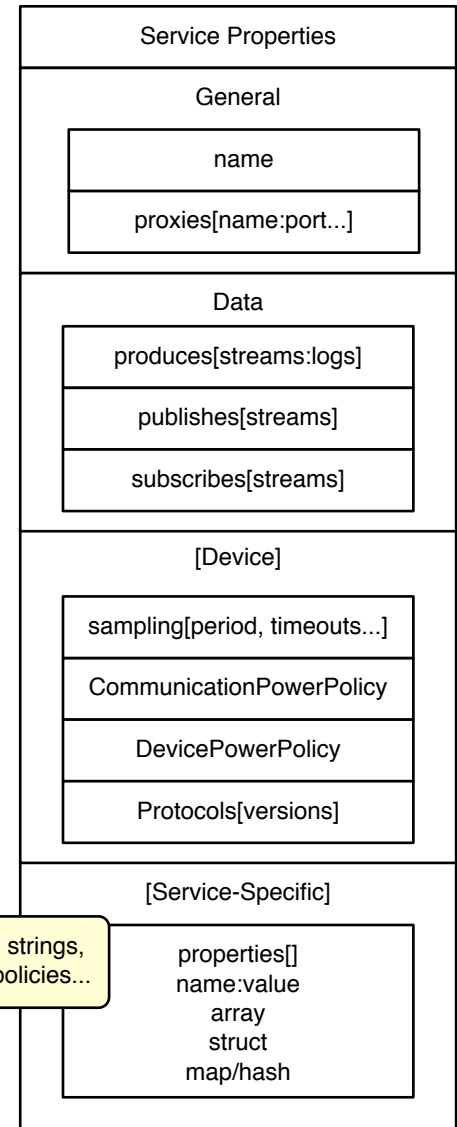
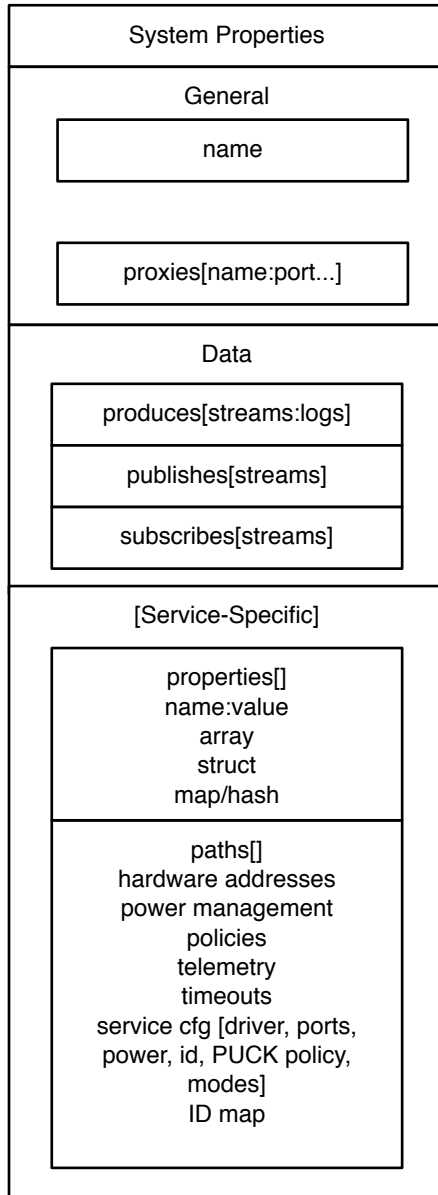
- Human-readable
- Machine-readable
- Persistence/serialization: support read/update/write
- Dynamic refresh (reload while running)

Configuration files persist a configuration data structure

Technology candidates:

JSON, BSON, MessagePack, libConf, libConfuse...

Configuration file contents



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

Things that generate data streams need IDs:

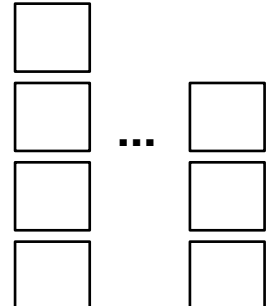
- devices (instruments, system devices)
- virtual devices (control system)
- gateway nodes

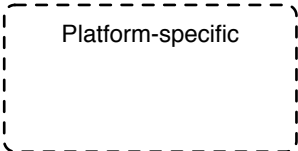
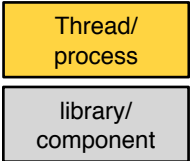
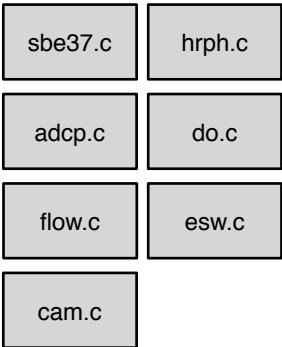
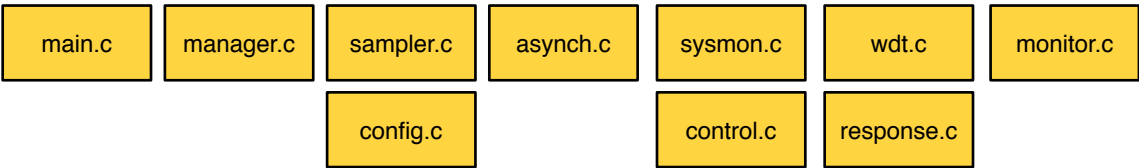
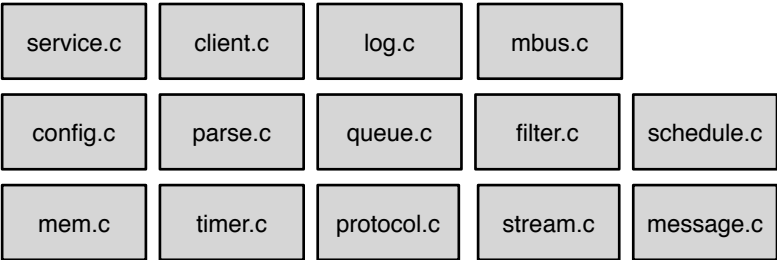
gateway.config

service.config

service-defaults.config

service-streams.xml





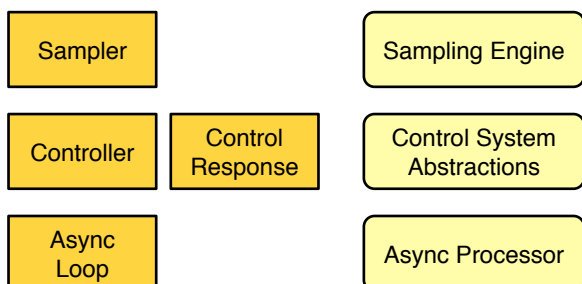
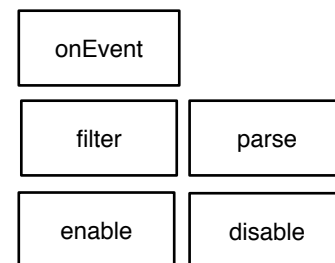
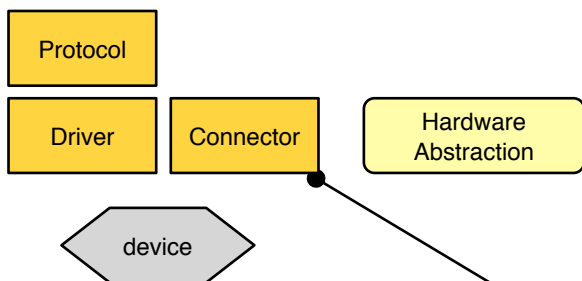
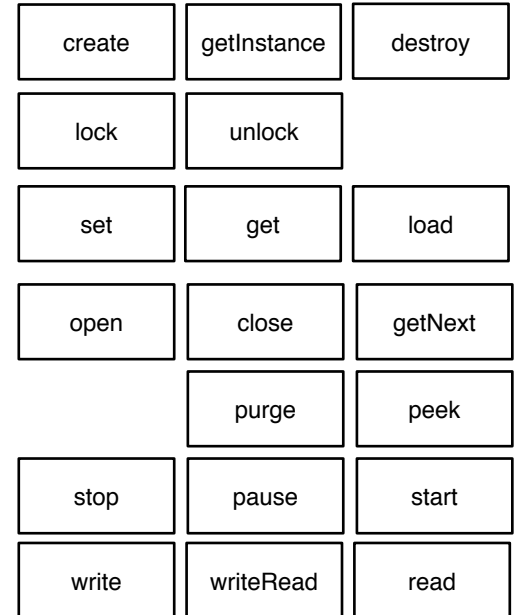
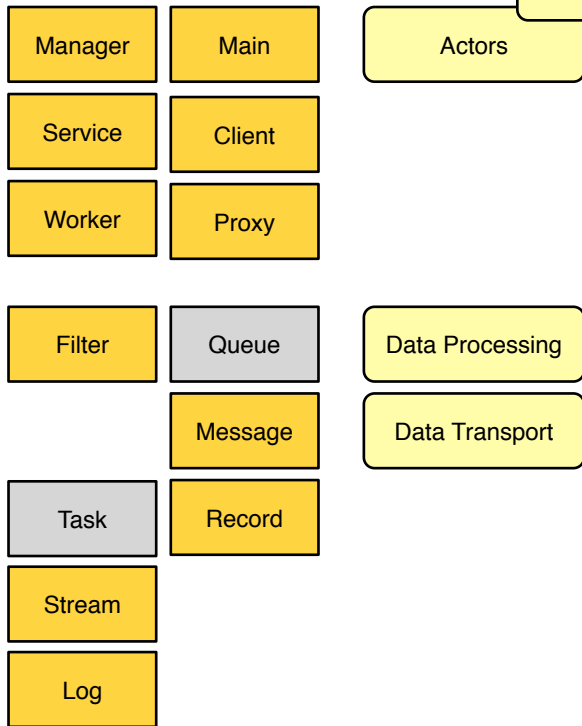
Data Structures

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



Variants, e.g.
SensorNodeSerial,
SensorNodePower,
GPIO,
SPI,
MemoryMap,
TCP/IP,
HTTP,
FileSys

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

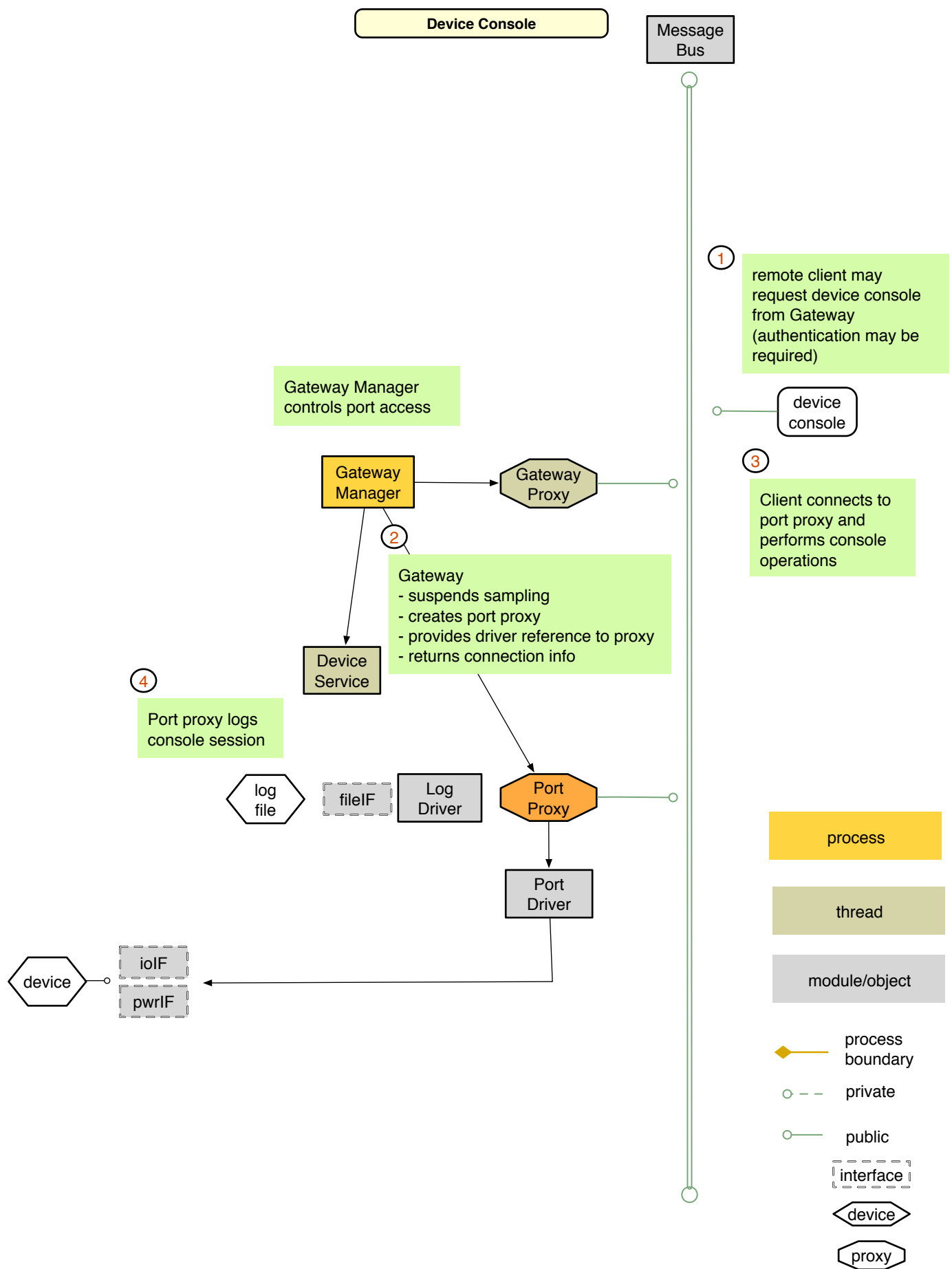
Data

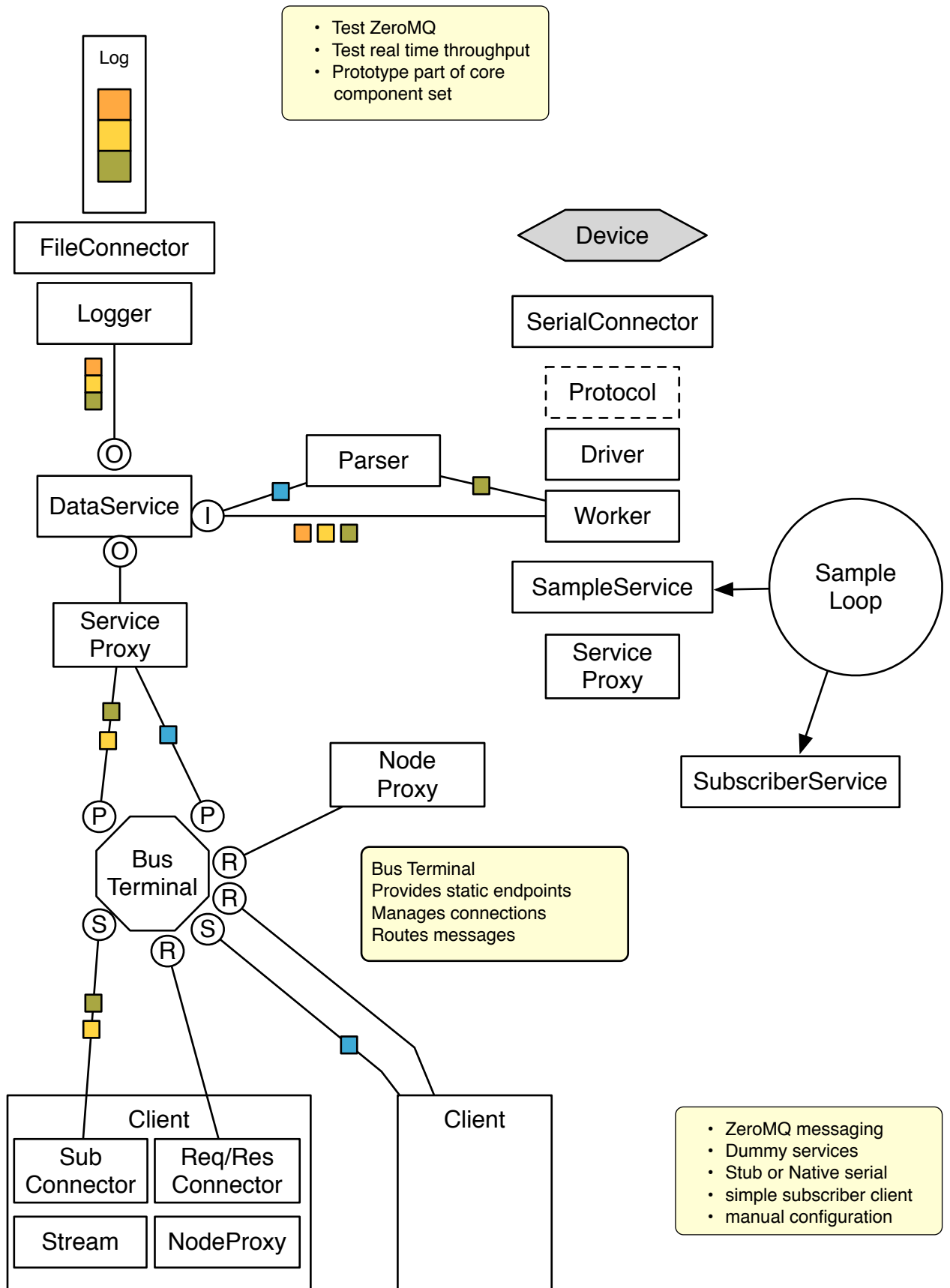
- archive data records
- filtered data

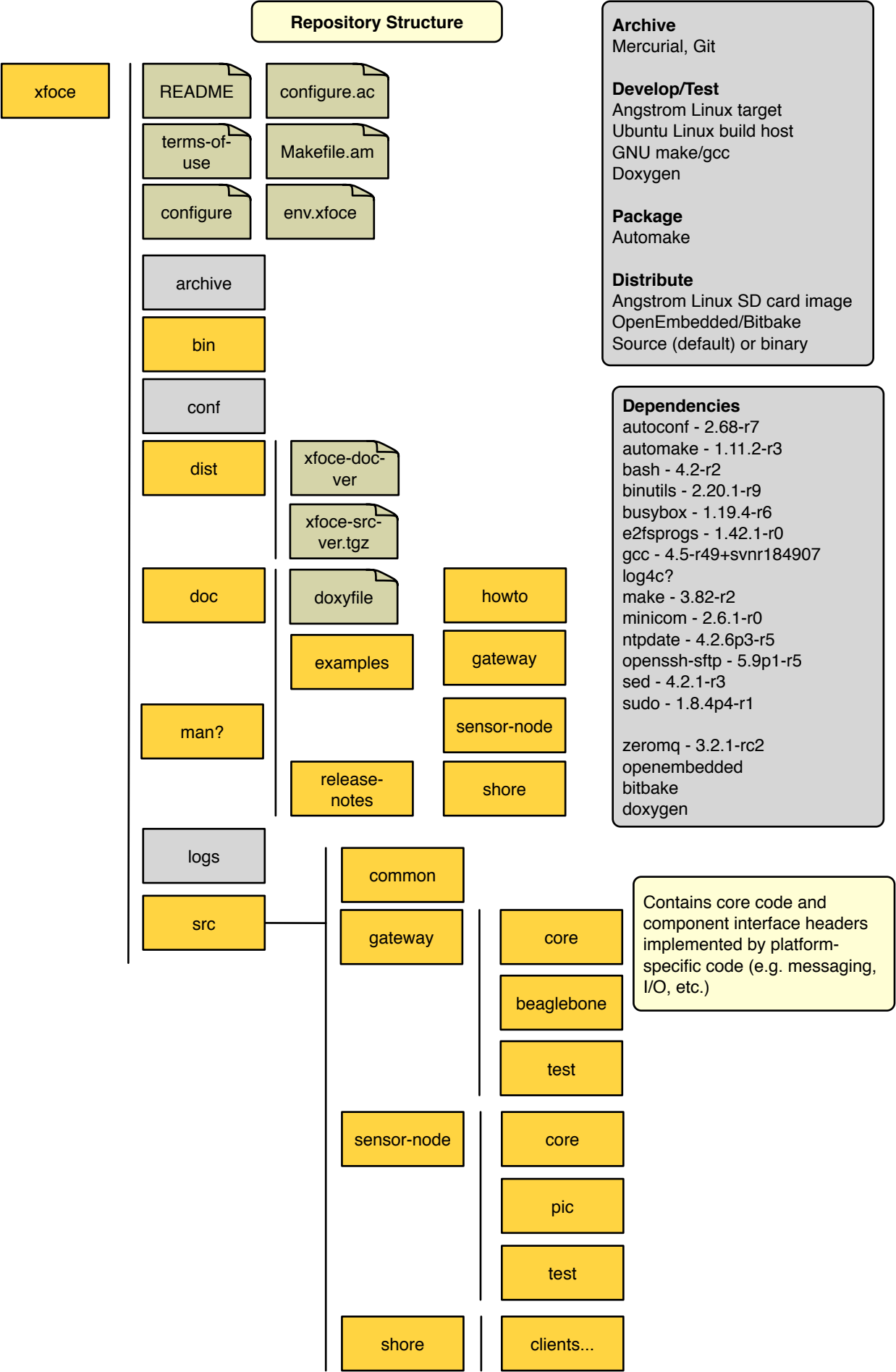
A Pipeline pattern has a notion of "flow control", using PUSH/PULL to govern the flow.

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

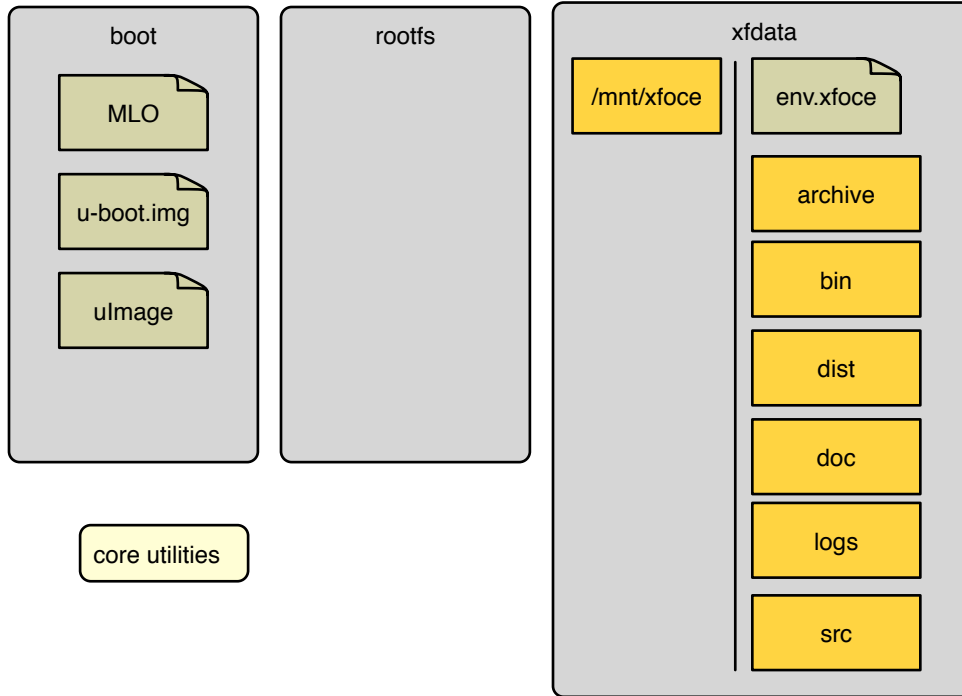






Gateway Installation Environment

MicroSD Card



USB Memory

tmpfs (RAM)

/dev

/tmp

EEPROM

Building xFOCE Gateway Distributions

Users

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)
- Build from repository

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

WWW

xfoce-x.x.img

git/mercurial

xfoce-x.x.tar.gz

xfoce-x.x.opkg

Download
[Edit]
[develop]

Beaglebone Gateway
(ARM Linux Target)

serial

HTTP

ssh

micro
SD

serial

HTTP

ssh

micro
SD

SDCard

PC
(Development Host)

Developers

WWW

xfoce-x.x.img

git/mercurial

xfoce-x.x.tar.gz

xfoce-x.x.opkg

autotools? CMake?
git/mercurial?

Build xfoce software
Create distribution images

Manage repository
Edit
Write SD Image

Mac OSX
(Development Host)

Ubuntu VM
(Development Guest VM)

/mnt/hgfs/
<share>

<share>/xfoce

HTTP

ssh

micro
SD

OE/Bitbake

GNU

editor

repository

dd

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 Beaglebone:

- 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/ familiar automake pattern:

configure, make, make install

Beaglebone Gateway
(ARM Linux Target)

HTTP

ssh

micro
SD

SDCard

Developers

xFOCE Gateway Development Tools (xfoce.dev)

May be distributed as a VMWare virtual disk:

Requirements: VMWare, bitbake recipes

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

Build xfoce software
Create distribution images

Manage repository
Edit
Write SD Image

Mac OSX
(Development Host)

Ubuntu VM
(Development Guest VM)

/mnt/hgfs/
<share>

<share>/xfoce

OE/Bitbake

GNU

editor

repository

dd

HTTP

ssh

micro
SD