Flexible data transport for online reconstruction

M. Al-Turany

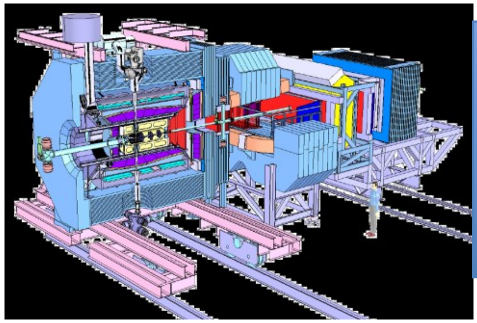
Dennis Klein

A. Rybalchenko

This talk:

- Introduction
  - Design requirement
  - Zero MQ
  - Socket Pattern
- Current Status
- Results

The Online Reconstruction and analysis



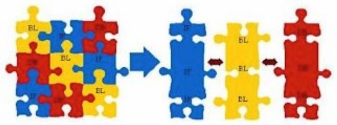
300 GB/s
20M Evt/s

< 1 GB/s
25K Evt/s

> 60 000
CPU-core
or Equivalent
GPU, FPGA, …

How to manage the data flow on such a huge cluster?
How to recover single/multiple processes?
How to monitor it?
……

Design constrains



- Highly flexible: different data paths should be modeled.
- Adaptive: Sub-system are continuously under development and improvement
- Should work for simulated and real data: developing and debugging the algorithms
- It should support all possible hardware where the algorithms could run (CPU, GPU, FPGA)
- It has to scale to any size! With minimum or ideally no effort.

Before Re-inventing the Wheel

- What is available on the market and in the community?
  - ALICE, ATLAS, CMS, LHCb, …
  - Financial and weather application have also huge data to deal with
- Do we intend to separate online and offline?
- Multithreaded concept or a message queue based one?
  - Message based systems allow us to decouple producers from consumers.
  - We can spread the work to be done over several processes and machines.
  - We can manage/upgrade/move around programs (processes) independently of each other.

ØMQ (ZeroMQ) Available since 2011

- A socket library that acts as a concurrency framework.
- Faster than TCP, for clustered products and supercomputing.
- Carries messages across inproc, IPC, TCP, and multicast.
- Connect N-to-N via fanout, pubsub, pipeline, request-reply.
- A synch I/O for scalable multicore message-passing apps.
- 30+ languages including C, C++, Java, .NET, Python.
- Most OSes including Linux, Windows, OS X, PPC405/PPC440.
- Large and active open source community.
- LGPL free software with full commercial support from iMatix.

Zero in ØMQ

Originally the zero in ØMQ was meant as "zero broker" and (as close to) "zero latency" (as possible). In the meantime it has come to cover different goals:

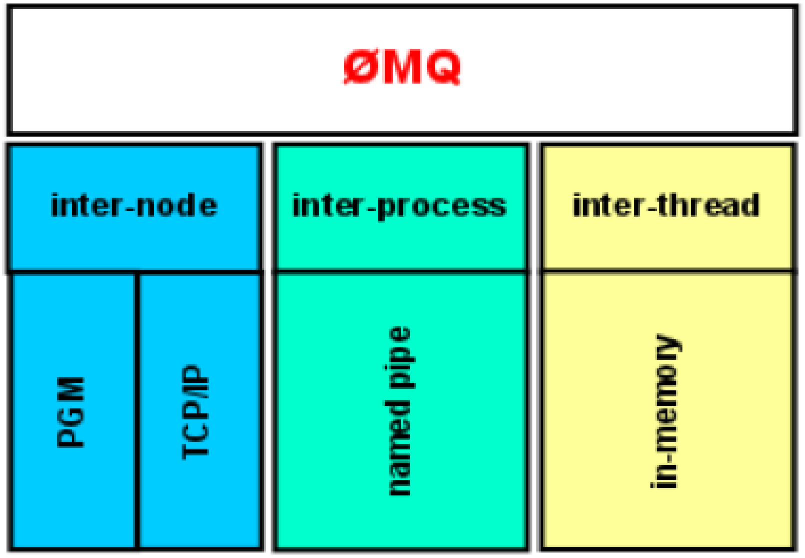
- zero administration,
- zero cost,
- zero waste.

More generally, "zero” refers to the culture of minimalism that permeates the project.

Adding power by removing complexity rather than exposing new functionality.

ZeroMQ sockets provide efficient transport options

- Inter-thread
- Inter-process
- Inter-node
  - which is really just inter-process across nodes communication



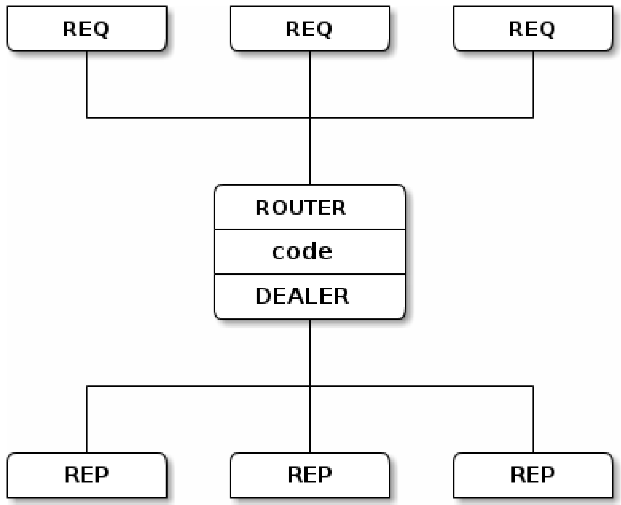
PMG : Pragmatic General Multicast (a reliable multicast protocol)
Named Pipe: Piece of random access memory (RAM) managed by the operating system and exposed to programs through a file descriptor and a named mount point in the file system. It behaves as a first in first out (FIFO) buffer

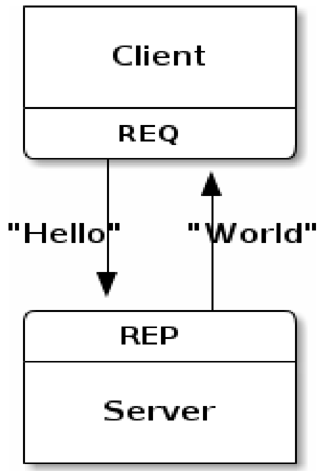
The built-in core ØMQ patterns are:

- **Request-reply**, which connects a set of clients to a set of services. This is a remote procedure call and task distribution pattern.
- **Publish-subscribe**, which connects a set of publishers to a set of subscribers. This is a data distribution pattern.
- **Pipeline**, which 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.
- **Exclusive pair**, which connect two sockets exclusively

Request-Reply Pattern

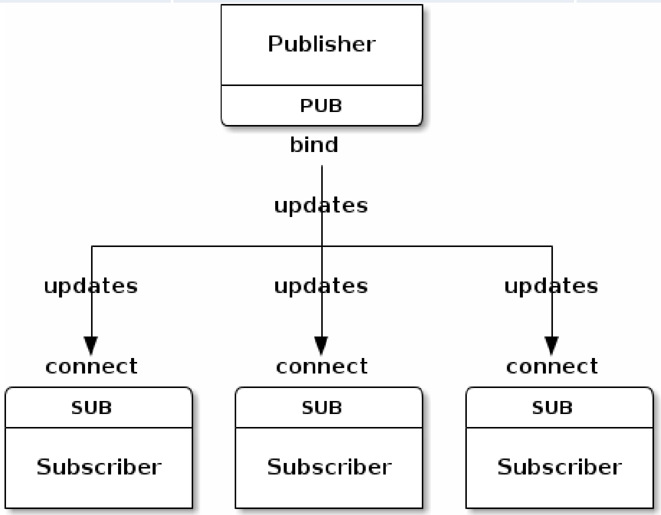
Socket type	REQ	REP
Compatible peer sockets	REP, ROUTER	REQ, DEALER
Direction	Bidirectional	Bidirectional
Send/receive pattern	Send, Receive	Send, Receive
Outgoing routing strategy	Round-robin	Last peer
Incoming routing strategy	Last peer	Fair-queued
Action in mute state	Block	Drop

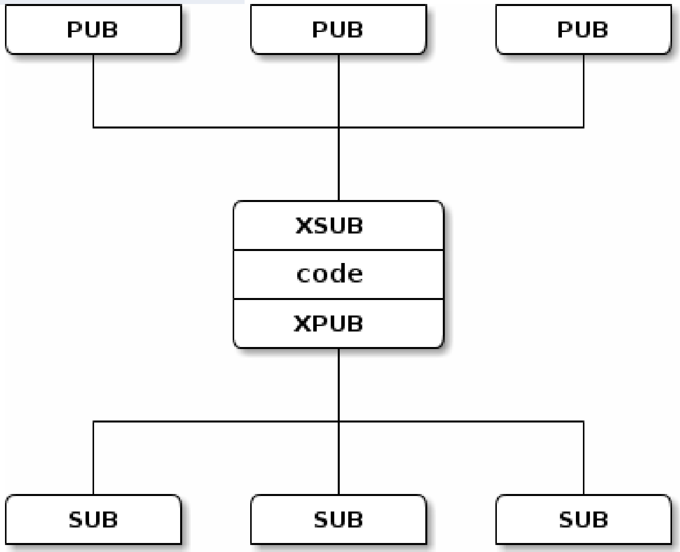




Publish-Subscribe Pattern

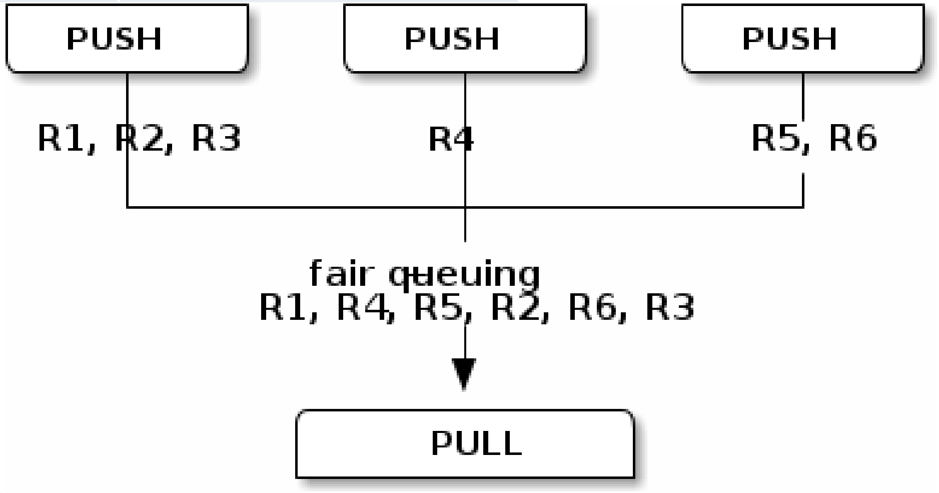
Socket type	PUB	SUB
Compatible peer sockets	SUB, XSUB	PUB, XPUB
Direction	Unidirectional	Unidirectional
Send/receive pattern	Send Only	Receive only
Outgoing routing strategy	Fan-out	N/A
Incoming routing strategy	N/A	Fair-queued
Action in mute state	Drop	Drop





Pipeline Pattern

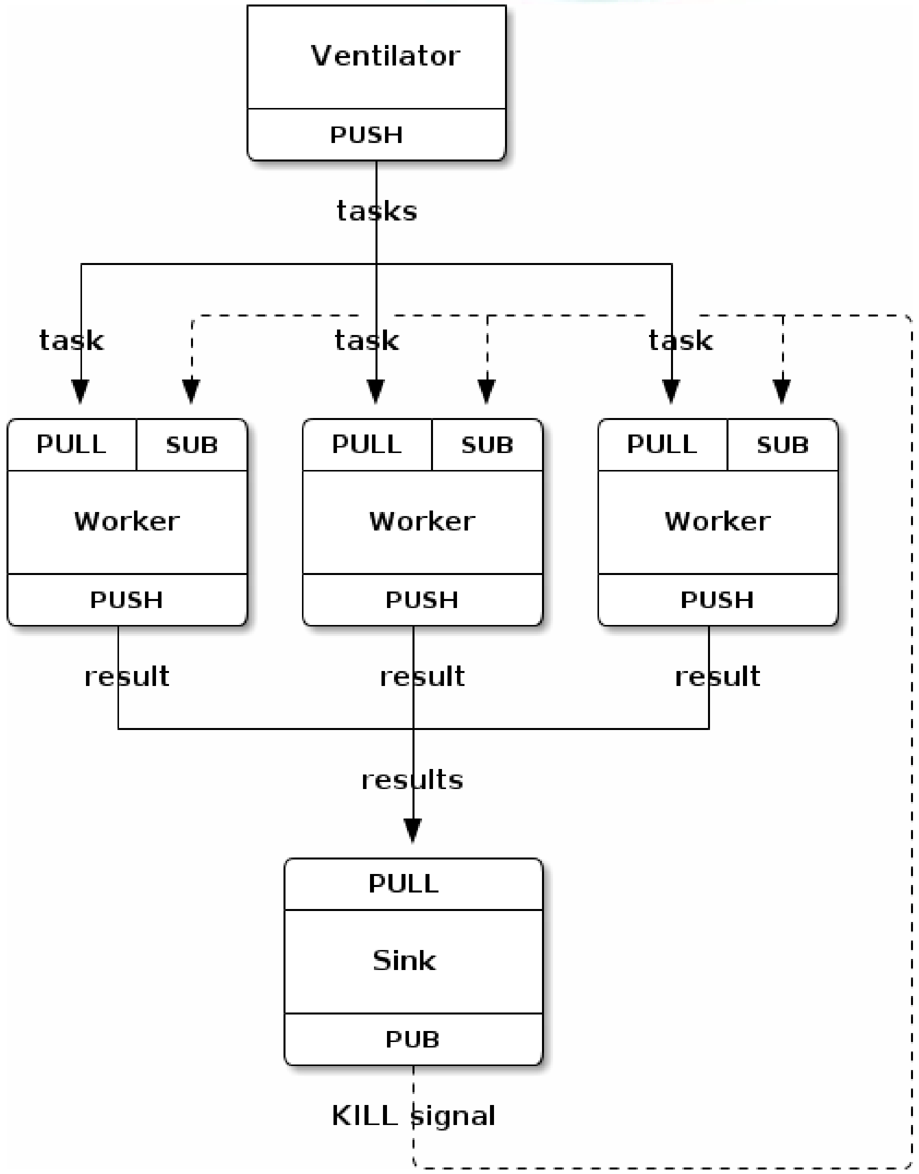
Socket type	**PUSH**	**PULL**
Compatible peer sockets	PULL	PUSH
Direction	Unidirectional	Unidirectional
Send/receive pattern	Send Only	Receive only
Outgoing routing strategy	Round-Robin	N/A
Incoming routing strategy	N/A	Fair-queued
Action in mute state	Block	Block



Example of sending control commands

- A worker process can manages two sockets (a PULL socket getting tasks, and a SUB socket getting control commands)

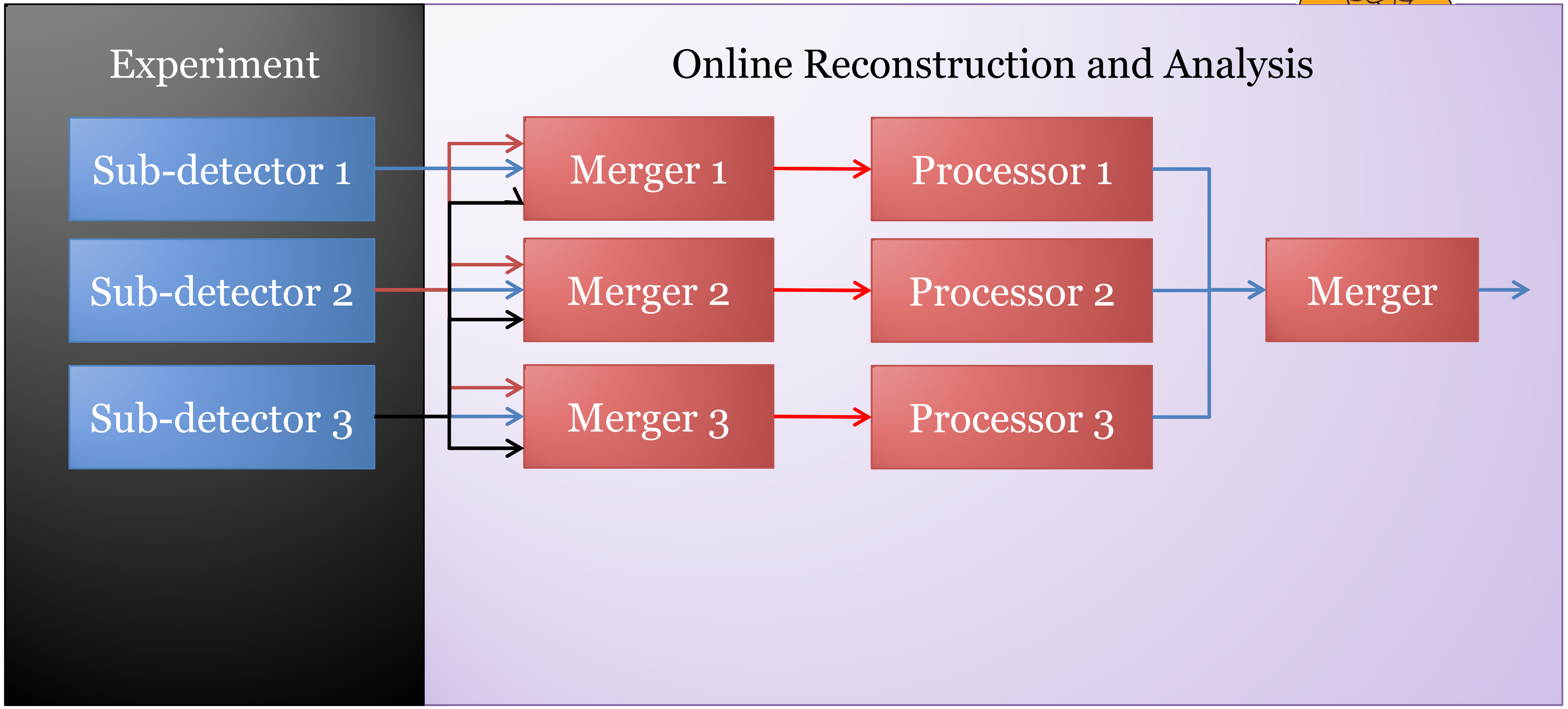
- Could be very useful for calibration and alignment parameter



Data Transfer Framework as Extension to FairRoot! Why?

- Modeling the pipeline processing within the online analysis
- Enable concurrency in FairRoot for offline analysis
- Reliable and efficient data transport through message queuing technology
- The long term plan is to have the same framework for online and offline

Data flow example

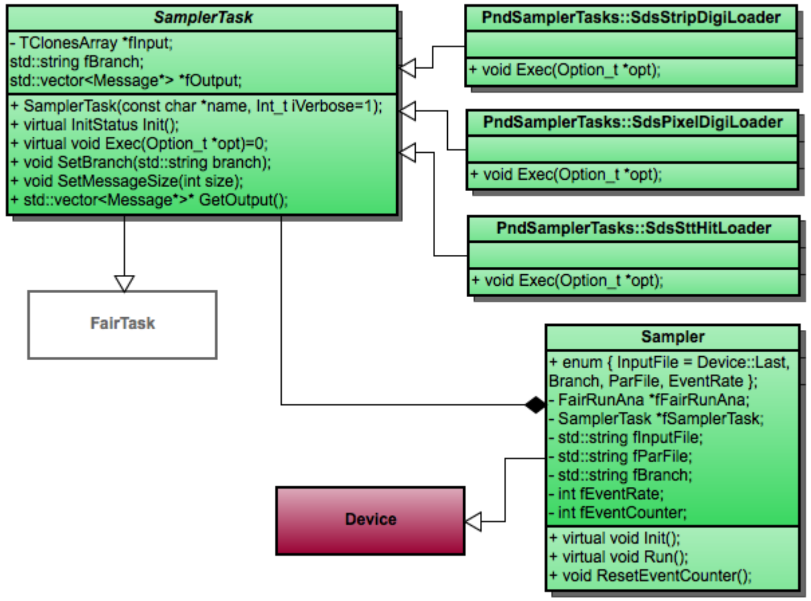


Current Status

- The Framework deliver some components which can be connected to each other in order to construct a processing pipeline.
- All component share a common base called Device (ZeroMQ Class).
- All devices are grouped by three categories:
  - Source: Sampler
  - Message-based Processor:
    - Sink, BalancedStandaloneSplitter, StandaloneMerger, Buffer
  - Content-based Processor: Processor

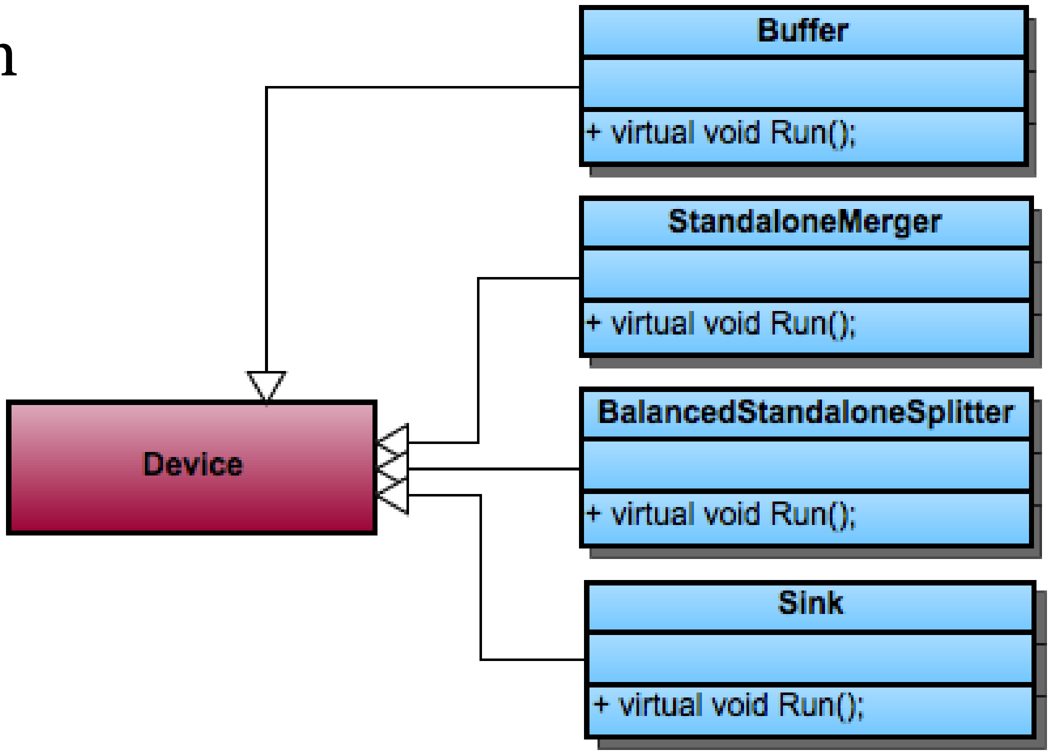
Sampler

- Devices with no inputs are categorized as sources
- During RUN state the sampler loops infinitely over the loaded events and send them through the output socket.
- A variable event rate limiter has been implemented to control the sending speed


SamplerTask
- TClonesArray *fInput;
std::string fBranch;
std::vector<Message*> *fOutput;
+ SamplerTask(const char *name, Int_t iVerbose=1);
+ virtual InitStatus Init();
+ virtual void Exec(Option_t *opt)=0;
+ void SetBranch(std::string branch);
+ void SetMessageSize(int size);
+ std::vector<Message*>* GetOutput();
PndSamplerTasks::SdsStripDigiLoader
+ void Exec(Option_t *opt);
PndSamplerTasks::SdsPixelDigiLoader
+ void Exec(Option_t *opt);
PndSamplerTasks::SdsSttHitLoader
+ void Exec(Option_t *opt);
FairTask
Sampler
+ enum { InputFile = Device::Last, Branch, ParFile, EventRate };
- FairRunAna *fFairRunAna;
- SamplerTask *fSamplerTask;
- std::string fInputFile;
- std::string fParFile;
- std::string fBranch;
- int fEventRate;
- int fEventCounter;
+ virtual void Init();
+ virtual void Run();
+ void ResetEventCounter();
Device

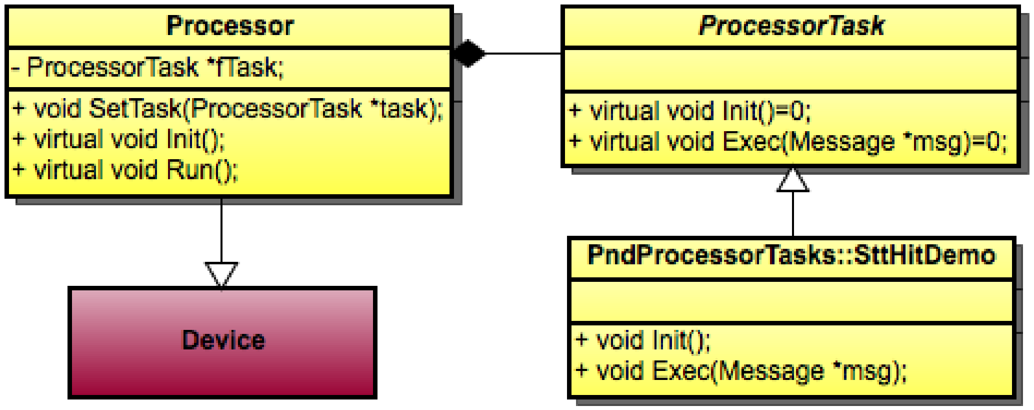
Message-based Processor

- All message-based processors inherit from Device and operate on messages without interpreting their content.
- Four message-based processors have been implemented so far

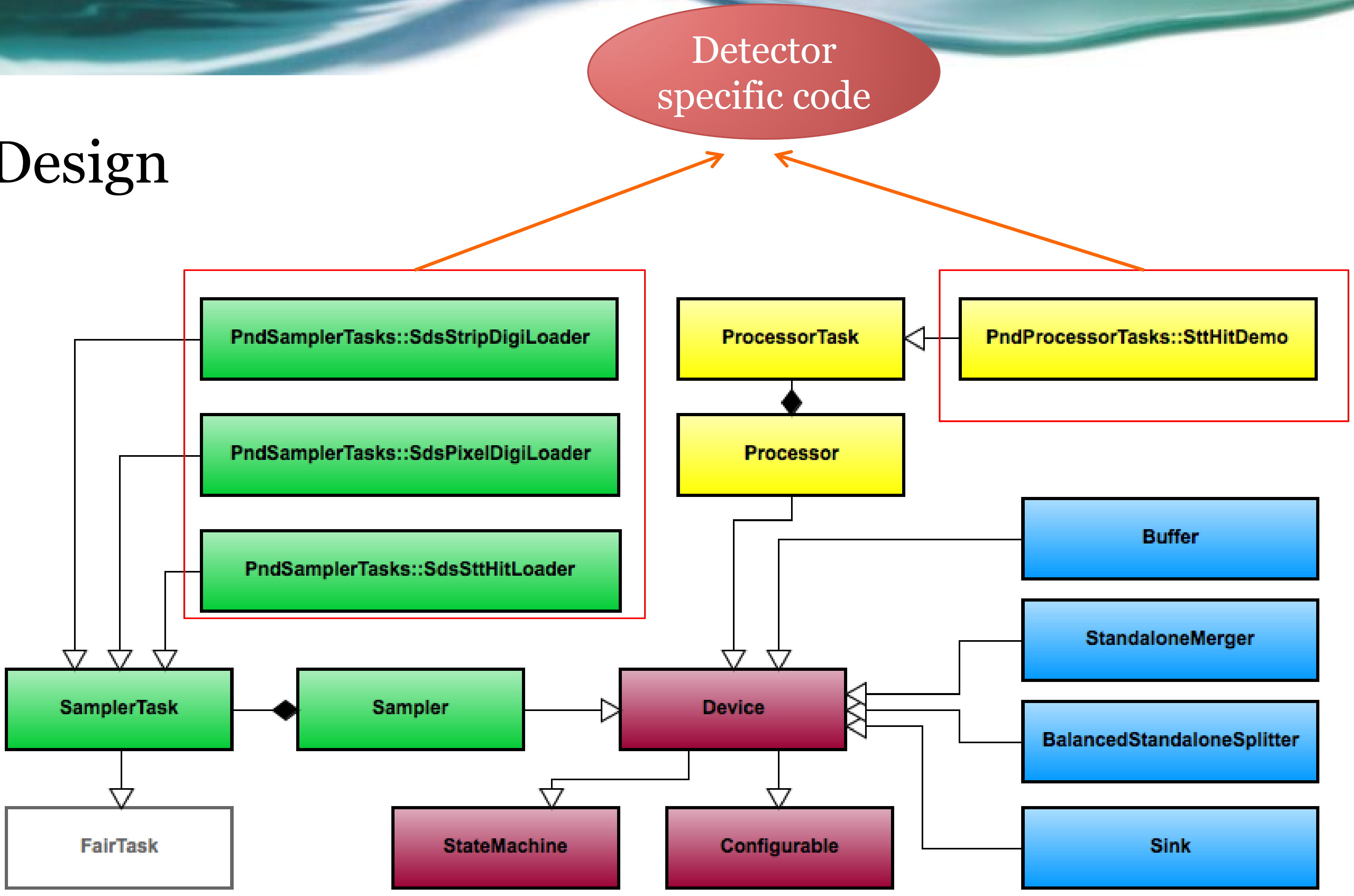


Content-based Processor

- The Processor device has one input and one output socket.
- A task is meant for accessing and potentially changing the message content.

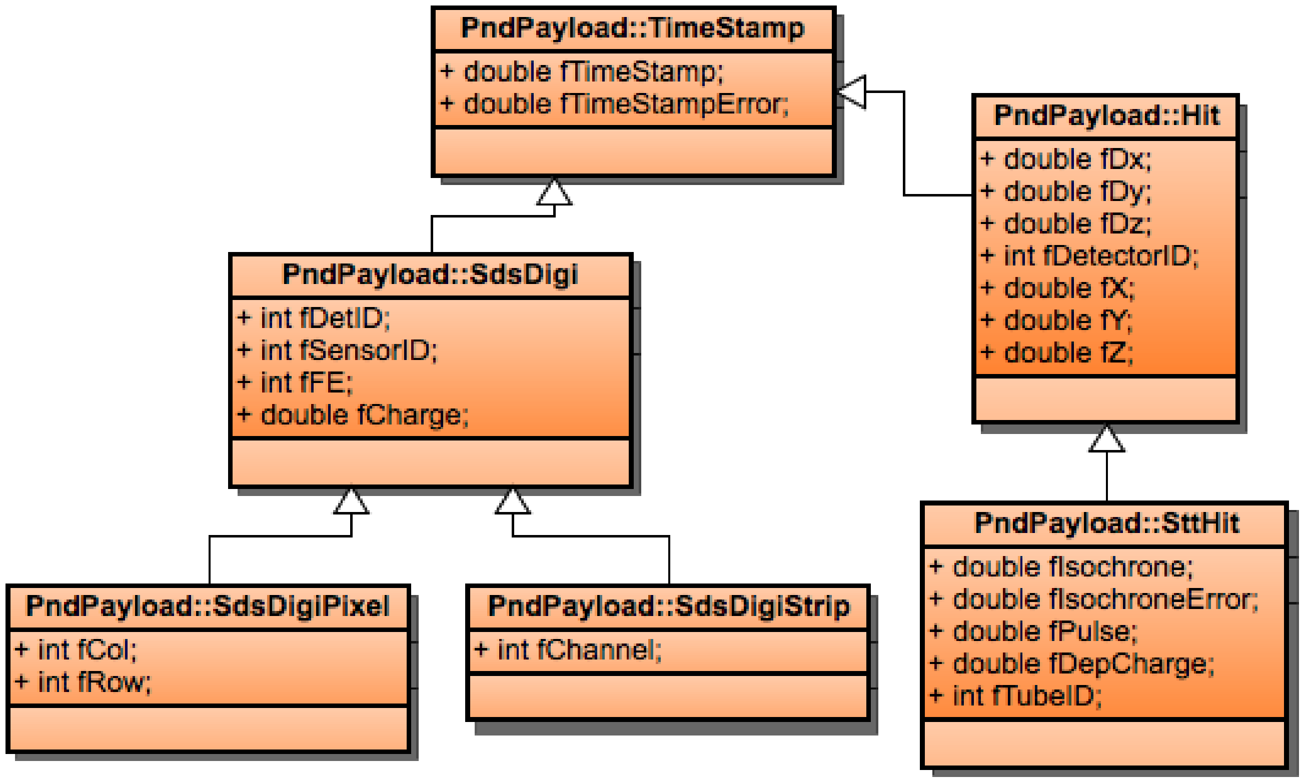


Design



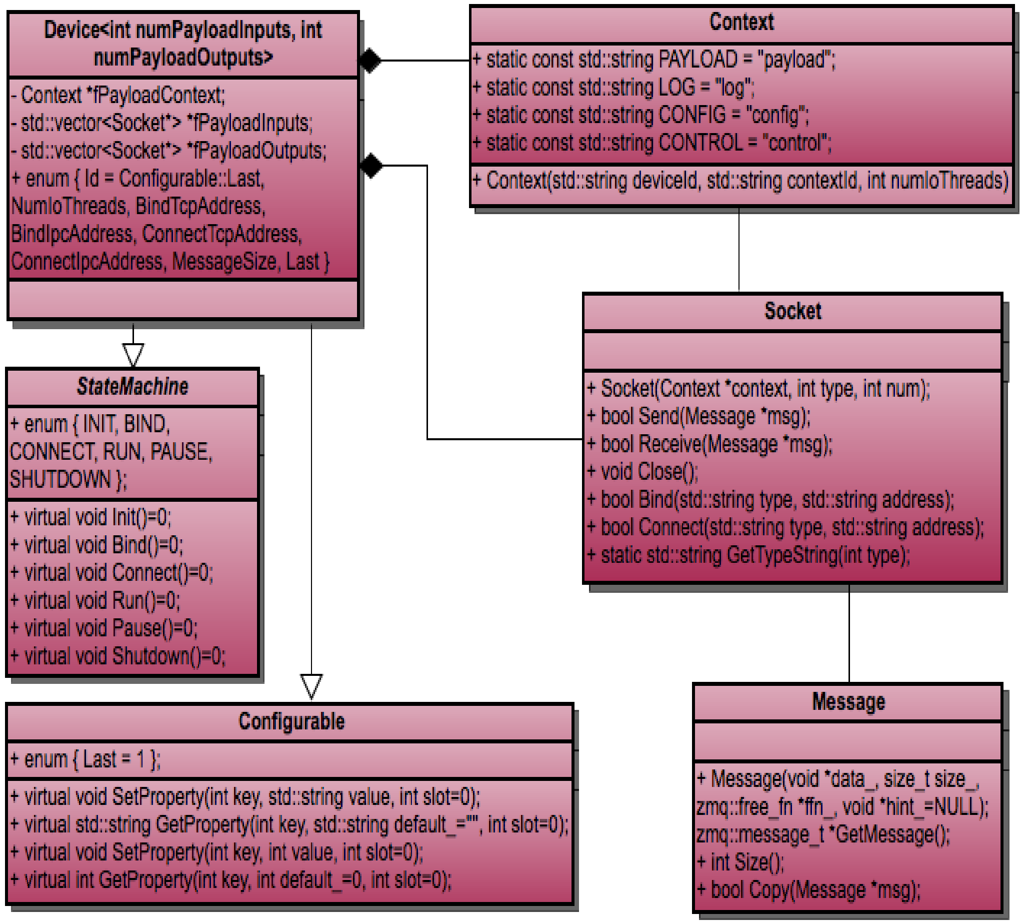
Detector specific code
PndSamplerTasks::SdsStripDigiLoader
PndSamplerTasks::SdsPixelDigiLoader
PndSamplerTasks::SdsSttHitLoader
ProcessorTask
PndProcessorTasks::SttHitDemo
Processor
Buffer
StandaloneMerger
BalancedStandaloneSplitter
Sink
SamplerTask
Sampler
Device
FairTask
StateMachine
Configurable

New simple classes without ROOT are used in the Sampler (This enable us to use non-ROOT clients) and reduce the messages size.



Device

- Each processing stage of a pipeline is occupied by a process which executes an instance of the Device class



Device<int numPayloadInputs, int numPayloadOutputs>
- Context *fPayloadContext;
- std::vector<Socket*> *fPayloadInputs;
- std::vector<Socket*> *fPayloadOutputs;
+ enum { Id = Configurable::Last, NumIoThreads, BindTcpAddress, BindIpcAddress, ConnectTcpAddress, ConnectIpcAddress, MessageSize, Last }
Context
+ static const std::string PAYLOAD = "payload";
+ static const std::string LOG = "log";
+ static const std::string CONFIG = "config";
+ static const std::string CONTROL = "control";
+ Context(std::string deviceId, std::string contextId, int numIoThreads)
StateMachine
+ enum { INIT, BIND, CONNECT, RUN, PAUSE, SHUTDOWN };
+ virtual void Init()=0;
+ virtual void Bind()=0;
+ virtual void Connect()=0;
+ virtual void Run()=0;
+ virtual void Pause()=0;
+ virtual void Shutdown()=0;
Socket
+ Socket(Context *context, int type, int num);
+ bool Send(Message *msg);
+ bool Receive(Message *msg);
+ void Close();
+ bool Bind(std::string type, std::string address);
+ bool Connect(std::string type, std::string address);
+ static std::string GetTypeString(int type);
Configurable
+ enum { Last = 1 };
+ virtual void SetProperty(int key, std::string value, int slot=0);
+ virtual std::string GetProperty(int key, std::string default_="", int slot=0);
+ virtual void SetProperty(int key, int value, int slot=0);
+ virtual int GetProperty(int key, int default_=0, int slot=0);
Message
+ Message(void *data_, size_t size_, zmq::free_fn *ffn_, void *hint_=NULL);
zmq::message_t *GetMessage();
+ int Size();
+ bool Copy(Message *msg);

Message format (Protocol)

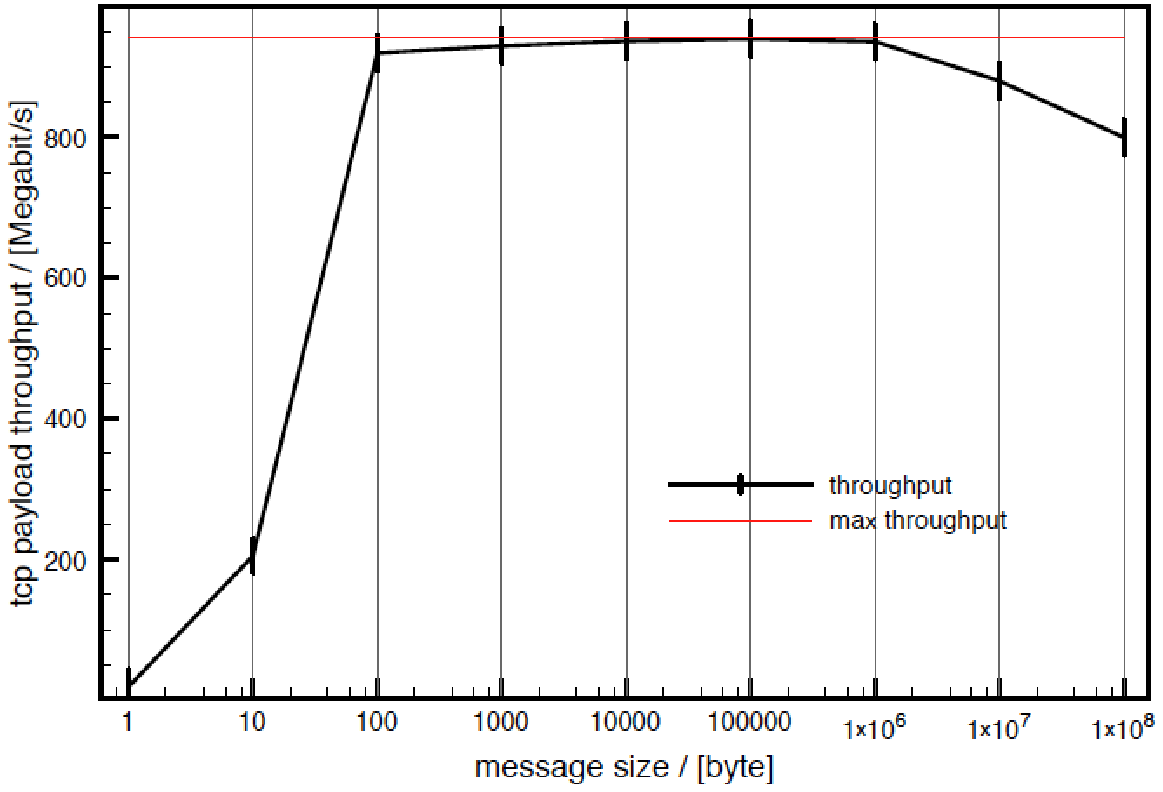
- Potentially any content-based processor or any source can change the application protocol.
- The framework provides a generic Message class that works with any arbitrary and continuous junk of memory.
- One has to pass a pointer to the memory buffer and the size in bytes, and can optionally pass a function pointer to a destructor, which will be called once the message object is discarded.

Test setup and results

Four identical nodes were connected to a GigabitEthernet switch for testing

- CPU: Intel Xeon L5506 @ 2.13 GHz
- Memory: 24 GiB, 6 4 GiB
- Network: Intel 82574L Gigabit Network Connection, speed=1Gbit/s
- Operating system GNU/Linux 3.2.32-1 x86_64, Debian 7.0
- ZeroMQ 3.2.0
- FairRoot PandaRoot oct12 release,
- Fairsoft development version from 18.12.2012

ZeroMQ reaches the upper TCP throughput for message sizes larger than a hundred bytes.



The last two measured values at 10 MB and 100 MB message size appear to be less efficient due to non-fractional output of the benchmark program. The near to maximum throughput for these last two values has been confirmed by monitoring the throughput with the linux tool iftop

Results

- TCP throughput of 117.6MB/s was measured which is very close to the theoretical limit of 117.7 MB/s for the TCP/IPv4/GigabitEthernet stack.
- This was achieved using the Linux default values for the Ethernet MTU (1500 B) and TCP buffer size (85.3 KB).

- The throughput for the named pipe transport between two devices on one node has been measured around 1.7 GB/s

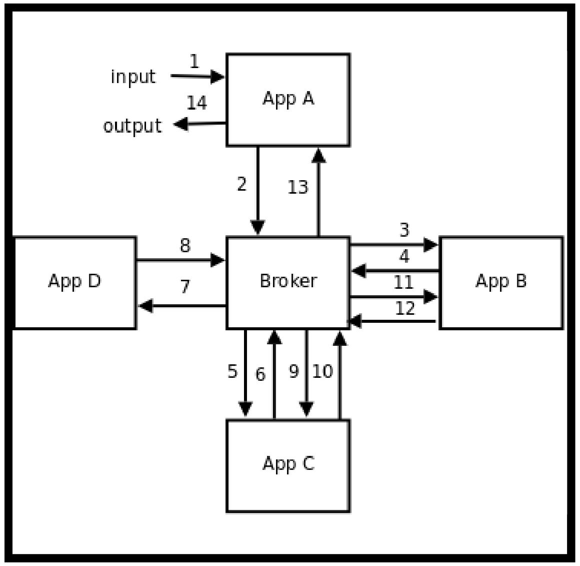
Thanks!

Don't tell me you climbed **all the way up here** before trying **Google??**
Pirillo & Fitz
© BLAUGH.COM · FITZ & PIRILLO · SAVE 10% AT GODADDY BY USING THE COUPON CODE "BLAUGH"

Backup slides

Broker

- Architecture of most messaging systems is distinctive by the messaging server ("broker") in the middle.
- Every application is connected to the central broker.
- No application is speaking directly to the other application. All the communication is passed through the broker.



Advantages

- Applications don't have to have any idea about location of other applications. The only address they need is the network address of the broker.
- Message sender and message receiver lifetimes don't have to overlap. Sender application can push messages to the broker and terminate. The messages will be available for the receiver application any time later.
- Broker model is to some extent resistant to the application failure. So, if the application is buggy and prone to failure, the messages that are already in the broker will be retained even if the application fails.

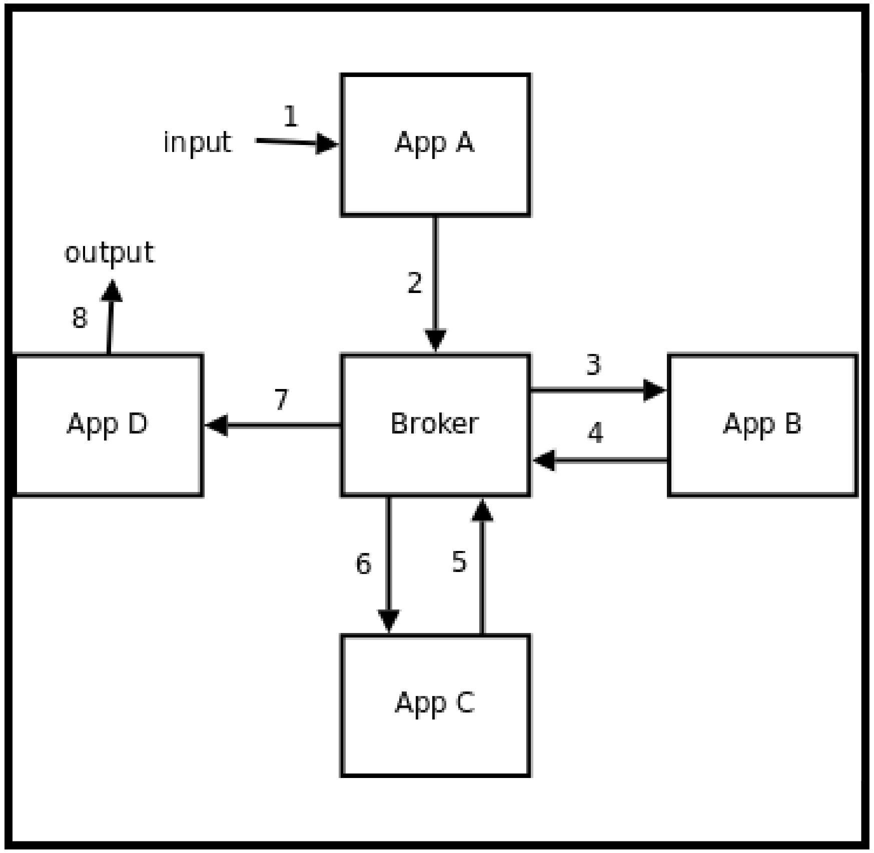
Drawbacks

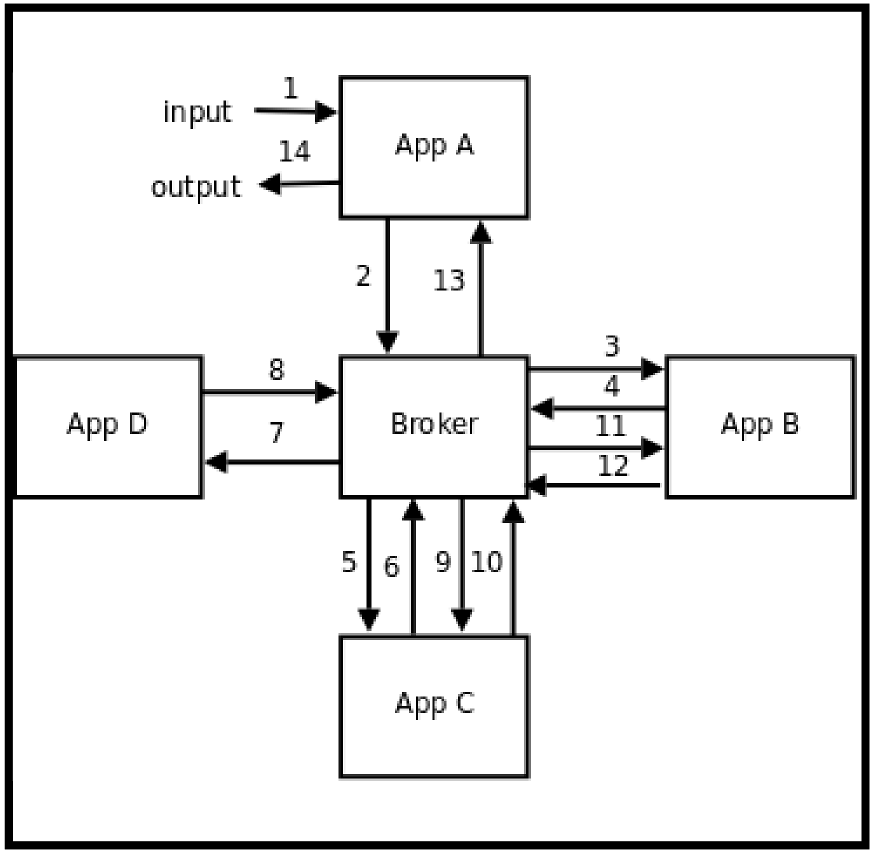
- It requires excessive amount of network communication.
- The fact that all the messages have to be passed through the broker can result in broker turning out to be the bottleneck of the whole system.

Broker pattern

pipelined fashion

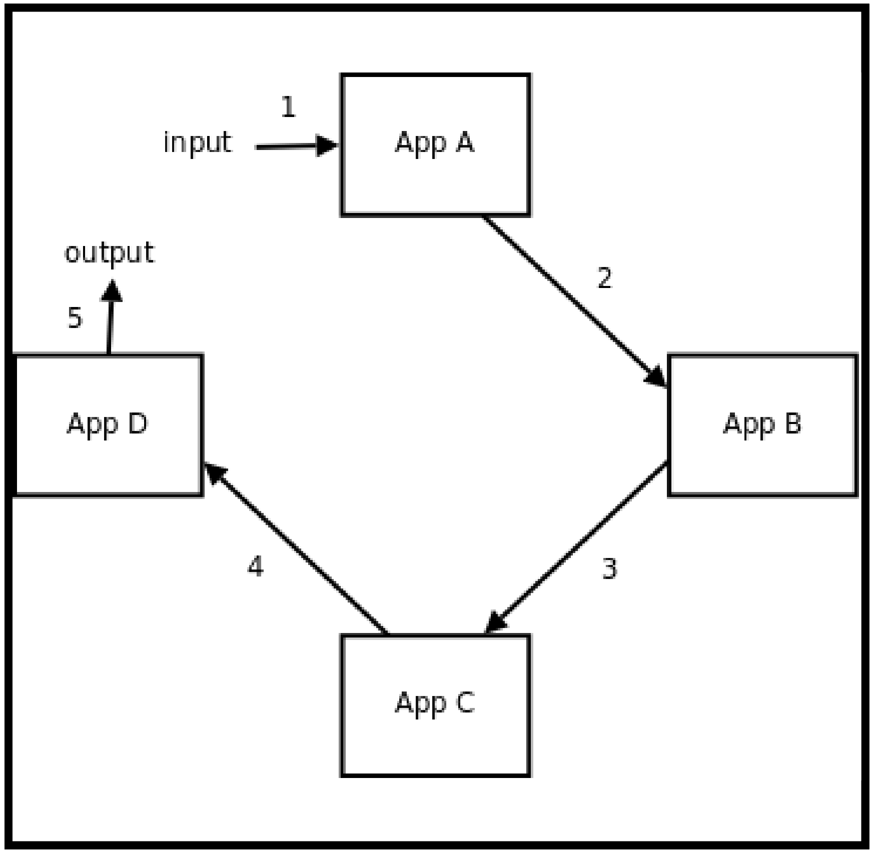
service-oriented architectures (SOA)



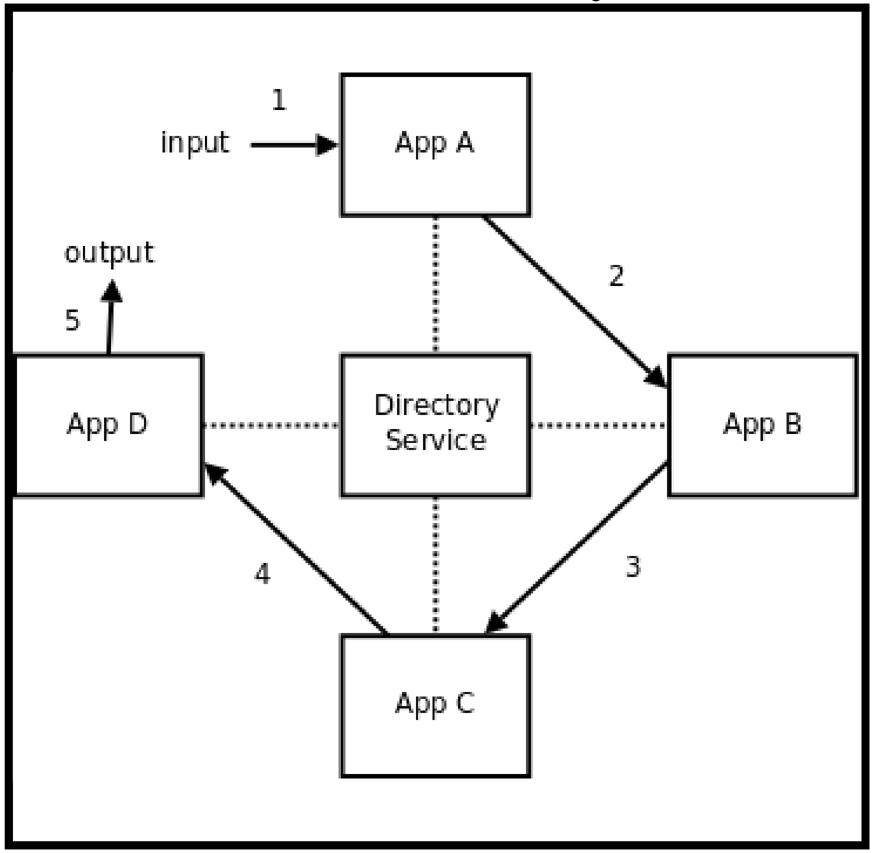


Examples of No Broker model in ZeroMQ

No Broker

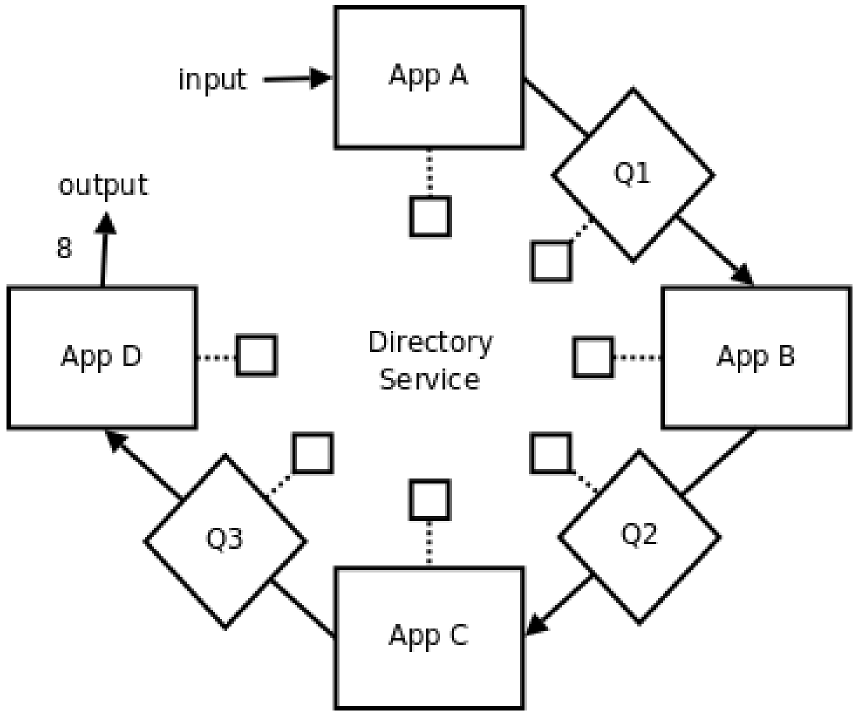


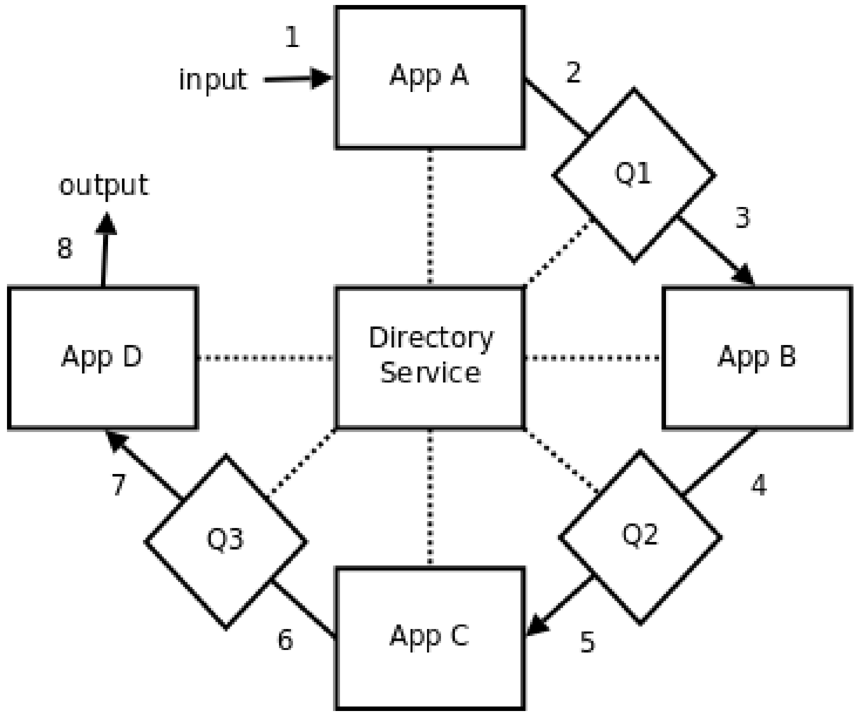
Broker as a Directory Service



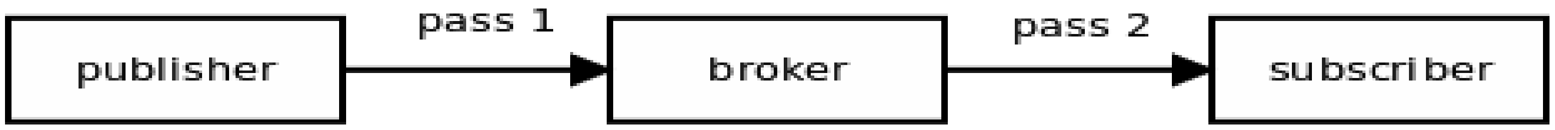
More Models

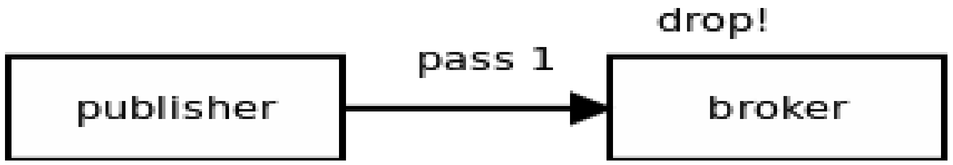
Distributed directory service **Distributed broker**





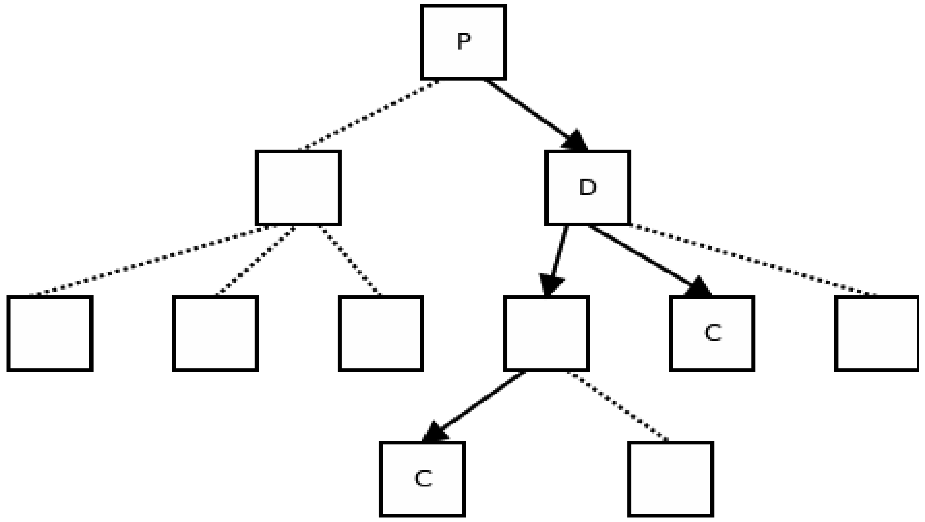
PUB/SUB (publish/subscribe)





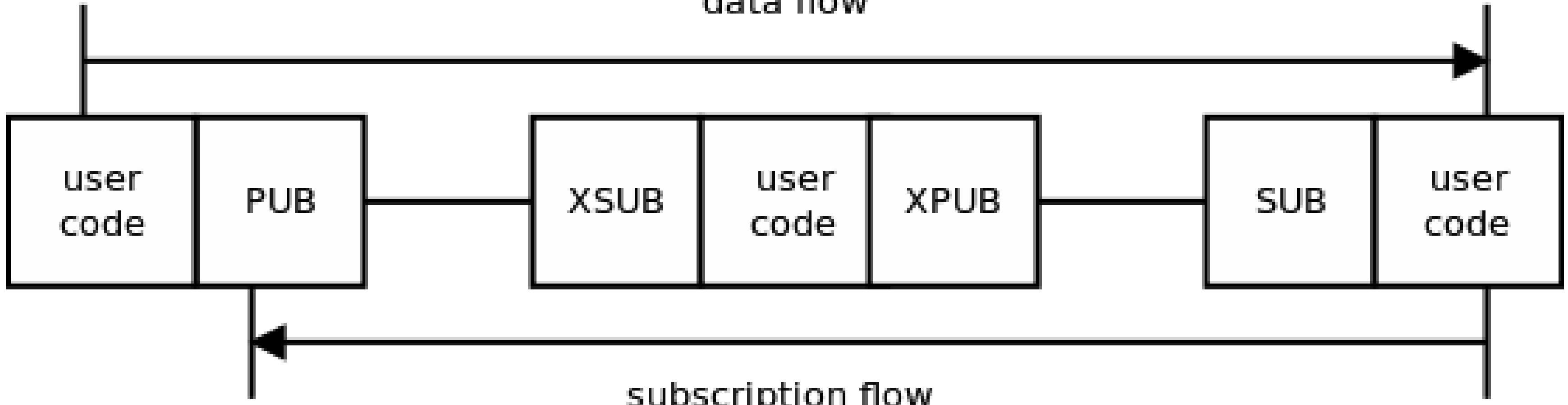
In most broker-based systems consumers subscribe for messages with the broker, however, there's no way for broker to subscribe for messages with the publisher. So, even if there is no consumer interested in the message it is still passed from the publisher to the broker, just to be dropped there

Messages should travel only through those lattices in the message distribution tree that lead to consumers interested in the message



Subscription Forwarding

- XPUB is similar to a PUB socket, except that you can receive messages from it. The messages you receive are the subscriptions traveling upstream.
- XSUB is similar to SUB except that you subscribe by sending a subscription message to it.
- Subscription messages are composed of a single byte, 0 for un-subscription and 1 for subscription


data flow
user code
PUB
XSUB
user code
XPUB
SUB
user code
subscription flow