WARWICK
THE UNIVERSITY OF WARWICK

DAQ and Trigger Systems At Present and Future Neutrino Experiments

Dr Benjamin Richards (benjamin.richards@warwick.ac.uk)

Introduction (Who am I)

based at Warwick University.
Been working in particle physics for >15 years. Mostly on neutrinos.
Research topics include: DAQ design and development, high performance software, triggering, calibration systems, supernova neutrinos, solar neutrinos and double beta decay

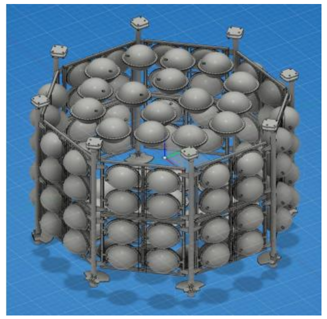
Currently work on 8 different experiments:

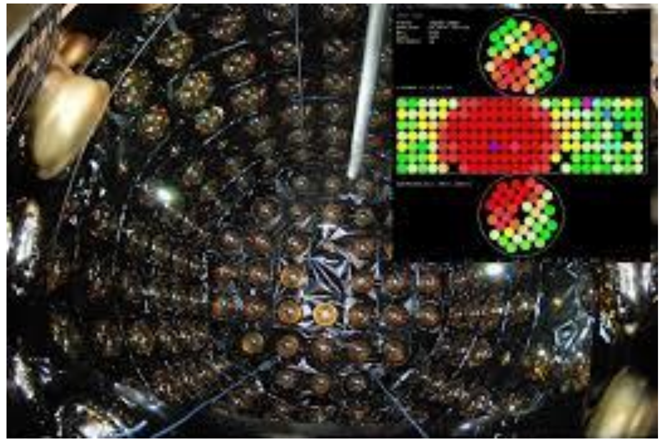
- 3 currently taking data: Super-K, EGADS and ANNIE
- 5 more in design and construction IWCD, WCTE, BUTTON, WATCHMAN and Hyper-K
- Designing the DAQ for a couple more proposed detectors
- Consult on another handful about DAQ.

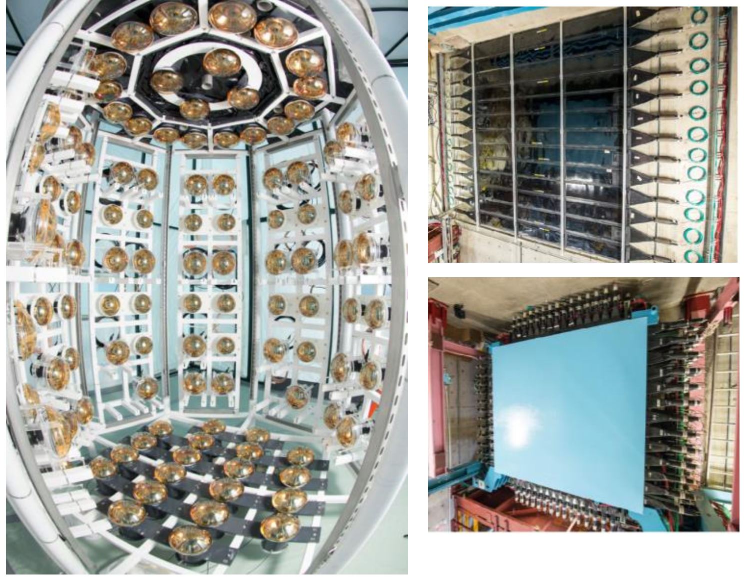
In charge of DAQ design and implementation for (Hyper-K, IWCD, WCTE, BUTTON, WATCHMAN and ANNIE) experiments.

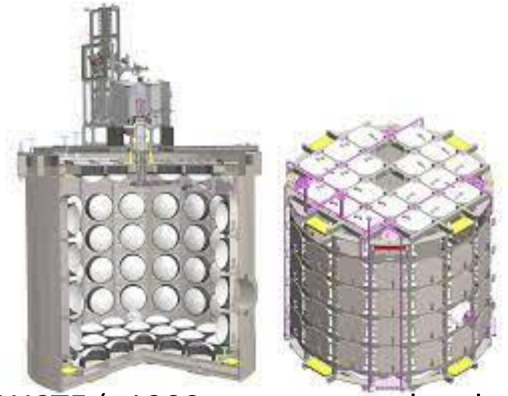
Author of advanced software framework ToolFramework (used for analysis, simulation, reconstruction etc)
And a derived DAQ framework ToolDAQ (used for building test-stands to full detector DAQ systems).

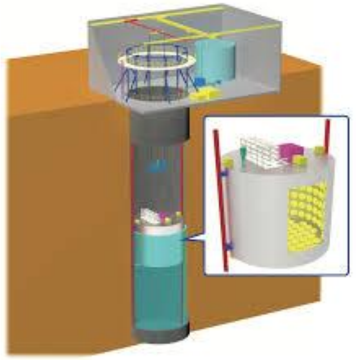
Detectors

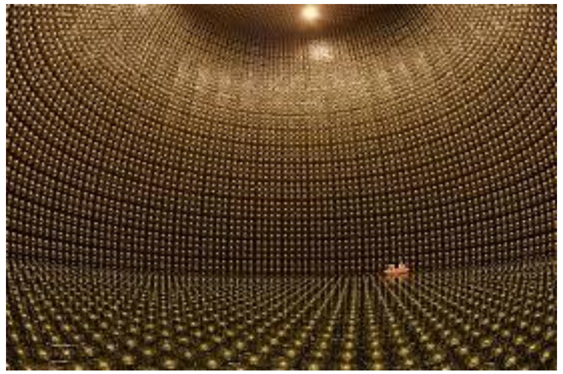

Button (~120 pmts, LAPPDS)

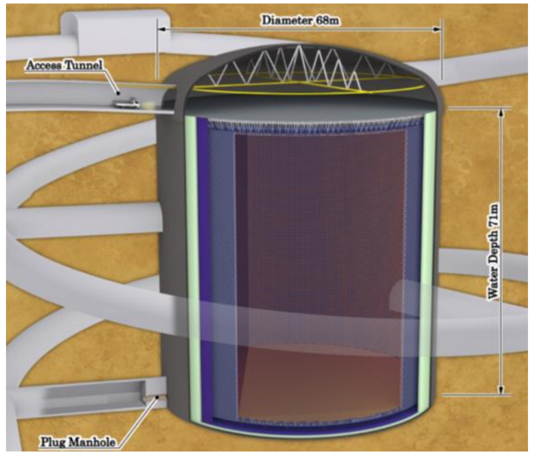

EGADS (240pmts)


ANNIE (~300pmts, front veto, muon range detector, LAPPS, 4 asynchronous data streams)


WCTE (~1900 pmts, aerogel and tof detectors, 2 asynchronous data streams)


IWCD (~6,600 pmts)


Super-k (~20,000 pmts)



Hyper-k (~35,000 pmts, 19,000 in MPMTs)

- Rather than describe the DAQ systems for all 8 experiments, their eccentricities and unique features that may not cross over into your experiments. I thought it would be more useful to explain the technology choices behind the some of them, the design principles and thinking and how we are applying to the next generation of detectors.

What We Want From A Modern DAQ System

- **Modular**:
  - allowing for easy upgrades, maintenance, additions and substitutions
- **Scalable**:
  - Can be easily expanded from testing a couple of sensors on a test bench to tens of thousands of sensors in a detector
- **Fault tolerant:**
  - If errors happen the system doesn’t go down
- **Error correcting:**
  - If errors happen the system can recover
- **Hot swappable:**
  - Components can be added and removed dynamically
- **Reliability / Least downtime**

In Hyper-k not miss a supernova that can happen at any time, cant loose any data and system must last for 20+ years.

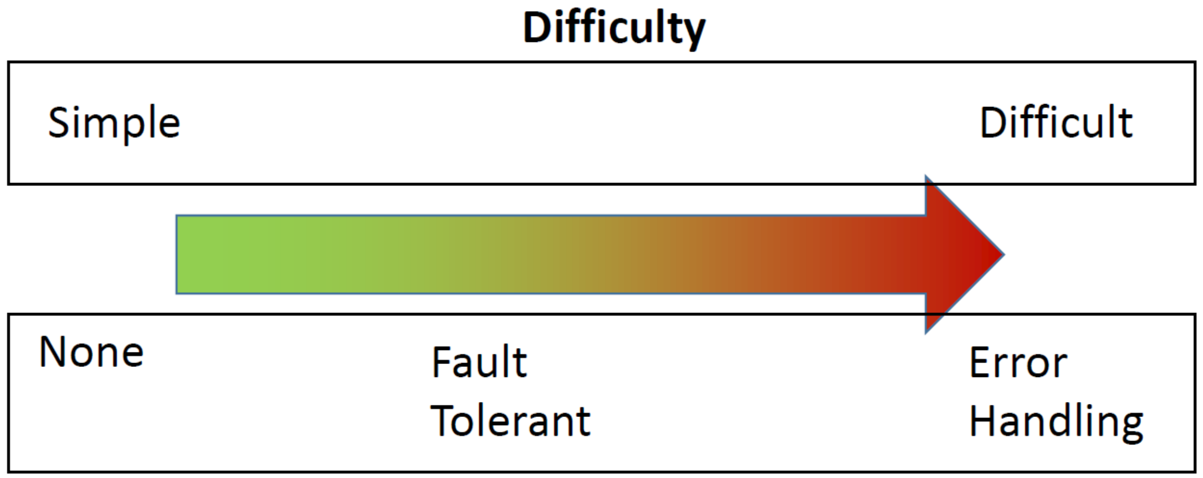
How To Design A Reliable DAQ system

- Remove single points of failure

- However Redundancy is easy, fault tolerance is harder, fault correcting even harder again

- Always have to ask what if

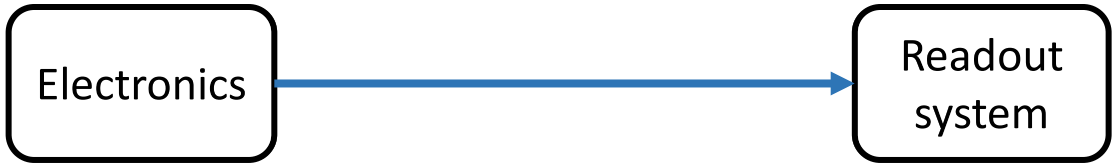
"**X**" goes wrong



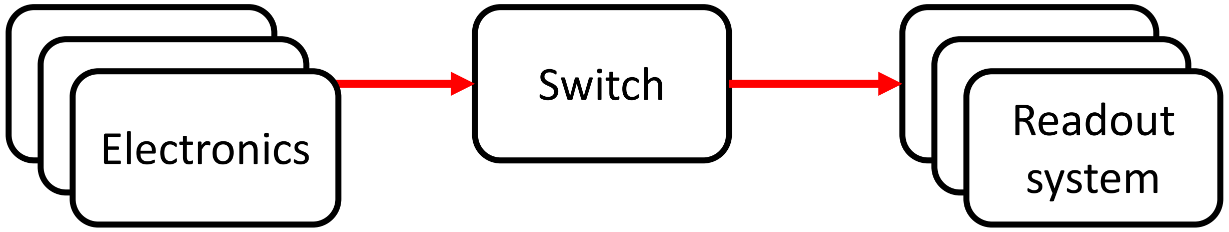
Communications / Networking (hardware)

When connecting your detector electronics to your read out system you have a couple of choices.

- **Custom link / protocol**
  - Not routable
  - Can be faster
  - More labour intensive
  - Custom hardware and software
  - Usually very low fault tolerance
  - Harder to scale



- **Standard protocols (e.g. Ethernet)**
  - Routable
  - Have some overheads
  - Standard off the shelf components
  - Off the shelf implementations
  - Tried and tested
  - Allow for high fault tolerance and scalability
  - Can double up as high precision timing e.g. (white rabbit)



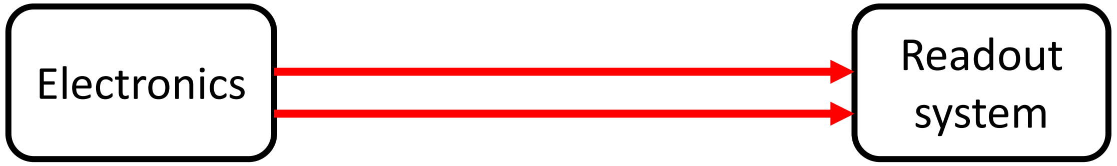
How To Make Networking Fault Tolerant

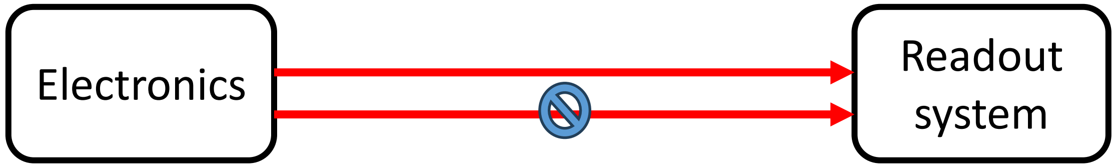
What if a link goes down?

Bonding:

Modern SOM can run lightweight Linux that have bonding. Bonding allows you combine multiple connections to achieve either increased throughput or fault tolerance

- **Round Robbin**:- send packets sequentially on each link (increased bandwidth)
- **Active backup**:- if one fails the other takes over (redundancy)
- **XOR**:- clients distributed on links
- **Broadcast**: replicated output
- **802.3ad**:- switch negotiated packet distribution
- **tlb**:- adaptive transmit load balancing on sending
- **alb**:- uses above plus arp requests to reroute inbound traffic.



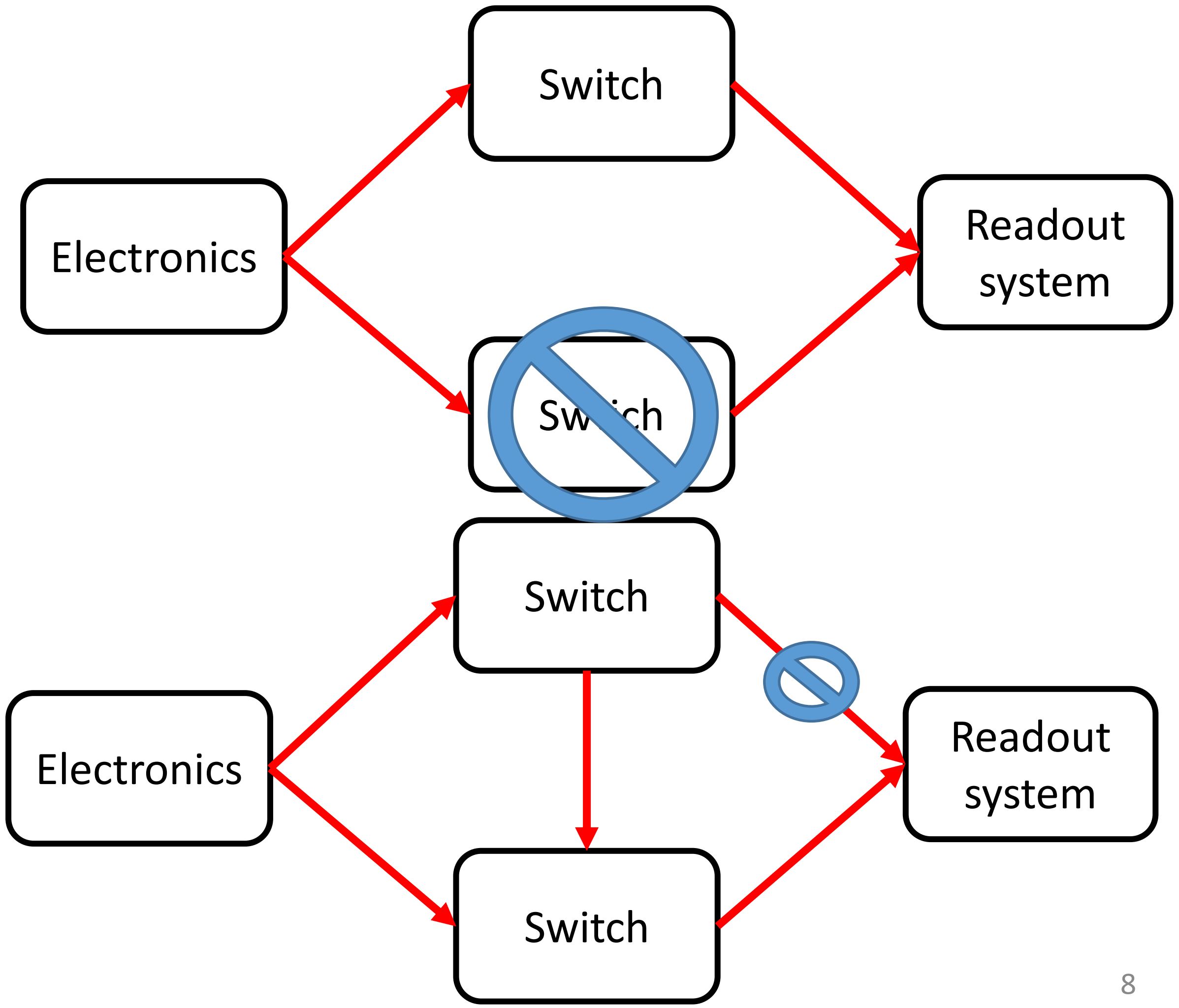


How To Make Networking Fault Tolerant 2

What if a switch goes down?

Multi chassis link aggregation:

- Each of the redundant links to your electronics is sent to a different switch
- Most companies have a form of switch interconnect to also handle rerouting traffic over transparent to clients.
- Can improve bandwidth further with multiple links to each switch

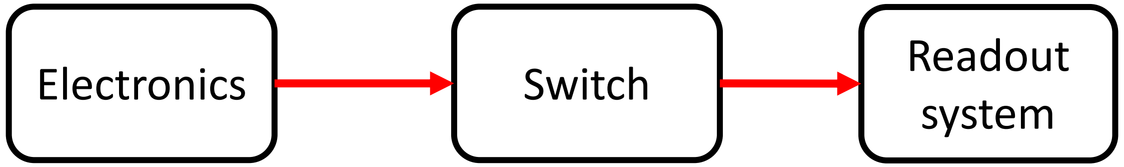


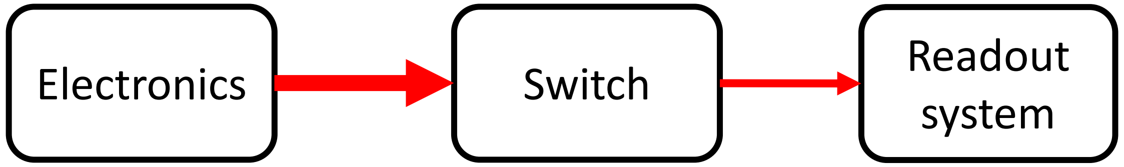
UDP/IP vs TCP/IP

UDP/IP (fire and forget)

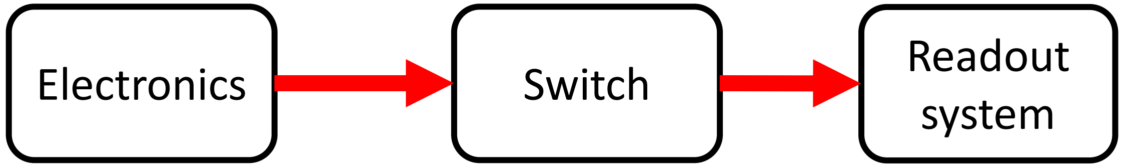
- Routable (good for scaling)
- Simple to implement in hardware (e.g. FPGAs)
- Fast No impact with data size
- Lacks any consistency checking, have to manually send and receive acknowledge statements and perform checksums / flow control

- **TCP/IP (guaranteed delivery)**
  - Routable (good for scaling)
  - More complex (although IP cores exist). Easier with Linux on SOM
  - Prefers larger packet sizes for good efficiency
  - Has built in built in error checking and re-transmission.





Saturating links can cause data loss



Eventually everything will arrive

Derivatives

RabbitMQ	KubeMQ
RocketMQ	IBMMQ
FairMQ	Google pub sub
ActiveMQ	AmazonMQ

Etc.

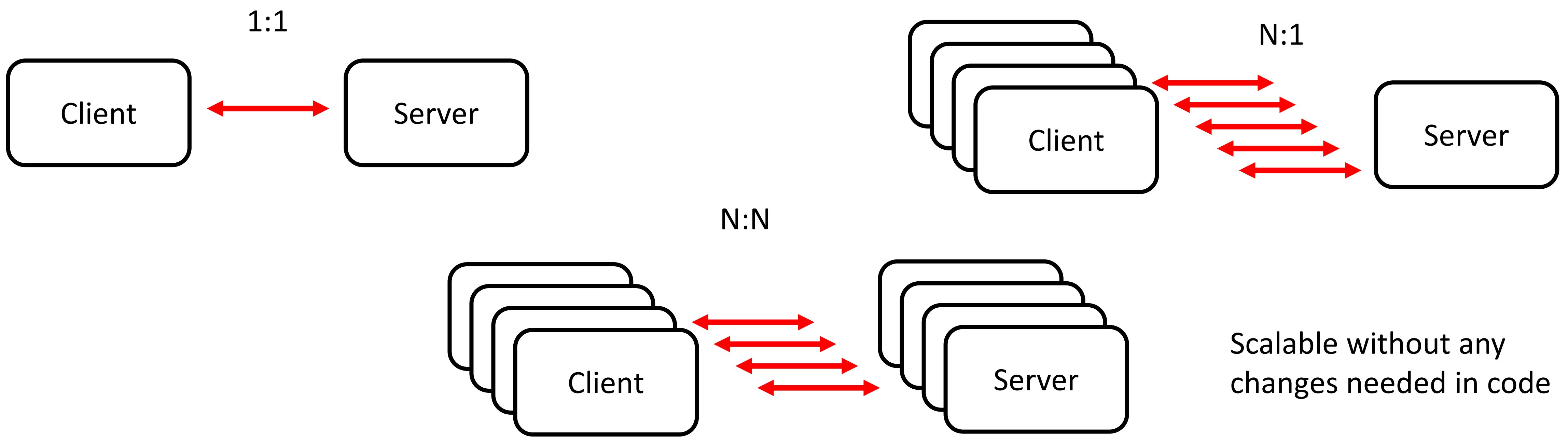
Advanced API ZeroMQ

What if your readout machine goes wrong? Distributed networking…

On top of TCP/IP protocol a number of message queuing API abstractions exist which can improve both redundancy and scalability.

These allow simplification of the software writing to allow N-N communications via a number of abstracted port paradigms.

- **Req-Rep**: request reply pattern (bi-directional messages are round robbined between peers)
- **Pub-Sub**: publisher subscriber pattern (uni-directional messages are sent to all subscribers [filters can be applied])
- **Dealer-Dealer**: Asynchronous distributed messaging pattern (bi-directional round robbibned)
- **Dealer-Router**: Asynchronous request reply pattern(bi directional round robbibned)
- **Push–Pull (pipeline):** 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.



Scalable without any changes needed in code

Allows simple expansion and add new components without any further code written

- Auto tries to reconnect
- Fair queues and distributes load/data
- Fault tolerance due as data rerouted to active peers
- Does the multiple connection threading for you
- Provides proxies and select functions for multiple socket multiplexing behaviour

Dynamic Scaling/Routing & Hot Swap Ability

- ZMQ by itself does not necessarily give you full dynamic fault recovery. This is because connections are still defined and made manually (hardcoded)
- What you really need is a way to dynamically adapt to changing topology and create new connections automatically, removing the need for hard defined detector readout structure.
- This effectively makes your DAQ components hot swappable.
  - Component fails its automatically removed form the system, replace it and the replacement is automatically added
  - Need more backend processing power or data rates getting to large, add a new readout machine or trigger processor and the system integrates it automatically

- To do this every DAQ component would need to know about the existence and state of every other DAQ component dynamically. This is where the principles of **dynamic service discovery** can help.

- This also allows decentralisation of your system, you don't need a single controlling node to determine how or where peers connect to and control the state of a system.

- Also for allows design in communications where upstream components to not need to know about downstream components simplifying their communications and making them more robust

ToolFramework & ToolDAQ

In 2015 faced with designing DAQ systems for multiple experiments at multiple scales, So I built a DAQ framework ToolDAQ to have all these features (presented at CHEP 2018).

Additionally also a general purpose framework on ToolFramework

Both are still being actively developed in the UK and used by a number of experiments.

It was deigned to incorporate a number of common DAQ features whilst:

1. Being very easy and fast to develop DAQ implementations in a very flexible and modular way.
2. Including dynamic service discovery and scalable network infrastructure to allow its use on large scale experiments.

Features

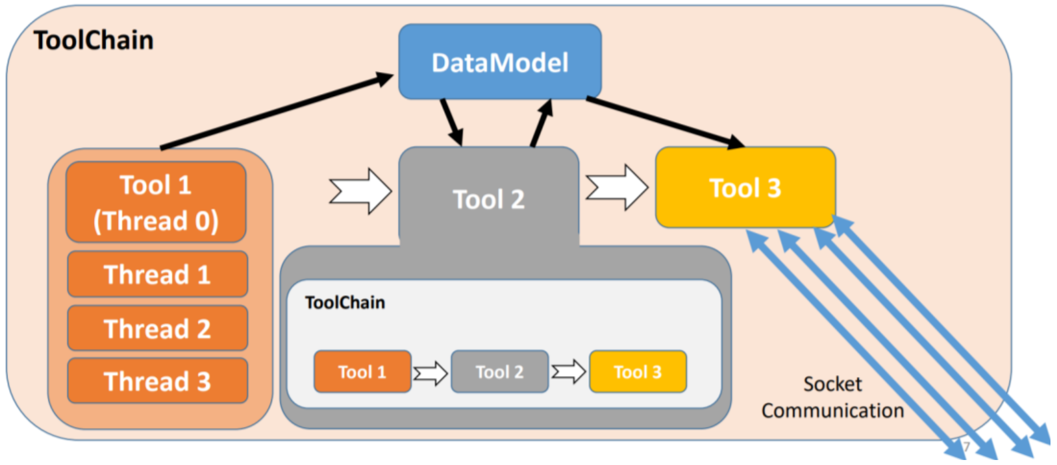
- Pure C++
- Fast Development
- Very Lightweight
- Modular
- Highly Customisable / Hot swappable modules
- Scalable (built in service discovery and control)
- Built in and external Integration libraries and API for device monitoring and slow control
- Fault tolerant (dynamic connectivity, discovery, message caching)
- Underlying transport mechanisms ZMQ
- (Multilanguage Bindings) (python)
- JSON formatted message passing
- Few external dependencies (Boost, ZMQ)
- Integrated Node management
- Web and command line frontends

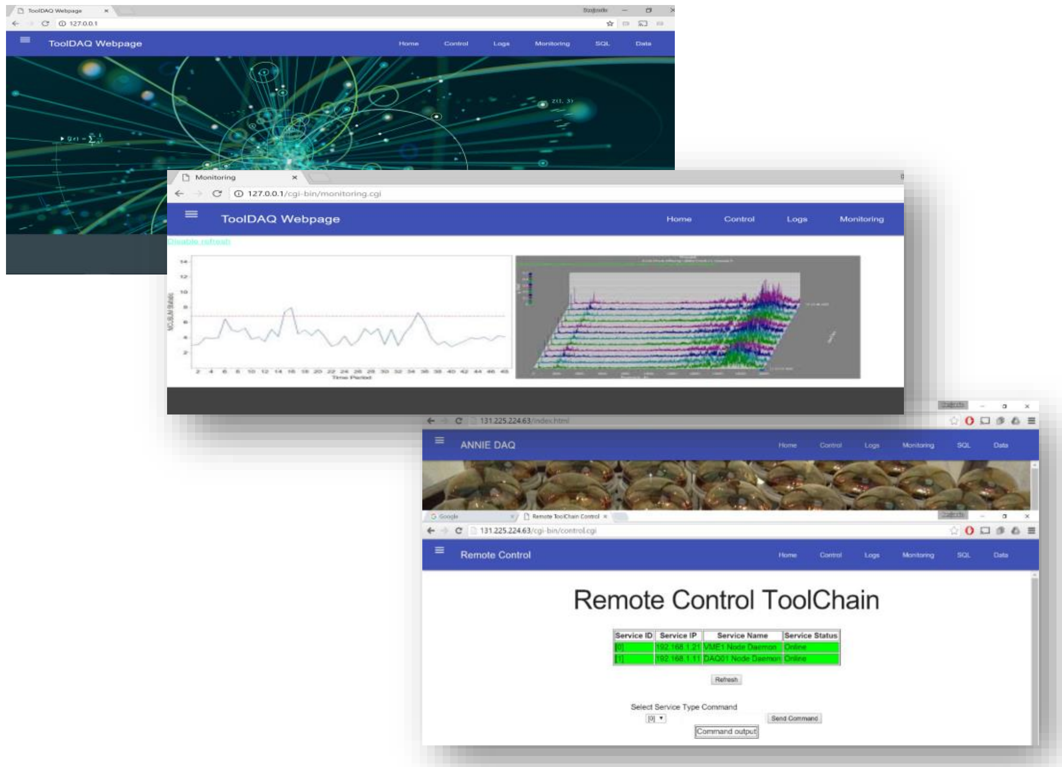
ToolDAQ

- Contains Toolchain which hold dynamic Tool objects (C++ or python) created from factories.
- These communicate through a transient data storage class
- It allows for flexible complex structures and paradigms, multithreading, sub toolchains, workers farms etc. etc.
- Full web front end and command line decentralised control systems
- Dynamic object based Storage and IO class libraries

Currently being used by about **10 particle physics experiments** in multiple roles including test stands, online calibration devices, all the way to full detector DAQ systems.

A nice feature is that multiple experiments are also using ToolFramework as their analysis, reconstruction and simulation, packages. This allows for offline tools to ported onto online systems as Tools are inter compatible





Remote Control ToolChain

Service ID	Service IP	Service Name	Service Status
[0]	192.168.1.21	VME1 Node Daemon	Online
[1]	192.168.1.11	DAQ01 Node Daemon	Online

Triggering: Data processing and accelerators

- One place this comes in handy is triggering, reconstruction and event building, as we can use the same tools online and offline. The framework also allows you to produce dynamically scalable job farms for the processing of event data (see next slide)
- We also make heavy use of accelerator cards (FPGAs or GPUs) in our detector triggering systems.

FPGAs

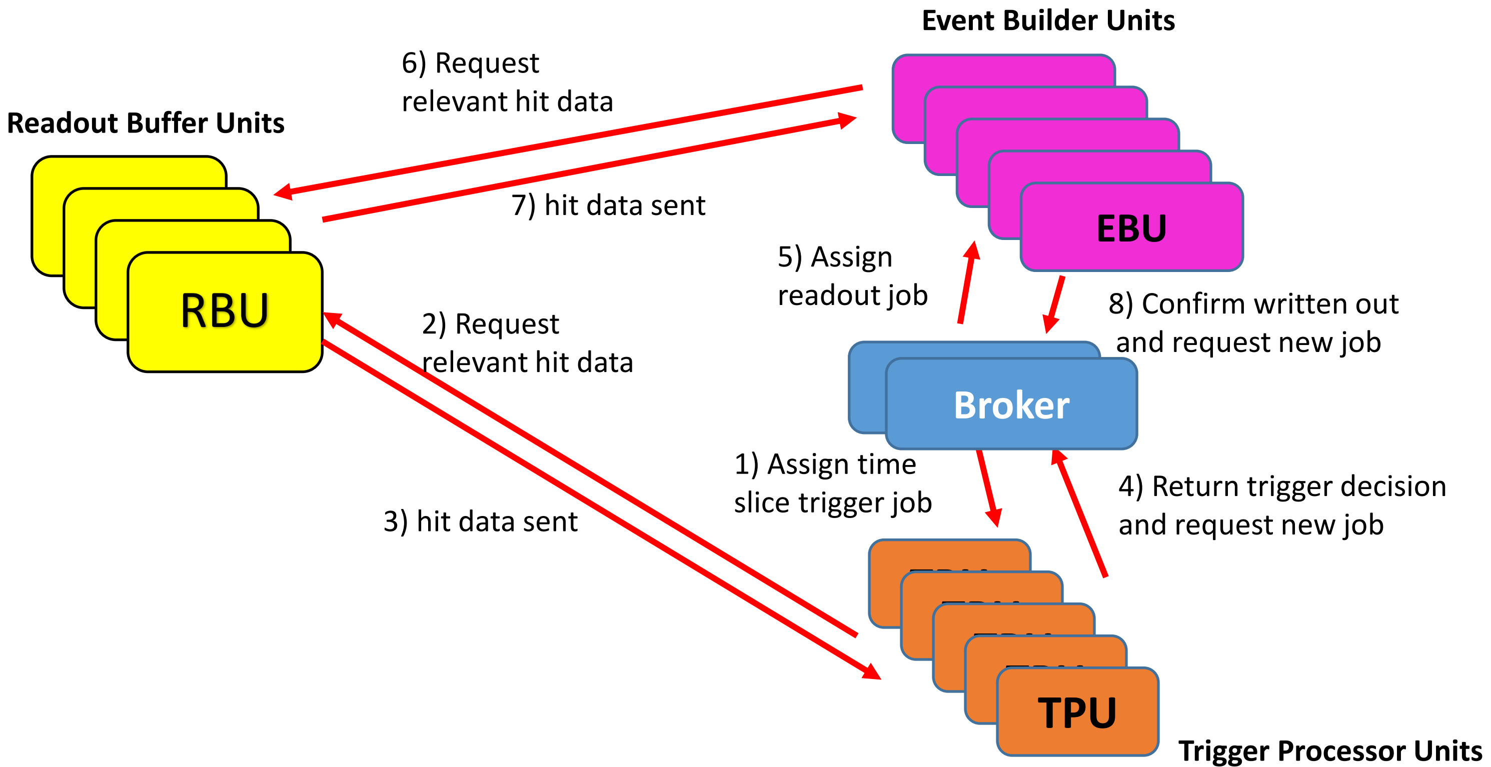
- Can be faster
- More expensive / less availability
- Less energy efficient
- Not general purpose (without firmware changes)
- Harder to program for (although some implementations of using C++ and C++ to Verilog exist)

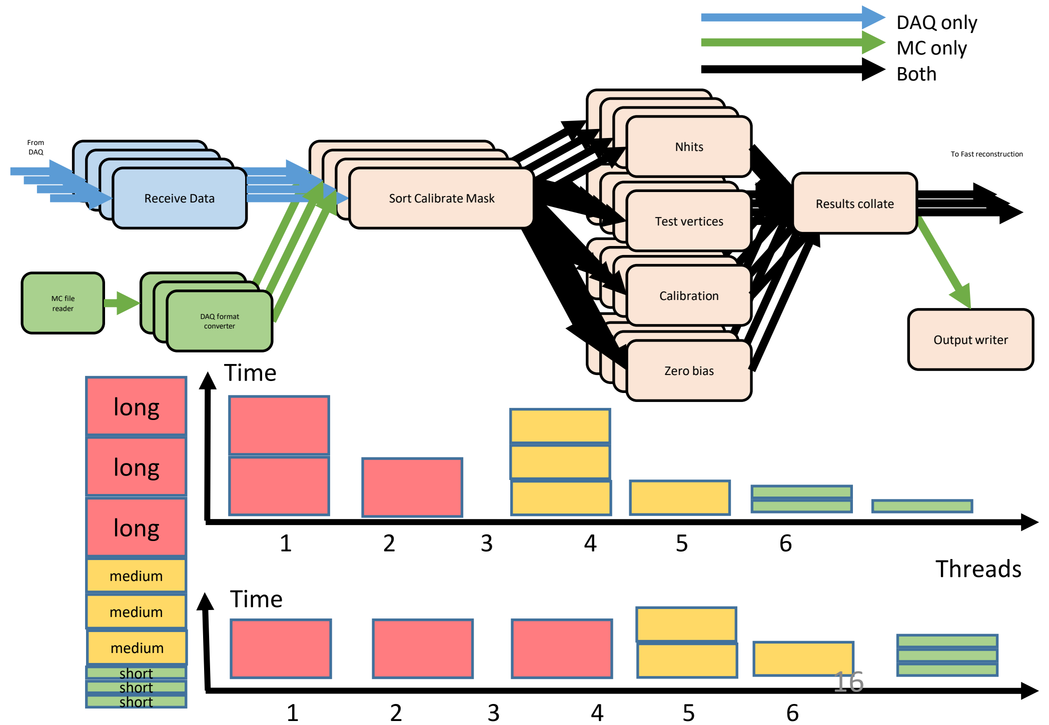
GPUs

- Cheaper
- Better availability
- More energy efficient
- General purpose
- Easy to program for libraries for multiple languages

Triggering In Hyper-K

- In hyper-k we make use of an online trigger farm using GPU accelerators and an event building farm all using a broker assisted paradigm.
- This allows us to ensure no data loss as triggering and event building jobs are tracked by the brokers and resubmitted in the event of hardware failure and that workload is balanced across nodes
- The triggering software itself running on the nodes intelligently manages CPU and GPU resources dynamically scaling threads and distributing work loads and has flexible expandable algorithms


Event Builder Units
Readout Buffer Units
6) Request relevant hit data
7) hit data sent
EBU
RBU
5) Assign readout job
8) Confirm written out and request new job
2) Request relevant hit data
Broker
1) Assign time slice trigger job
4) Return trigger decision and request new job
3) hit data sent
TPU
Trigger Processor Units



Front End Interfaces Slow control and monitoring

More and more of our work is done remotely. X window forwarding or VNC clients are slow, so custom local applications are less favourable.

Also need solutions that can last for 20+ years and still be supported allow for multiple clients, so we opt for web based interfaces. These make use of backend server side socket coms (via ZMQ and web sockets) and databases to provide, monitoring information, detector configuration and slow controls.

As much of the workload as possible can also be pushed client side with JavaScript and web assembly e.g. for rendering for plots monitors and event displays.

How do you make build in fault tolerance and scalability?

Backbends Reverse proxy / custom solutions

- Can have more servers for redundancy. However then you need to handle synchronisation and work load balancing (hot swapping?)
- For web servers this is easier, off the shelf solutions exist for reverse proxies like Nginx that allows a single server to offload its workload to multiple backend components

- Not necessarily going to work for SQL databases (notoriously difficult) its ok if you want cold or hot spares with replication, but if you want two active running databases, then you have to manage load balancing data requests and synchronisation. etc etc.
- In Hyper-K we address this with a custom middle man software built using ToolDAQ that allows for a scalable number of active running databases that form a Quorum. Read requests are load balanced between servers with write requests going to a master and changes disseminated. Automatic promotion occurs in the event of failure using aided by ToolDAQ's the built in service discovery

Virtualisation High Availability

Virtualisation can help. For a long time now both hardware virtualisation and kernel level virtualisation(e.g containers) have been around. This allows us to take a single server and divide up it CPU ram HDD resources as we see fit, allowing us to treat is as multiple machines.

When paired with multiple servers and a suitable hypervisor (VMWARE, Proxmox) and a shared storage (more on that in a second) we can also acheive high availability.

We have a quorum of servers that monitor each others state and the state of containers and virtual machines running on them.

Allows live migration (sub 10ms down time) and also allows high availability, where if the quorum sees a virtual machine is down it can automatically boot a replacement on a running server. Shared storage and/or snapshots allow for the server to start up and not loose data.

Data Storage

- How do we improve fault tolerance of data storage?
- Raid is ok:
  - Raid 0: striped (no drive failures)
  - Raid 1: mirrored (1 drive failure)
  - Raid 5 & 6: parity info (1-2 drive failures)
  - Raid 10…. 50…60: extra redundancy
- Hardware raid used to be a thing, built in battery cache to aid power cycle losses. However hardware raid has issues and is effectively dead in modern day systems design.
- Software implementations mdadm, ZFS, butterFS, lvm are the better option these days, as can provide better consistency checks and scalable and mismatched raided storage. However they still have the issue they are not distributed storage solution (i.e. loose your raid server loose your access to data)

Distributed File Storage Solutions

- Have multiple backup copies so multiple servers with multiple raid arrays so the data is protected.
- If you want these available at the same time and with load balancing etc, your looking at writing custom software.

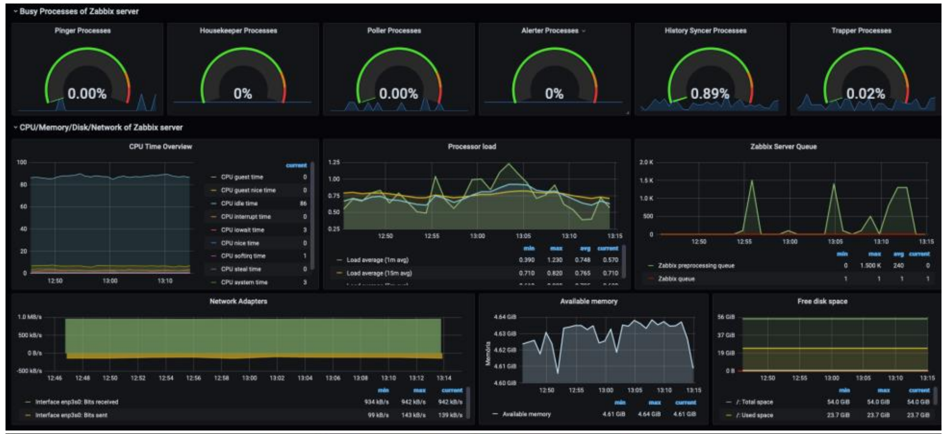
- Better solution, Ceph / WEKA. Distributed file storage. Have a quorum of servers that all over the network act as a single storage volume using deterministic file locations (so no query overhead) .
- Self healing, self managing storage solution with multiple data copies
- Can handle tiered storage and different storage media (SSD HDD etc)
- Does require fast networking for decent IO speeds (10Gb+)

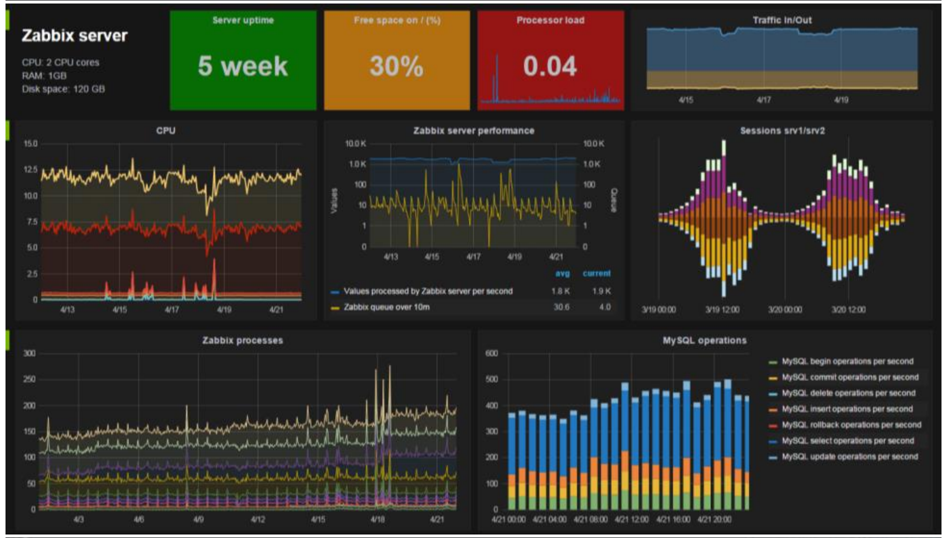
Server Resource monitoring

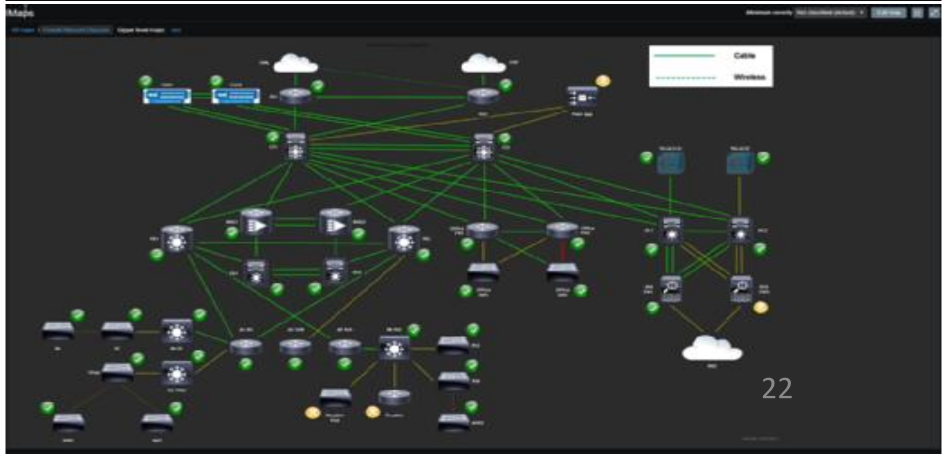
- It's important to monitor DAQ system resources and networking throughput.

- Many options exist for SNMP, however we use Zabbix

ZABBIX

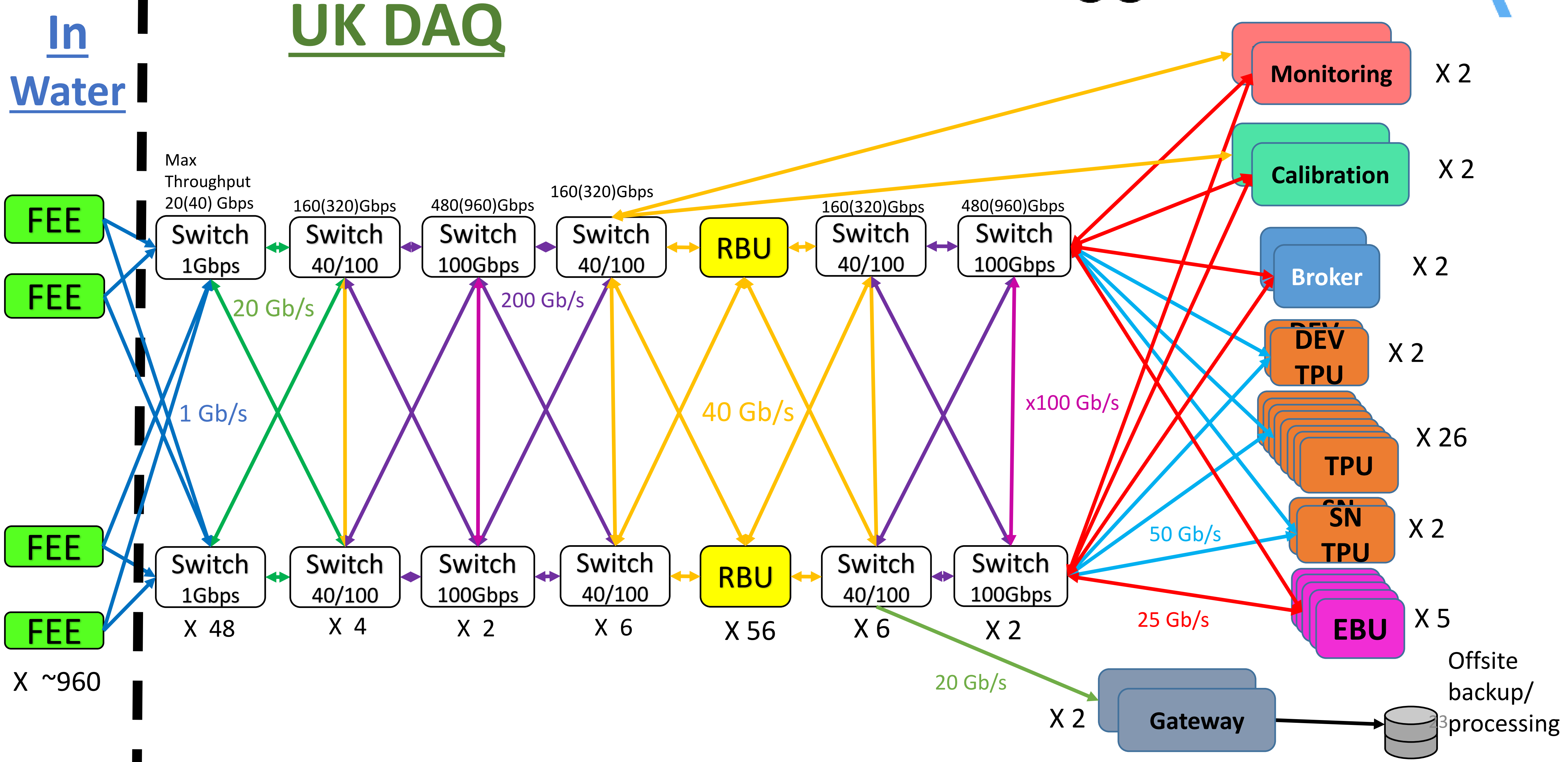






Hyper-K Reference Design

ToolDAQ



Thank You

Any Questions?