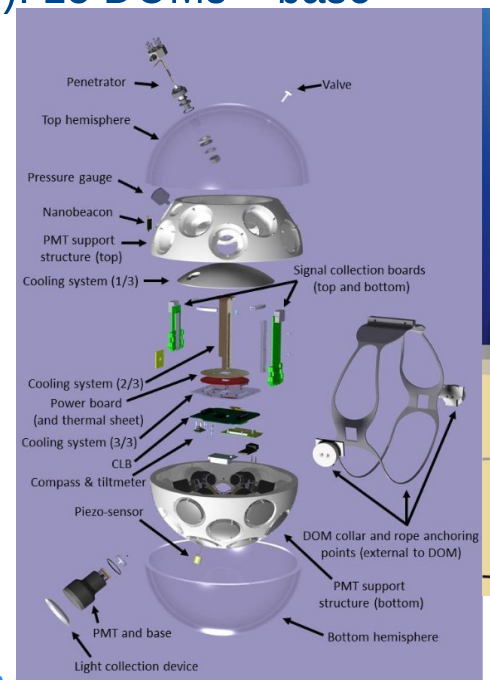
Scalable solutions for the Control Unit of the KM3NeT DAQ system

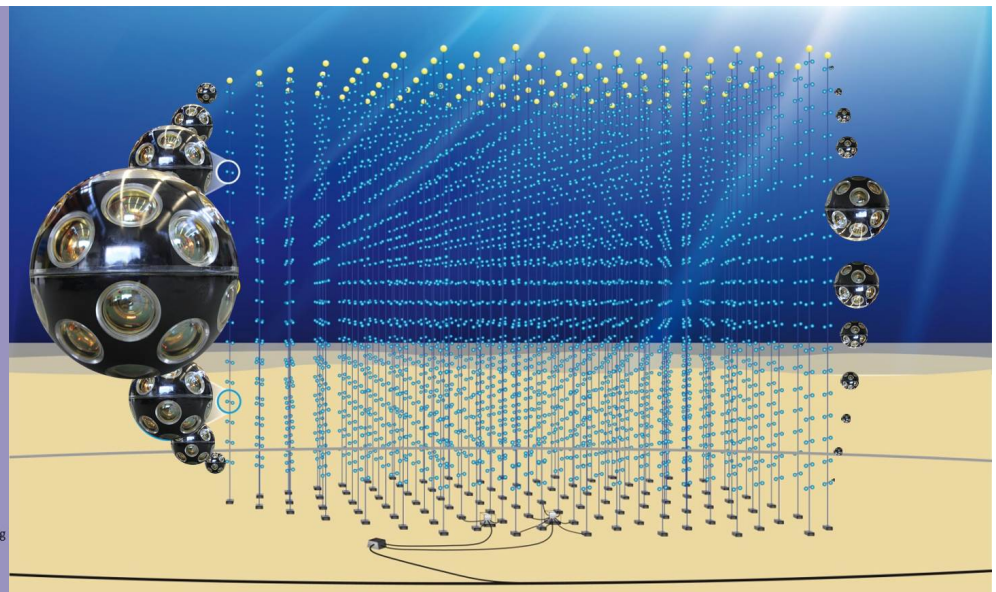
Cristiano Bozza
University of Salerno and INFN
for the KM3NeT Collaboration

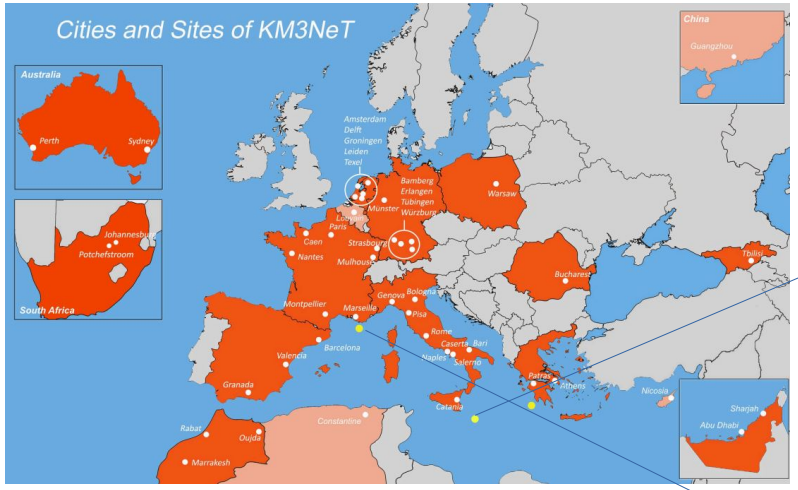
RICAP – Roma, Italy – September 2022

KM3NeT overview

- KM3NeT: modular neutrino telescopes in the Mediterranean Sea
- The basic unit: the Digital Optical Module (DOM) with 31 photomultipliers (PMTs)
  - Cherenkov radiation from charged particles produced in neutrino interactions
- Detection Unit (DU): 18 DOMs + base
- The logical core of each DOM/base: the **CLB** (Central Logic Board)





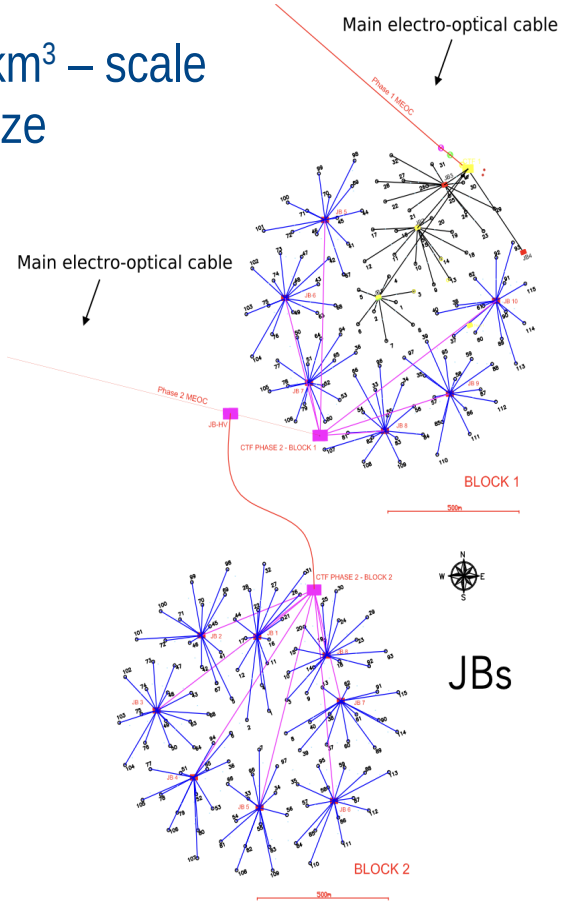


The Collaboration: worldwide partnership with strong EU core

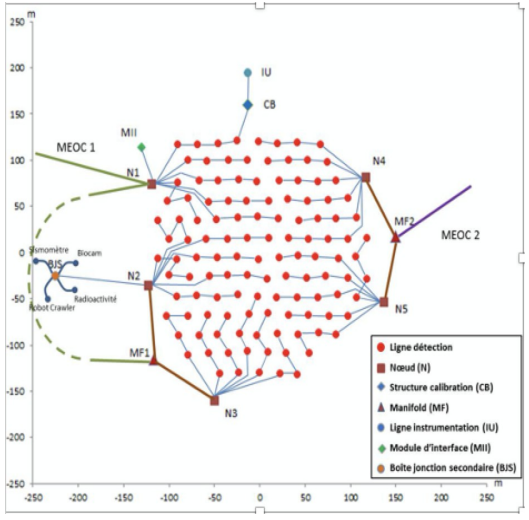
Two neutrino telescopes

ARCA footprint, off the Italian shore

Multi-km^3 – scale final size



ORCA footprint, off the French shore



KM3NeT construction status

- ARCA status:
  - 19 DUs operational
- ORCA status:
  - 11 DUs operational
- More DUs coming soon
  (deployment ongoing in these days)
  - Each DU is tested as an independent detector

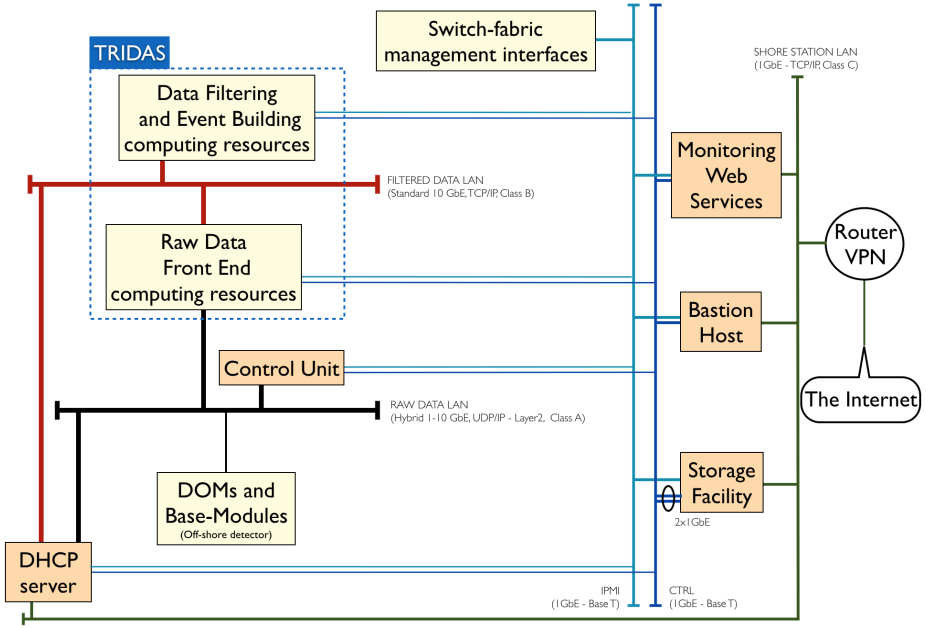
KM3NeT control SW figures

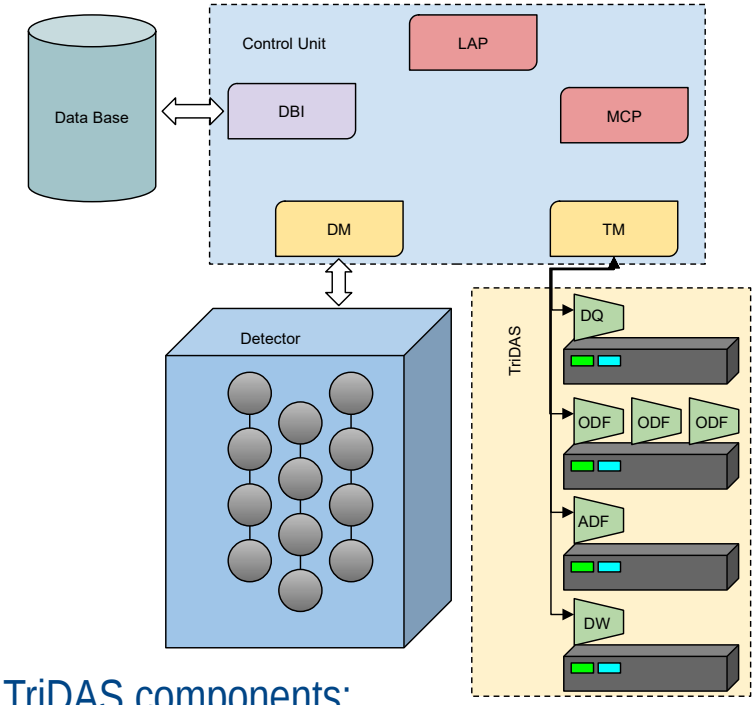
- ARCA: 2×115 DUs
  - 230 Base Modules, 4140 DOMs
  - 128340 PMTs
  - **~400,000** parameters to be monitored continuously (every 10~60s)
- ORCA: 115 DUs
  - 115 Base Modules, 2070 DOMs
  - 64170 PMTs
  - **~200,000** parameters to be monitored continuously (every 10~60s)

- **ARCA with 19 DUs, control data exchange with CLBs:**
  - ~76 incoming UDP datagrams/s
  - ~175 bytes/s (from CLBs), ~19 bytes/s (to CLBs)
  - CPU load: 1.39 core (Intel Xeon E5-2640 v2 @ 2.00GHz)
  - 18 MB monitoring data every 10 minutes (**~ 1 TB/year**)

DAQ structure

- Functional components:
  - Control Unit (CU)
  - Trigger and Data Acquisition System (TriDAS – controlled by the CU)
  - General monitoring (not controlled by the CU)





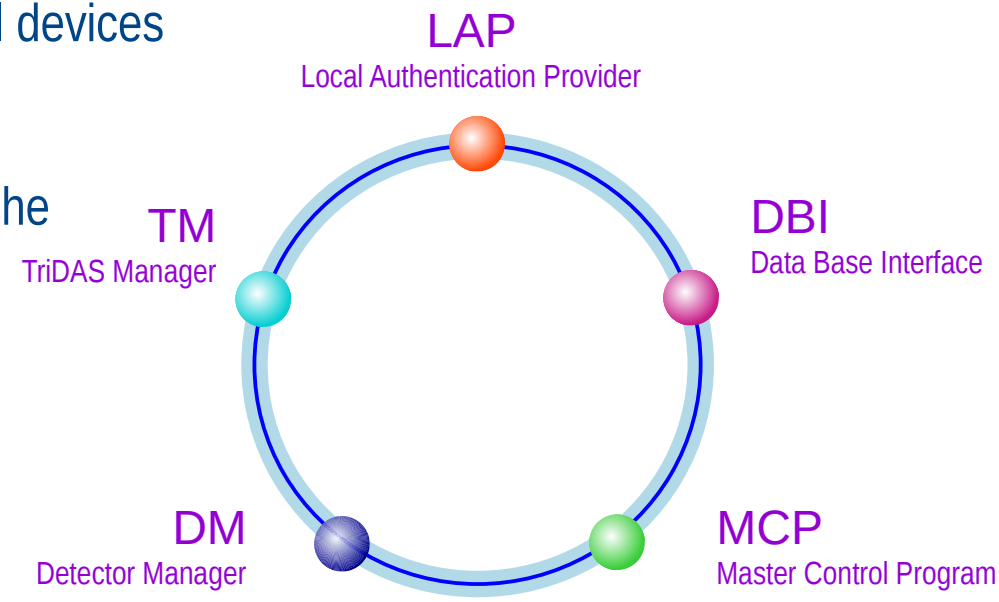
TriDAS components:

- Data Queues (DQs) build timeslices for
- Optical Data Filters (ODFs) which trigger events to be written using
- Data Writers (DWs)
- Acoustic Data Filters (ADFs) receive data from DQs

DAQ Control Unit

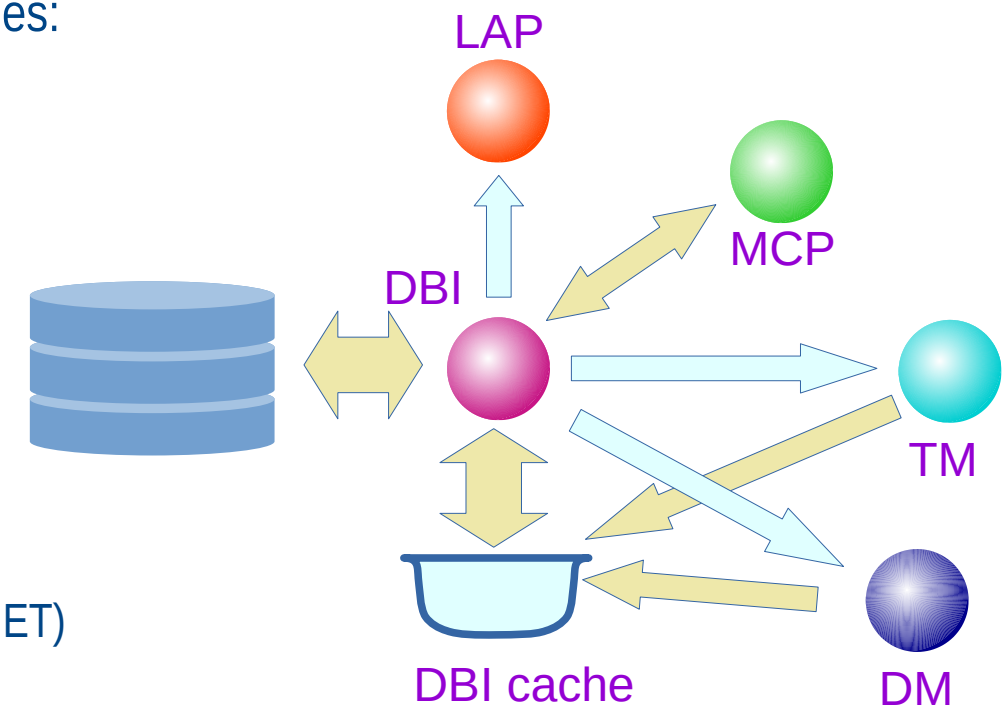
- LAP: provide authentication to users and services
- DBI: data base interface, connected to remote DB
- MCP: manages data acquisition tasks and schedules
- DM: controls DOMs, bases and additional devices
- TM: controls data processing software

- Each Control Unit system has its own cache
  - Work even if others are missing
  - Remote DB may be unreachable
    - Keep up operation using DB cache + local upload buffers
- Cross-process communication via HTTP-based messaging protocol (SAWI)
- Browser as debugger
- GUI interface built on top of messaging API



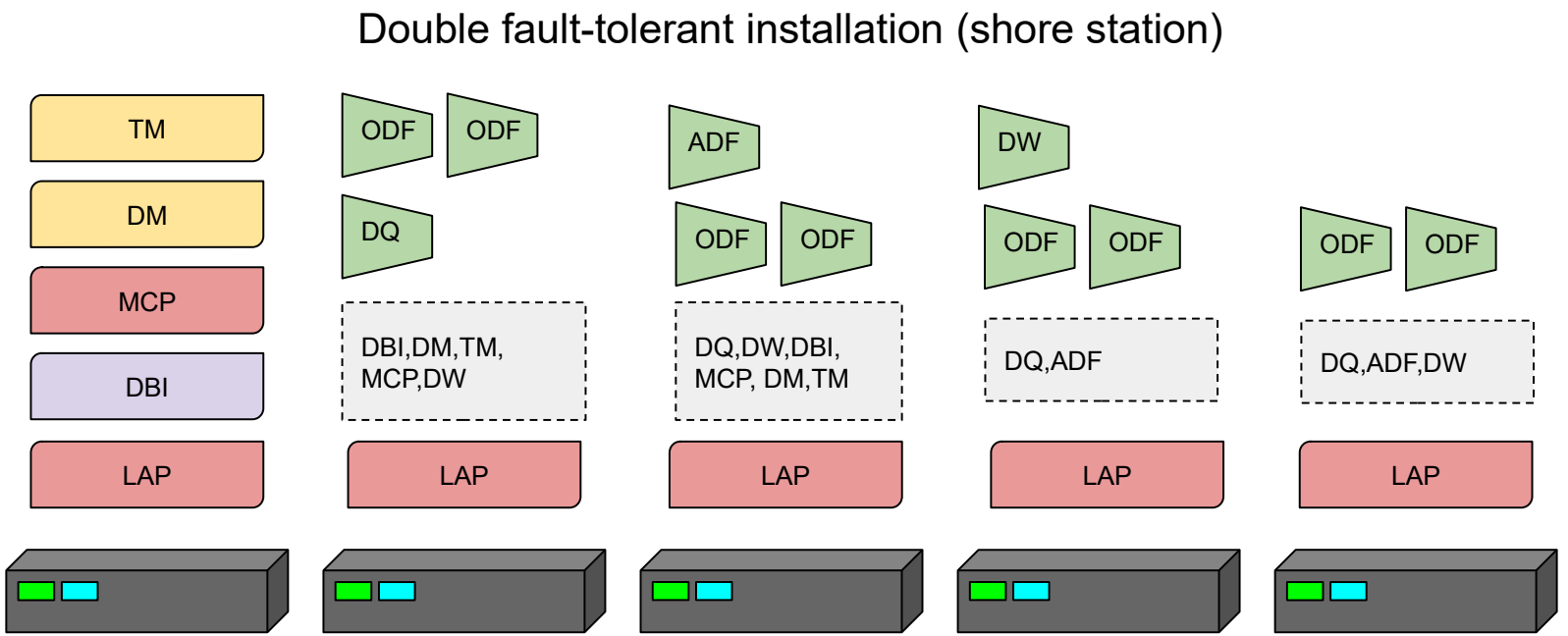
Control Unit data flow

- Most relevant data flows are shown
- Local download/upload cache based on files:
  - Simplify development
  - Keep the architecture flexible
  - Avoid local DBs
    - Licensing
    - Maintenance
- Monitoring/slow control data (DM, TM, MCP, LAP), Run bookkeeping (MCP)
  - Written to DBI cache via NFS
  - Regularly uploaded to remote DB (SQL.NET)
- Text logs for diagnostics (all):
  - Written to file (NFS)
  - Uploaded to remote file storage (file copy)



CU Dynamic Provisioning

- CU and TriDAS software can run on various (performant) commercial HW
- Failure of one or more servers can be overcome by automatic reallocation of tasks
  - Admin intervention not needed!

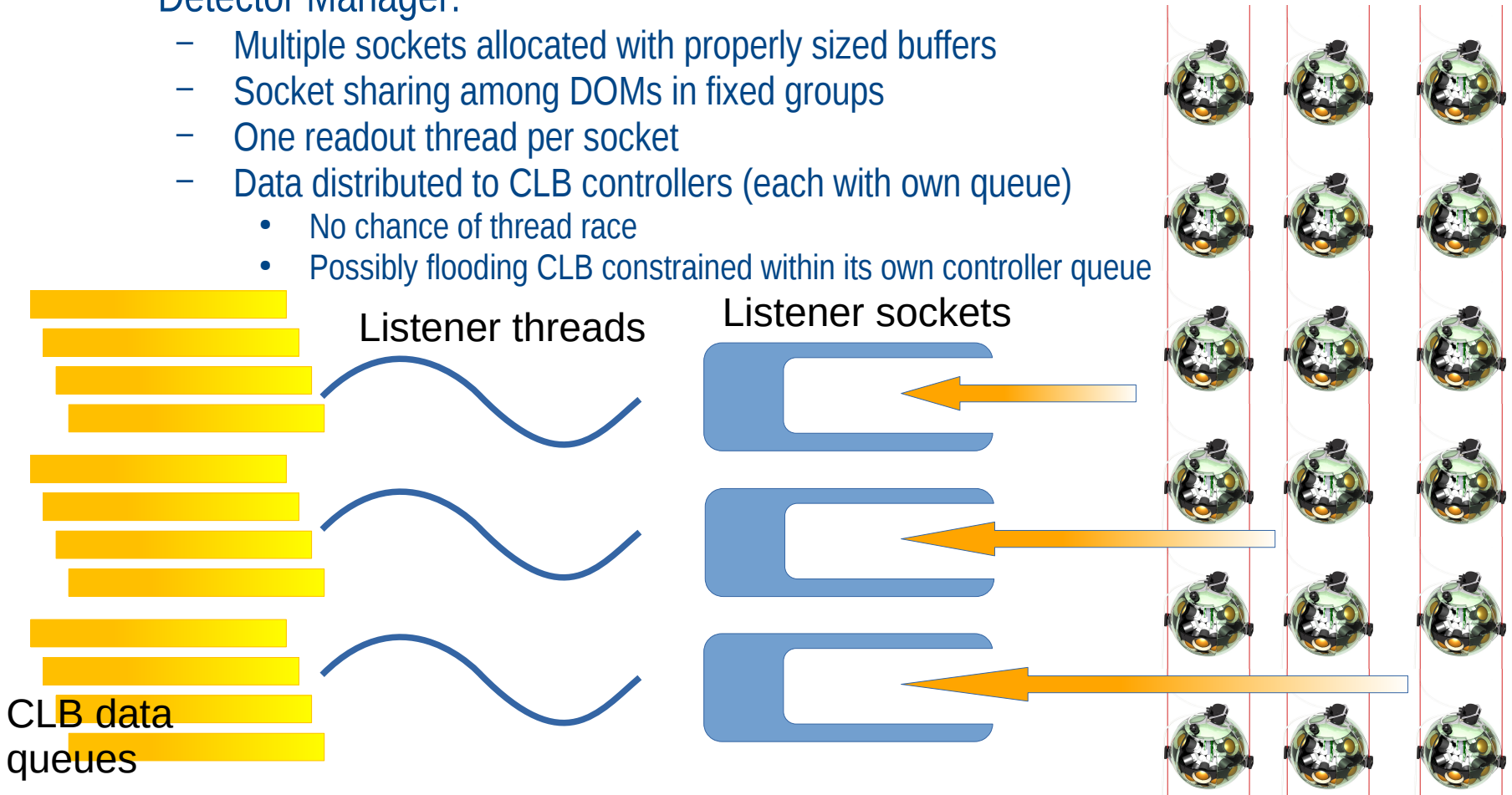


CU performance-impacting tasks

- Network communications: ~4000 DOMs, each with one CLB, communicating asynchronously with the DM via SRP (UDP-based messaging protocol)
- CPU: sorting information (by time or ID) requires *N log N* effort (or N^2 in case of bad design)
- Disk: detailed logging needs continuous writes of small buffers
- Presenting monitoring information in GUI
  - Graphical representation is left to client
  - Indexing and retrieval of different subsets of information with multiple clients
- Warnings:
  - Thread starvation (threadpools exhausted or excessively growing)
  - Deadlocks
    - Internal: multiple threads with data consistency requirements
    - Cross-process: logical error loops

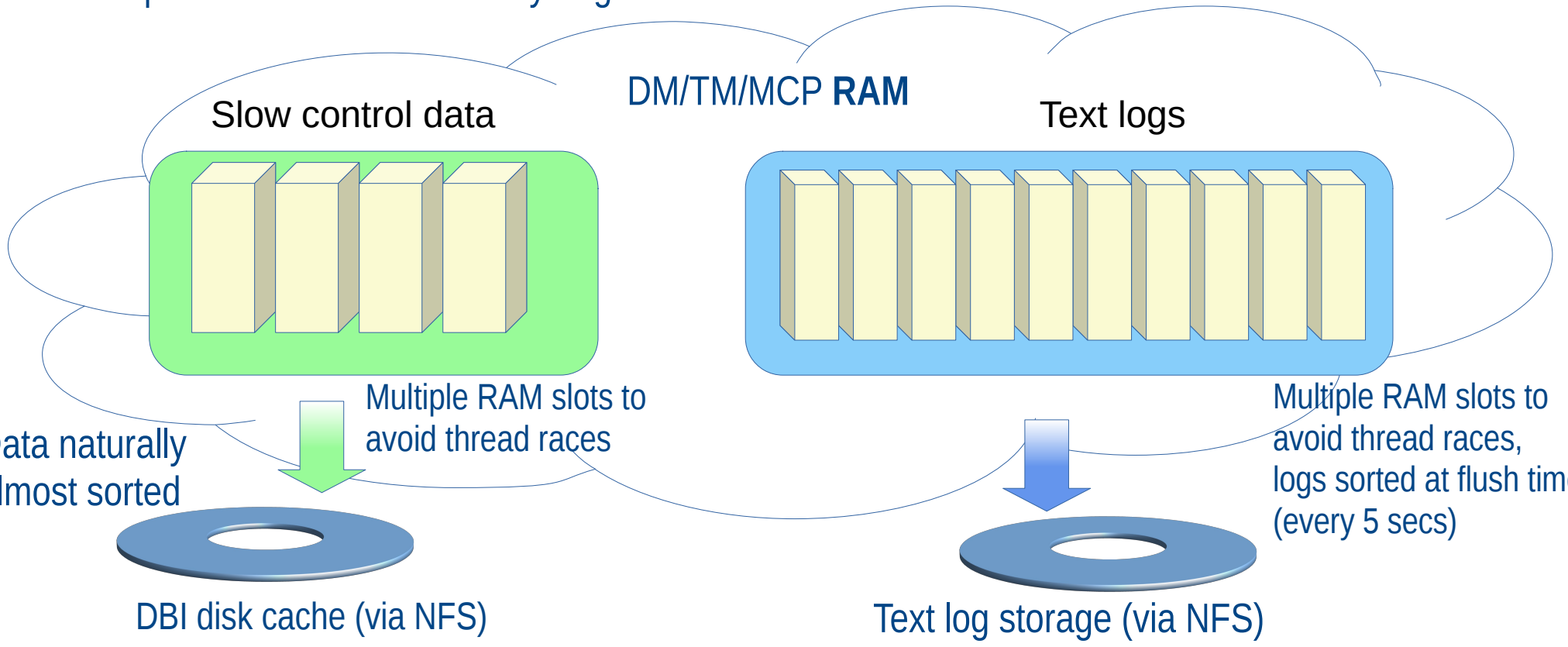
CU network access

- Detector Manager:
  - Multiple sockets allocated with properly sized buffers
  - Socket sharing among DOMs in fixed groups
  - One readout thread per socket
  - Data distributed to CLB controllers (each with own queue)
    - No chance of thread race
    - Possibly flooding CLB constrained within its own controller queue



CU disk access

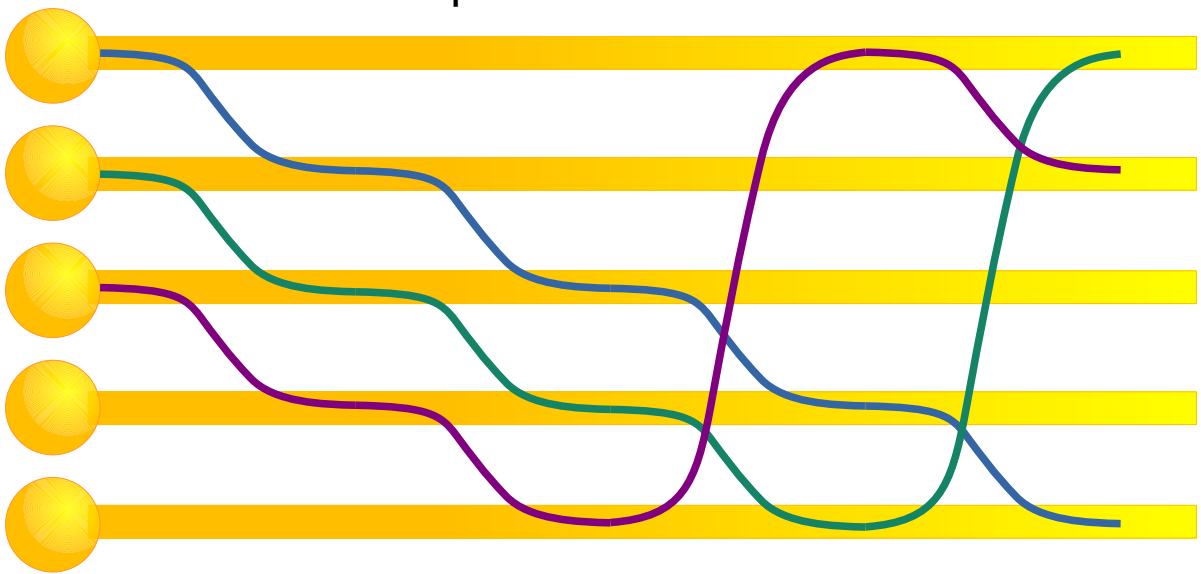
- Disk access optimized by caching slow control data and logs in memory
- Optimized writes in relatively large chunks



CU CPU load analysis

- CLB controllers are powered by controller threads in “round robin” fashion
  - Balance care-needing CLBs with overall performance
  - Avoid unforeseen CPU loads: a coarser slow control is preferred over an unresponsive server
    - The number of threads can anyway be tuned according to needs.
  - Excess CLB messages are filtered out: for each parameter, only the last value is considered

CLB controllers and queues



Event processing threads

All CLB control operations scale linearly with the data rate and number of CLBs

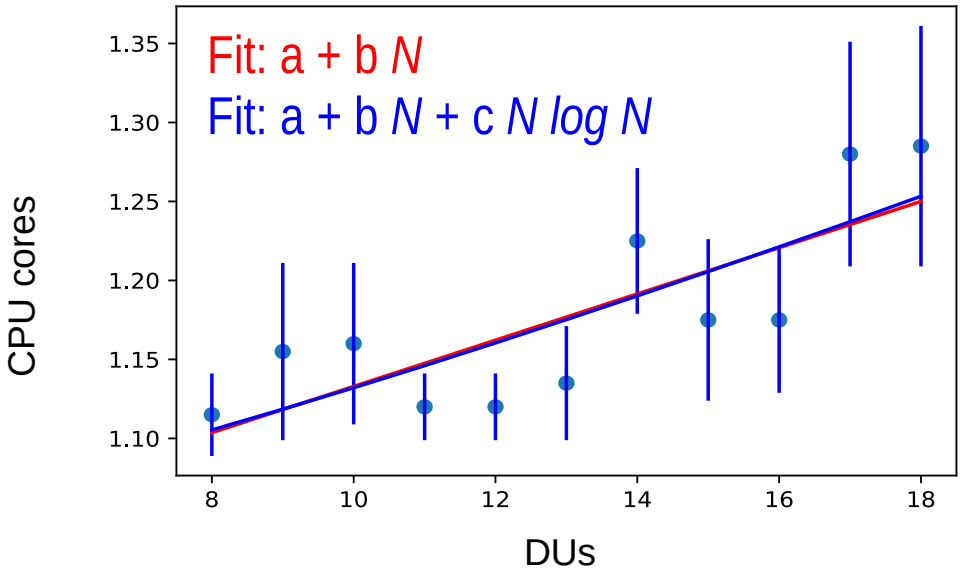
CU GUI Monitoring

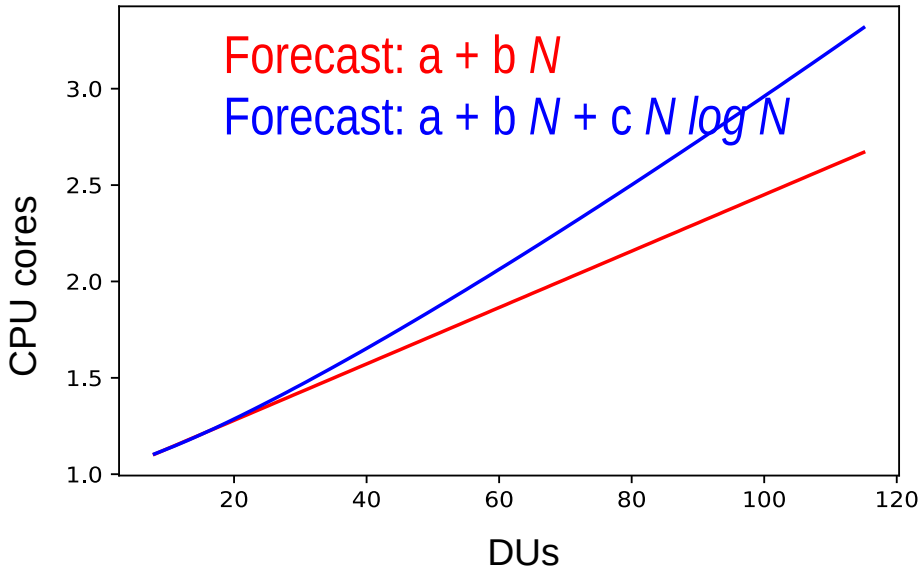
- GUI monitoring uses JSON data over HTTP
  - 4 parameters/CLB read for the GUI, all retrieved with a single HTTP query
  - Data extracted using a “direct path” access, i.e. without searching or sorting
    - Every GUI client establishes a direct reference to its list of monitoring variables

CU CPU load figures from ARCA

Dependency of CPU load of Detector Manager on the number of active Dus

- Measured in ARCA with regular activity (including monitoring GUIs that increase the baseline)
- Extrapolated to a full block





Heaviest activities in the DM include accessing sorted lists, for which a reasonable dependency is *N log N*. Other tasks are linear, and $O(N^2)$ tasks are limited to small sets.

CU overload protections 1/2

- CU services are unlikely to cause CPU overload
  - Only bugs could cause runaway CPU or memory consumption
    - **Always fasten your seat belt, accidents happen!**
- TriDAS processes require plenty of resources and already now use several servers
  - Each DataFilter/DataWriter has a single thread, so it can't saturate more than 1 core
    - **Intrinsic protection**
  - DataFilter memory buffers are statically allocated and controlled by resource allocation
    - **Intrinsic protection**

CU overload protections 2/2

- TriDAS processes require plenty of resources and already now use several servers
  - DataQueue memory buffers may grow indefinitely if there is insufficient computing power
    - Hardware failure of one or more DataFilter servers and downgraded operation
    - Specific circumstances that slow down processing
    - **LAP-based protection: the resident LAP of each server watches for overall memory consumption and stops/restarts all local processes if the available memory falls below a tuneable limit**
      - Data loss, but system recovers in few seconds
  - DataWriter memory buffers may grow indefinitely
    - Network/disk problems
    - **LAP-based protection: the resident LAP of each server watches for overall memory consumption and stops/restarts all local processes if the available memory falls below a tuneable limit**
      - Data loss, but system recovers in few seconds
      - Does not solve the cause but keeps the system responsive, (remote) interventions are possible

KM3NeT detector evolution

Entities to be driven in a KM3NeT detector – CU must be **flexible**!

Entity	Hardware	Firmware	Sub-firmware
DOM	CLB-v2	FirstGen	
DOM	CLB-v2	NewGen	
DOM	CLB-v4	NewGen	
Base	CLB-v2+BPS	FirstGen	BPSv1
Base	CLB-v2+BPS	NewGen	BPSv1
Base	CLB-v2+BPS	FirstGen	BPSv3
Base	CLB-v2+BPS	NewGen	BPSv3
Base	CLB-v4+BPD+BPC	NewGen	BPSv3_CU
Calibration Unit	CLB-v4	NewGen	BPSv3_CU
Instrumentation Unit	CLB-v4	NewGen	
More devices...	CLB-v4	NewGen	

CU code sustainability

Code development policy: **prefer readability and long-term maintenance over pure speed**

Codebase	C# on Mono CLR
External dependencies	Oracle Data Provider (access to remote DB)
Binary image tagging	Self-check on startup
Hardcoded constants	Absent
General constants	Centralized in dedicated libraries
Loops	Implemented almost anywhere with LINQ
Filters	Implemented with LINQ
Sort operations	Implemented with LINQ
Behavioral “if” chains	Replaced with tables of actions
State machines	Tables of lambda functions
Device controller source code	Single file (one per firmware variant)
Event reactions	Actions; lambda functions
Interprocess calls	HTTP library
GUI	HTML+JS on HTTP
Enforcing correctness	Static type checking; unit tests

Conclusions

- The Control Unit software for KM3NeT is a mature software suite...
- ... but still very lively!
  - Supporting new versions of firmware for devices
  - Supporting new devices
  - Adapting to more complex scenarios
- The mission of SW design and development:
  - Produce clear code that is easy to maintain
  - Prevent, mitigate or eliminate risks
- Performances are well under control
- Control Unit needs for 2 blocks (230 DUs) are likely to stay within one server
  - The architecture would anyway allow for multiple CU servers
  - For fault tolerance, more resources are already allocated and kept standing by

Maximize detector livetime!

Thank you for your attention!

BACKUP

CU code sustainability

Code development policy examples:

- Making logic clear by tables of actions instead of if/switch chains

```
Tuple<string, SubsystemState, Action>[] GetTransitions()
{
    return new Tuple<string, SubsystemState, Action>[]
    {
        new Tuple<string, SubsystemState, Action>(DeviceControl.StateMachineControl.Target.Run, SubsystemState.Error, new Action(this.Act_GetError) ),
        new Tuple<string, SubsystemState, Action>(DeviceControl.StateMachineControl.Target.Run, SubsystemState.Idle, new Action(this.Act_Init) ),
        new Tuple<string, SubsystemState, Action>(DeviceControl.StateMachineControl.Target.Run, SubsystemState.StandBy, new Action(this.Act_Configure) ),
        new Tuple<string, SubsystemState, Action>(DeviceControl.StateMachineControl.Target.Run, SubsystemState.Ready, new Action(this.Act_Conditional_Quit_Or_Start) ),
        new Tuple<string, SubsystemState, Action>(DeviceControl.StateMachineControl.Target.Run, SubsystemState.Paused, null ),
        new Tuple<string, SubsystemState, Action>(DeviceControl.StateMachineControl.Target.Run, SubsystemState.Running, new Action(this.Act_Conditional_Stop) ),
```

- Prefer functional programming style over procedural

```
_NextScheduleChangePoint = RunSchedules.SelectMany(x => new DateTime[] { x.Start, x.End }).Where(x => x > now).OrderBy(d => d).FirstOrDefault();
```

- Use static type checking to discover mistakes at compile time rather than at run time

```
else if (o is KM3NeT.CLBv2RemoteLite.BPSReadInfo bpsri)
{
    Program.Log("SRP log for " + clb.CLBUPI + " BPSReadInfo: " + bpsri.ToString(), LogInfoType.DebugL2);
    clb.EnqueueEvents(new CLBEvent[]
        {
            new evOnBPSVersion() { Timestamp = tst, Version = BPSFWVersionValue.V1 },
            new evOnOutSRPParam() { Timestamp = tst, ProcessVar = KM3NeT.CLBv2RemoteLite.ProcessVarInfo.BPSFWv1.BSE_BPS_I5, Values = new object[] { bpsri.I5VDC } },
            new evOnOutSRPParam() { Timestamp = tst, ProcessVar = KM3NeT.CLBv2RemoteLite.ProcessVarInfo.BPSFWv1.BSE_BPS_I12, Values = new object[] { bpsri.I12V } },
            new evOnOutSRPParam() { Timestamp = tst, ProcessVar = KM3NeT.CLBv2RemoteLite.ProcessVarInfo.BPSFWv1.BSE_BPS_I375, Values = new object[] { bpsri.I375V } },
            new evOnOutSRPParam() { Timestamp = tst, ProcessVar = KM3NeT.CLBv2RemoteLite.ProcessVarInfo.BPSFWv1.BSE_BPS_V375, Values = new object[] { bpsri.V375V } },
            new evOnOutSRPParam() { Timestamp = tst, ProcessVar = KM3NeT.CLBv2RemoteLite.ProcessVarInfo.BPSFWv1.BSE_BPS_IBACK, Values = new object[] { bpsri.IBackbone } },
            new evOnOutSRPParam() { Timestamp = tst, ProcessVar = KM3NeT.CLBv2RemoteLite.ProcessVarInfo.BPSFWv1.BSE_BPS_ALARM, Values = new object[] { bpsri.AlarmStatus } },
            new evOnOutSRPParam() { Timestamp = tst, ProcessVar = KM3NeT.CLBv2RemoteLite.ProcessVarInfo.BPSFWv1.BSE_BPS_BREAKER, Values = new object[] { bpsri.BreakerStatus } }
        }
    );
}
```