www.software.ac.uk

Online reconstruction (Manchego) Status report

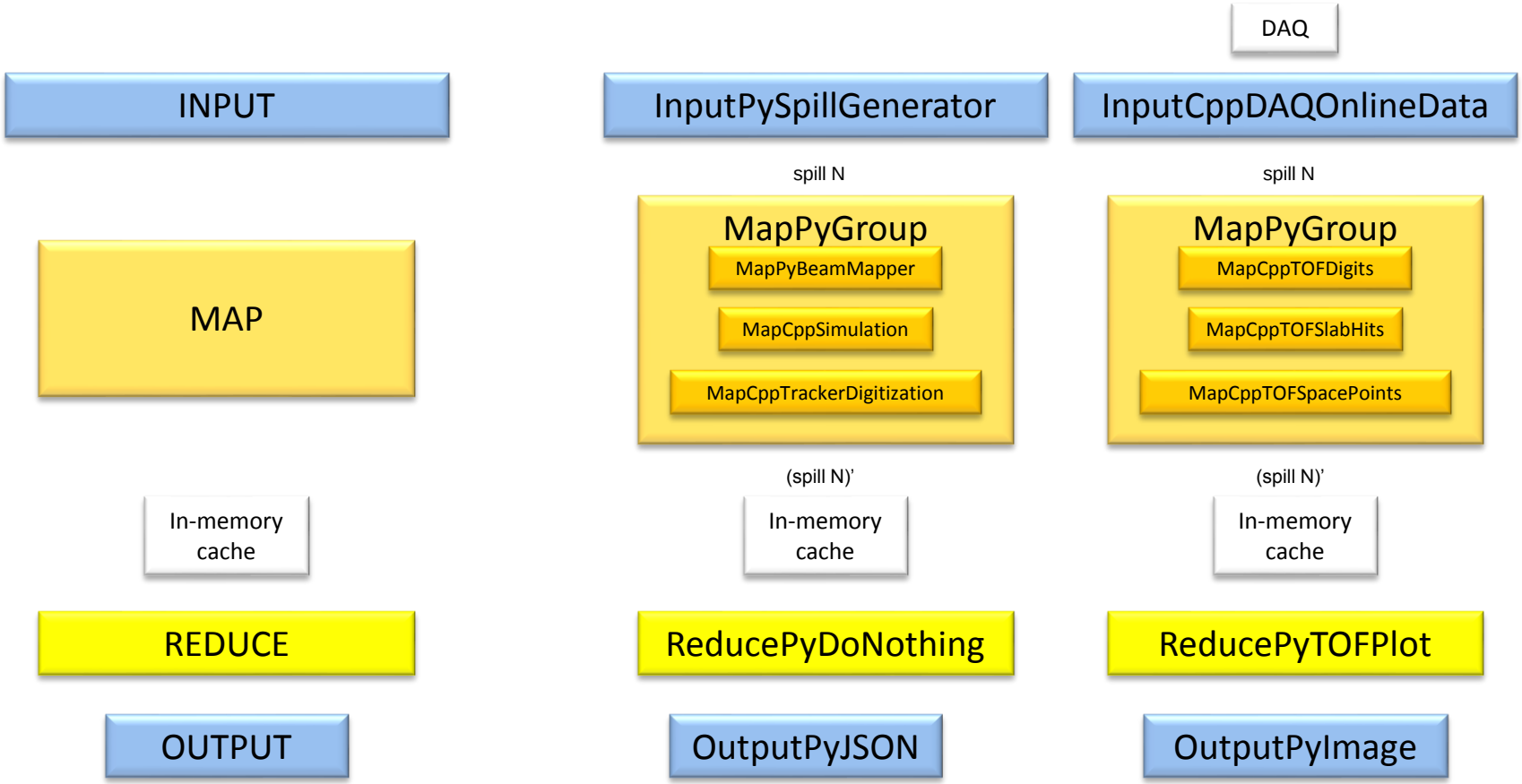
09/02/12

Mike Jackson

M.Jackson@software.ac.uk

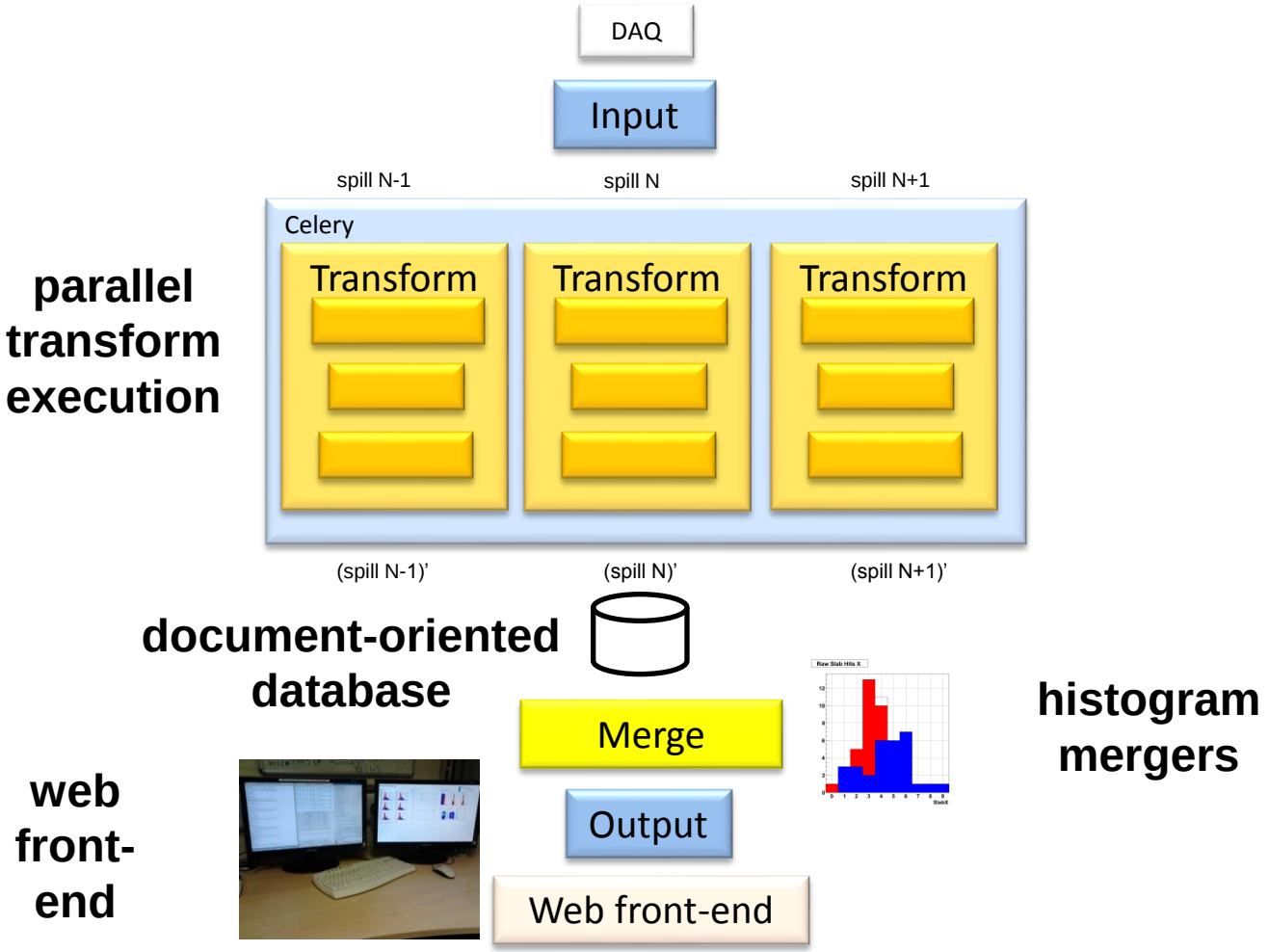
Architecture

www.software.ac.uk



Software development

www.software.ac.uk



parallel transform execution

document-oriented database

histogram mergers

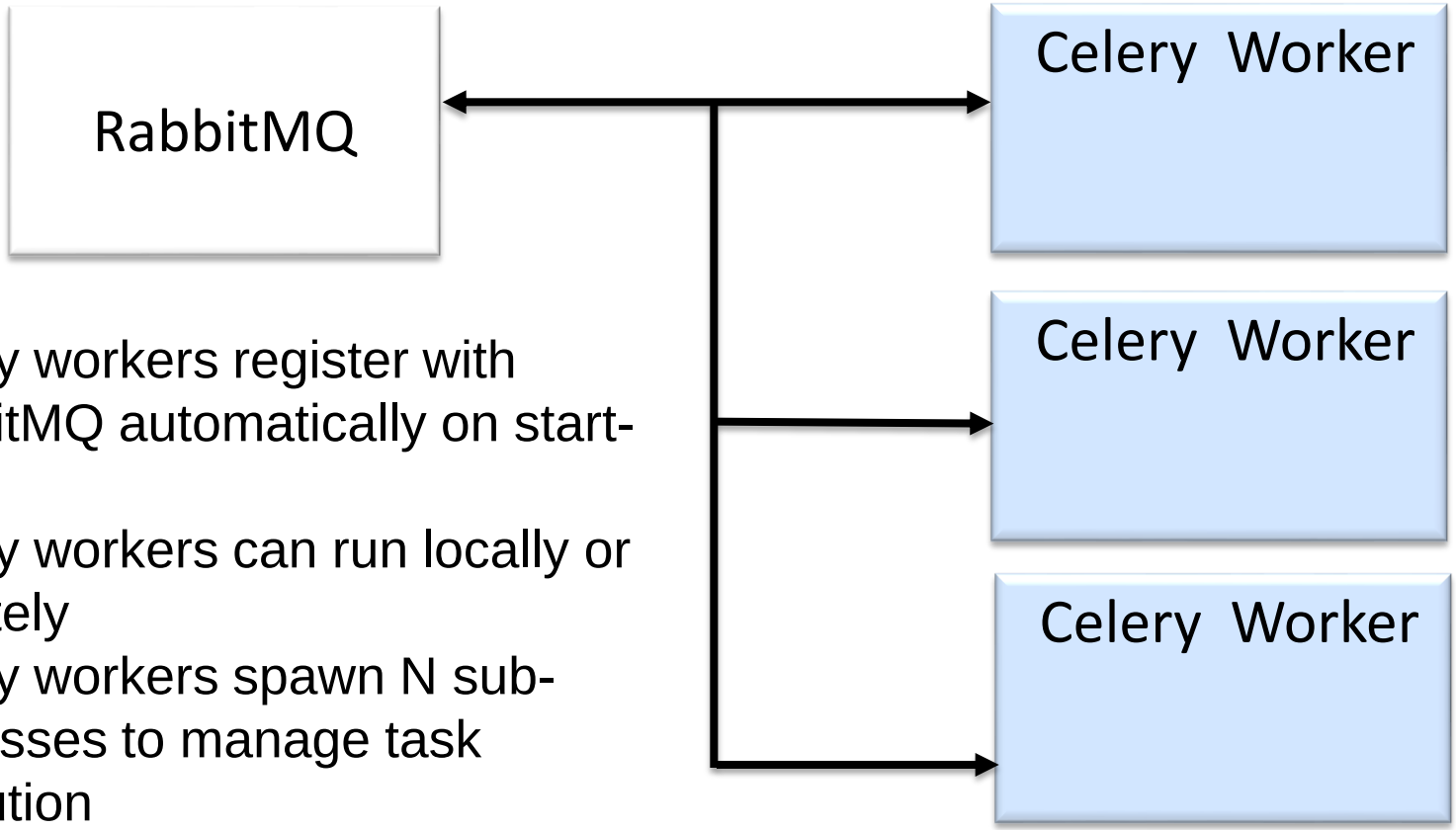
web front-end

Parallel transform execution

Parallel transform execution

- Spills are independent so can be transformed in parallel
- Celery
  - Python asynchronous task queue
  - Multi-processing
- RabbitMQ
  - Message broker

Celery and RabbitMQ

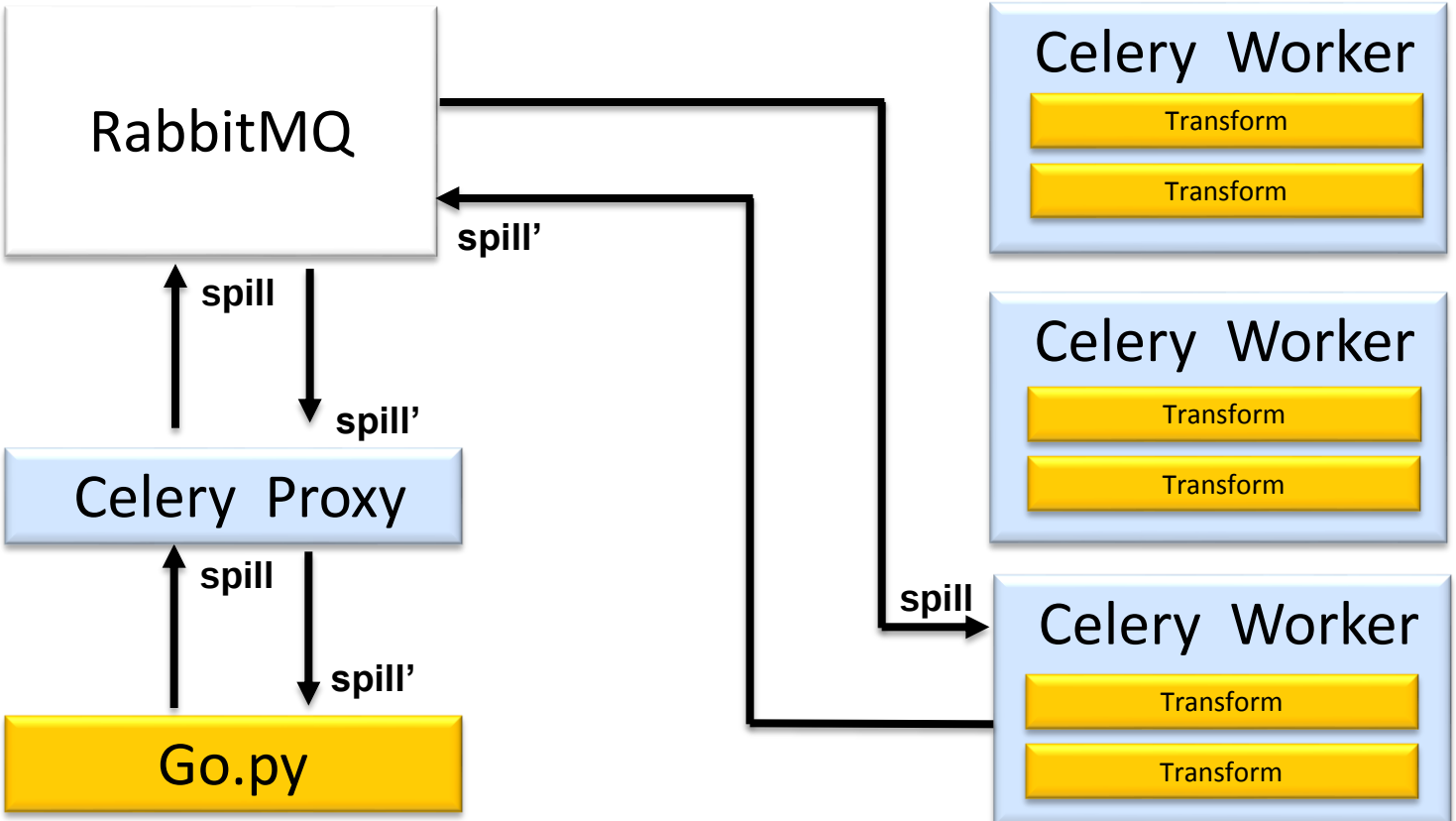


- Celery workers register with RabbitMQ automatically on start-up
- Celery workers can run locally or remotely
- Celery workers spawn N sub-processes to manage task execution

Celery workers and tasks

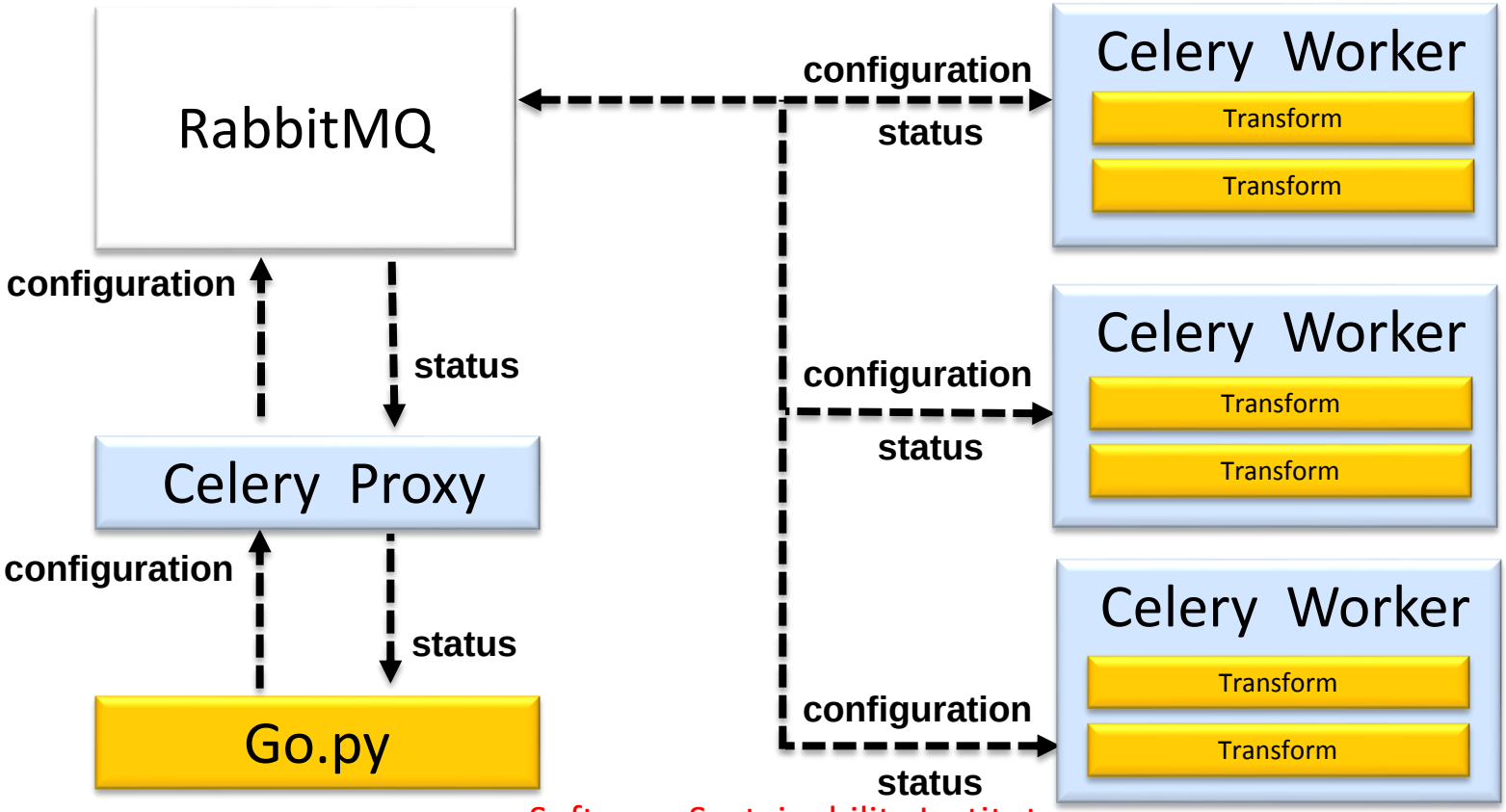
www.software.ac.uk

- Celery tasks (request to transform a spill) are dispatched to the next available worker
- Worker dispatches task to a free sub-process
- Each sub-process is configured to apply a transform
- Each worker has its own MAUS deployment



Celery workers and broadcasts

- Celery broadcasts are dispatched all workers
- Custom code to force broadcast into all sub-processes
- Broadcast is used to ensure all workers have the same MAUS configuration and transform – dynamic worker configuration



Celery workers

- Start up Celery worker executable
  - **`celeryd --c 2 --l INFO --purge`**
  - --c is number of sub-processes (default is number of cores)
  - --l is default Celery logging level
  - --purge clears any backlog of messages from RabbitMQ
- Celery spawns sub-processes
  - Up to --c value
  - Sub-processes execute tasks i.e. transforms

Dynamic configuration and Go.py

- Uses reflection to get transform name(s)
  - MapPyGroup(MapPyBeamMaker, MapCppSimulation, MapCppTrackerDigitization)
  - “Transform specification”
- Invokes a Celery broadcast
  - Transform specification + MAUS configuration + configuration ID (Go.py process ID)
  - Waits for a maximum of 5 minutes to hear from workers
- Synchronisation
  - Celery workers and Go.py client should run the same MAUS version

Dynamic configuration and Celery workers

- Celery worker main process
  - Receive broadcast command
  - If configuration ID has changed
    - Crams transform specification/configuration down to sub-processes
    - If all sub-processes update correctly then main process saves the configuration ID, transform specification/configuration too
  - If sub-process dies, a new sub-process will spawn with the current configuration
  - Catches and converts any exceptions
    - Avoid non-Pickleable exceptions from causing unexpected errors
- Celery worker sub-processes
  - Death existing transform
  - Create and birth new ones
  - Catches and converts any exceptions

Transforming spills and Go.py

- execute_transform
  - Celery task to execute MAUS transforms
  - Client-side proxy
  - Submits spill to RabbitMQ
  - Returns AsyncResult to Go.py
- Polls AsyncResult
  - Status – SUCCESS, FAILURE, PENDING
  - Results – the transformed spill
  - Errors

TODOs

- Document error messages that can appear in Celery worker terminal window
  - Draft already done but out-of-date due to recent reimplementation
  - http://micewww.pp.rl.ac.uk/projects/maus/wiki/MAUSCeleryRabbitMQRecovery
- Relate to a “recovery” guide for control room

Document-oriented database

Document-oriented database

www.software.ac.uk

- Cache spills between input-transform and merge-output phases
- Products
  - CouchDB
    - (id, document)
    - 0.1.0 –yum install
    - 1.1.0 –day-wasting unable to build from source experience
  - MongoDB
    - Collections of (id, document)
    - Latest version –yum install

Document-oriented database

- Go.py currently can use both or just cache spills in-memory
- Current naïve algorithm
  - Read spill from database
  - Pass to merge-output
  - Delete spill from database

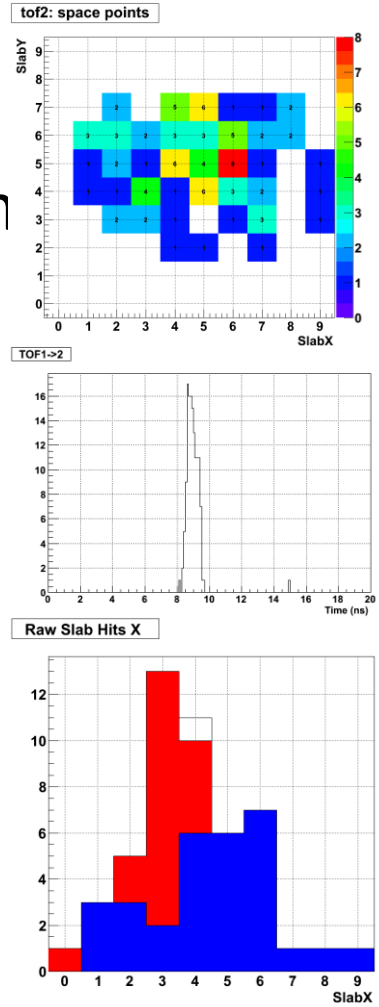
TODOs

- Resolve how document cache is used within a single run
  - By single instance of Go.py running merge-output?
  - By multiple instances of Go.py running merge-output?
- Determines
  - How spills are identified
  - How they're marked as having been "reduced"
  - When they can be deleted
- Update Go.py appropriately

Histogram mergers

Histogram mergers

- Aggregate spill data and update histogram
- Super-classes for graph packages
  - Matplotlib – ReducePyMatplotlibHistogram
  - PyROOT – ReducePyROOTHistogram
- Examples:
  - ReducePyHistogramTDCADCCounts
  - ReducePyTOFPlot (Durga)
- Mergers do **not** display the histograms



Histogram mergers

- Configuration options
  - Image type e.g. EPS, PNG, JPG,…
  - Refresh rate e.g. output every spill, every N spills
  - Auto-number image tag
- Output JSON document
  - Base64-encoded image data
  - Image tag used for a file name
  - Meta-data e.g. English description

Image outputter

- OutputPyImage
- Configuration options
  - Filename prefix
  - Directory
- Extract and save base64-encoded image data
  - Image file e.g. EPS, PNG, JPG,…

TODOs

- Image JSON document
  - Add keywords field
- OutputPyImage
  - Save image and JSON document with meta-data
  - JSON document becomes part of the online reconstruction-web server API

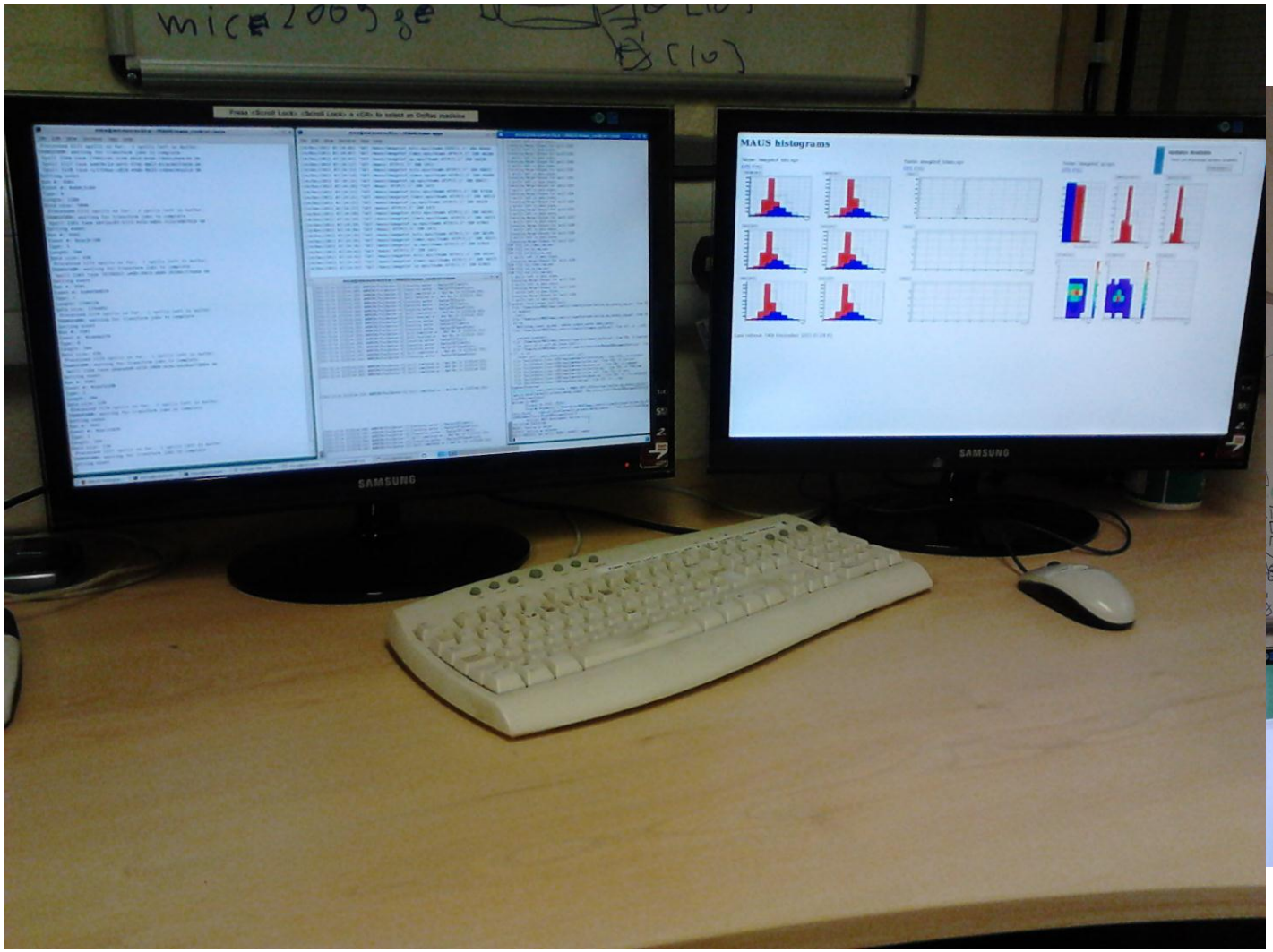
Web front-end

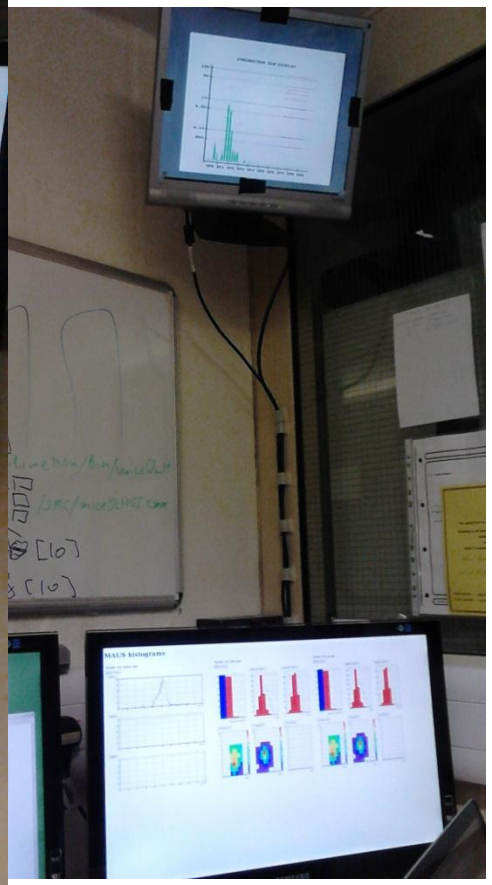
Web front-end

- Django
  - Python web framework
  - Refresh every 5 seconds
  - Currently using Django test web server
- Serve up images from a directory
  - “API” between online reconstruction framework and web front-end is just this directory
  - Can run web-front end anywhere so long as images are made available “somehow”

Current state

www.software.ac.uk





TODOs

- Web server
  - Deploy under Apache 2.2 and mod_wsgi
  - Render images and meta-data
  - Search-by-keyword option
- Extensible
  - Customise layout and presentation later