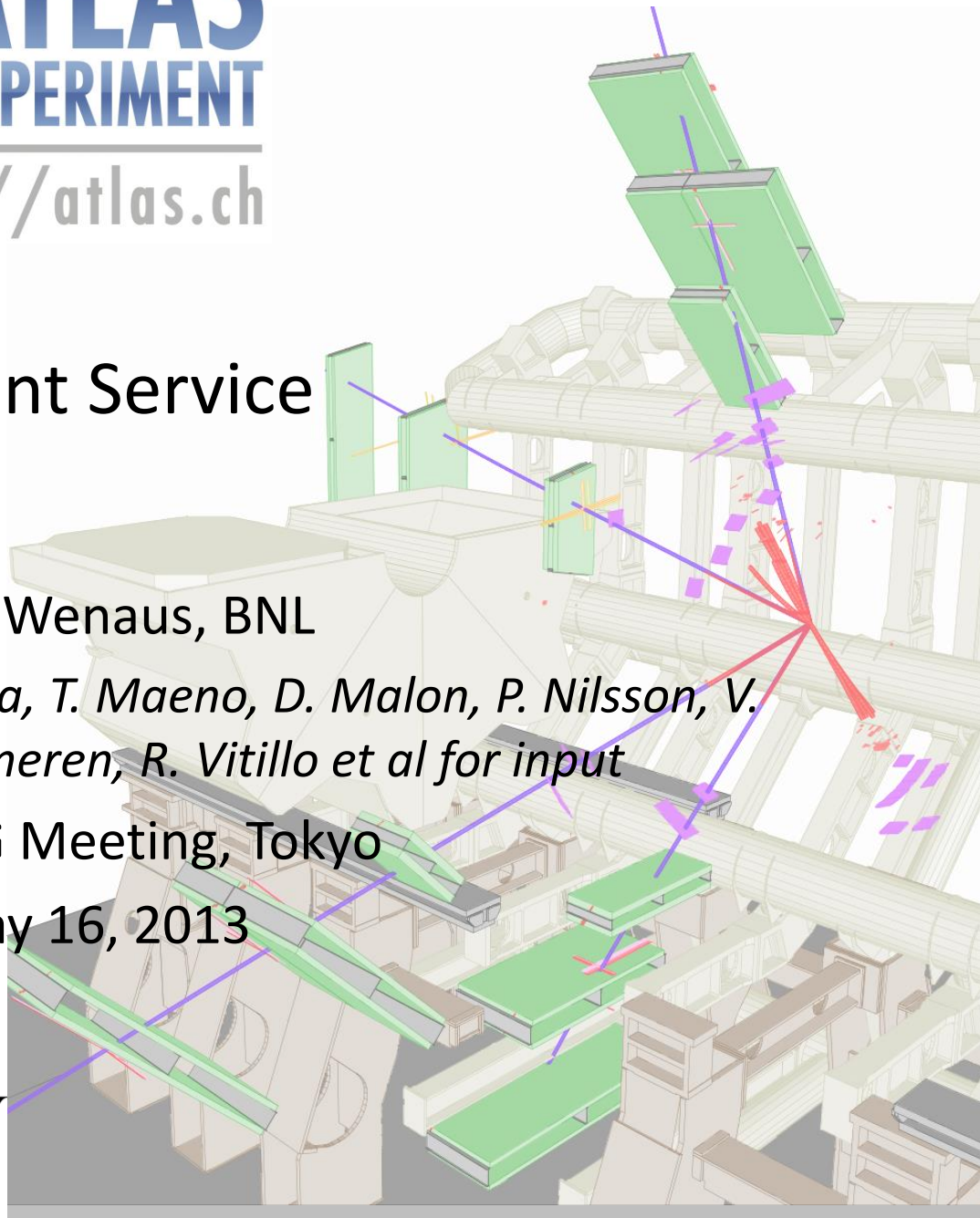


Event Service

Torre Wenaus, BNL

Thanks to K. De, P. Calafiura, T. Maeno, D. Malon, P. Nilsson, V. Tsulaia, P. Van Gemmeren, R. Vitillo et al for input

ADC TEG Meeting, Tokyo

May 16, 2013

BROOKHAVEN NATIONAL LABORATORY

Opportunistic resources: an important means of expanding our computing throughput

- HLT during LS1, and opportunistically after LS1
- Allocations on supercomputers (HPC)
  - Over-engineered for us but we can soak up free cycles opportunistically
    - In the US for example our funding agencies are insistent that we do so – we are high-profile science
  - Free research clouds, commercial clouds
    - More/cheaper resources available if we can be highly opportunistic (e.g. Amazon spot market)
- Others we haven't explored yet
  - e.g. 'ATLAS@Home' via BOINC?
  - By one estimate, BOINC = ~500k volunteer computers, ~7 Petaflops, ~$5M/hour in 'Amazon EC2 dollars'
    - (from Eric Lancon heard at a FR-cloud meeting in Beijing)

Leveraging opportunistic resources

- Common characteristic to these opportunistic resources: we have to be agile in how we use them
  - Quick onto them (software, data and workloads) when they become available
  - Quick off them when they're about to disappear
  - Robust against their disappearing under us with no notice
  - Use them until they disappear – don't allow holes with unused cycles, fill them with fine grained workloads
- Means of enabling agile use of opportunistic resources: **an event service**

Event Service

- JEDI brings event-level workflow management
  - Event server support is part of the JEDI design
- Tasks and jobs are managed in terms of events processed/to be processed
- Invites allocating work to pilots that way, rather than allocating work in bulk (input files or big chunks of input files)
- Serve pilots/payloads a steady stream of events, and send outputs back in a steady stream
  - Well suited to opportunistic resources: no heavy data prestaging, and we can make a hasty exit, losing almost nothing
- Predicated on workable event streaming: effective WAN data access is essential, preferably buffered by async caching

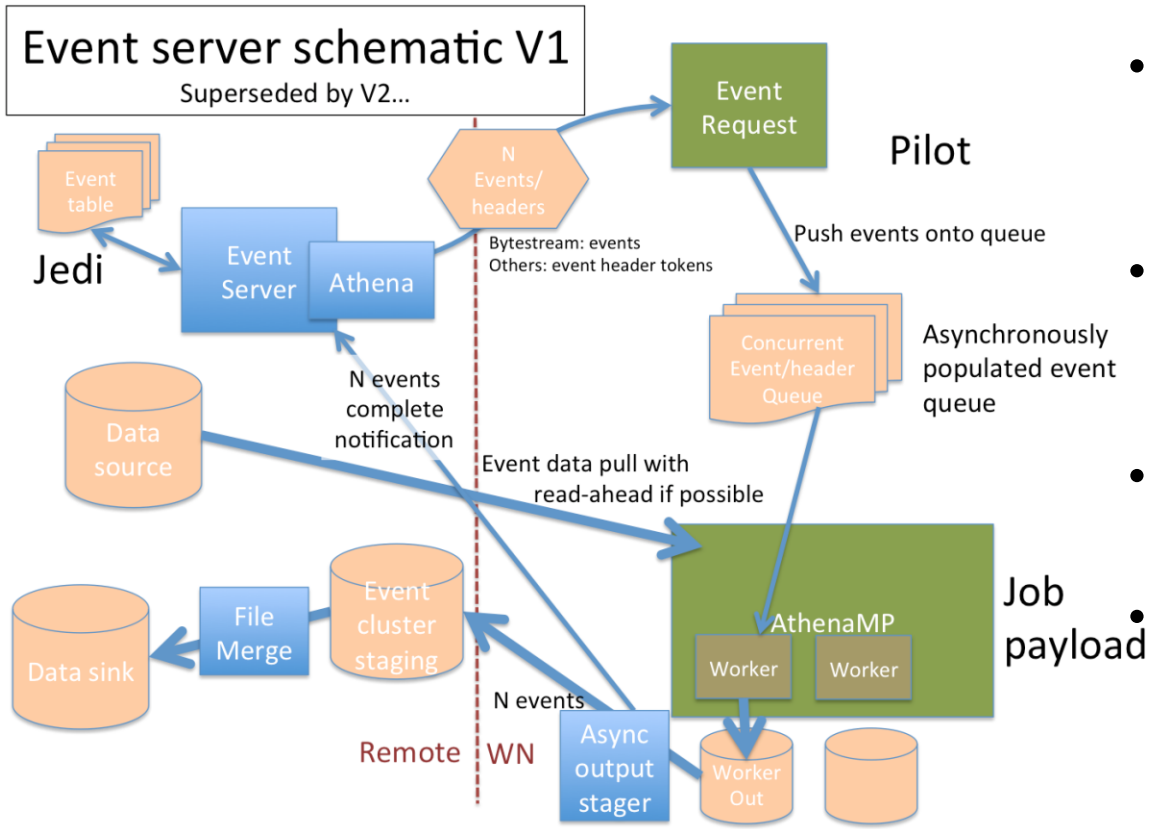
Event Service and Core Software

- Just as JEDI is an enabler for the event server scheme on the grid side, athenaMP and associated I/O developments and optimizations are enablers on the core sw side
  - Queueing and streaming events to be consumed by workers
  - I/O optimizations making event reading over WAN practical
  - Asynchronous pre-fetch to remove network latencies from processing workflow
- Event server support being actively worked by core services/event I/O experts

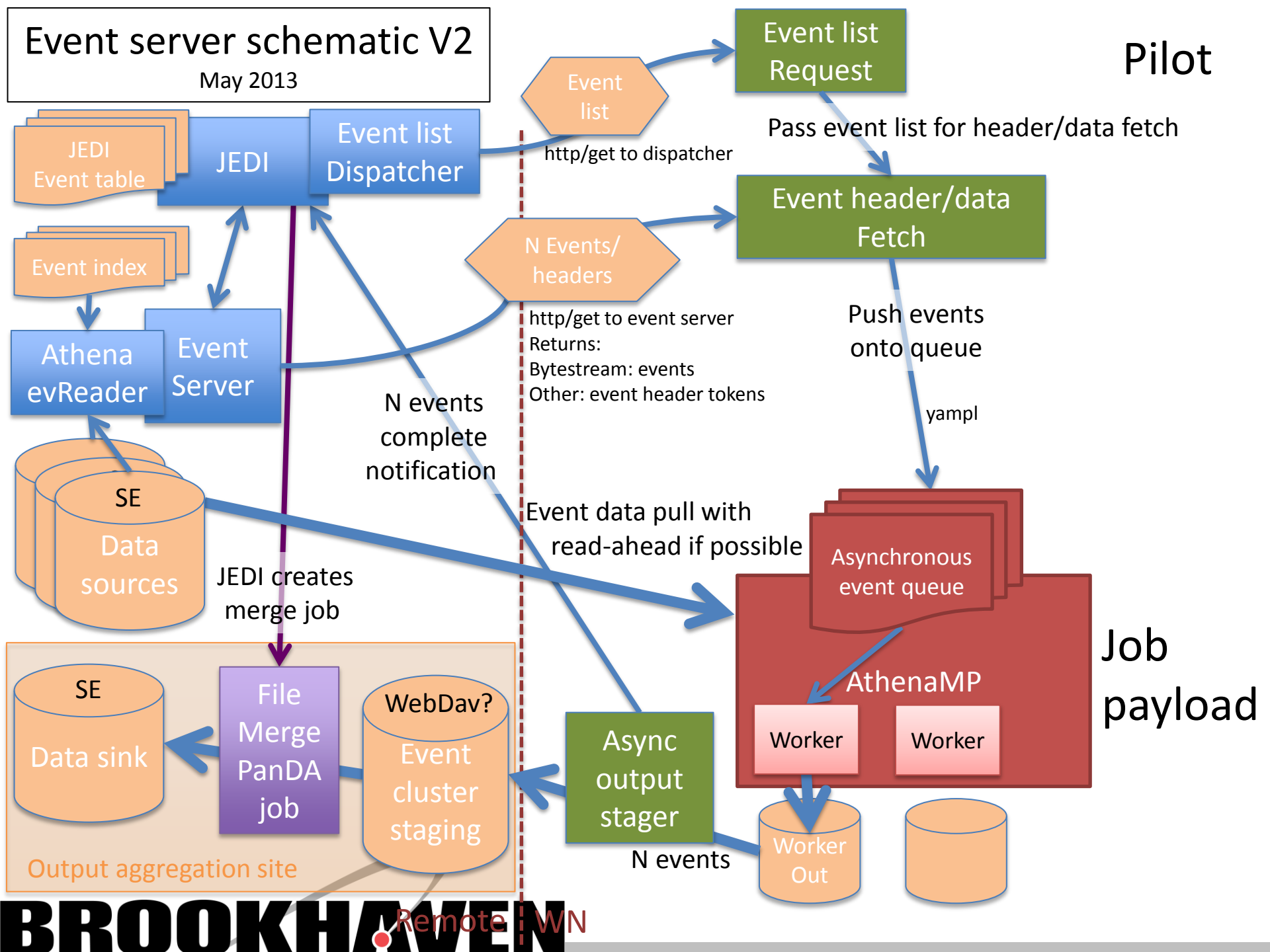
Other event service advantages

- Avoid idle cores in athenaMP processing – if parallel threads process events in large chunks (files), cores can sit idle while the slowest one completes
- DDM simplification – no DDM involvement on input or output
- Output merging flexible and simple: file size is tunable, aggregation of outputs to merge site proceeds concurrently with processing, JEDI knows when to trigger merge
- A crash doesn't lose the job, only a few events which will be re-dispatched

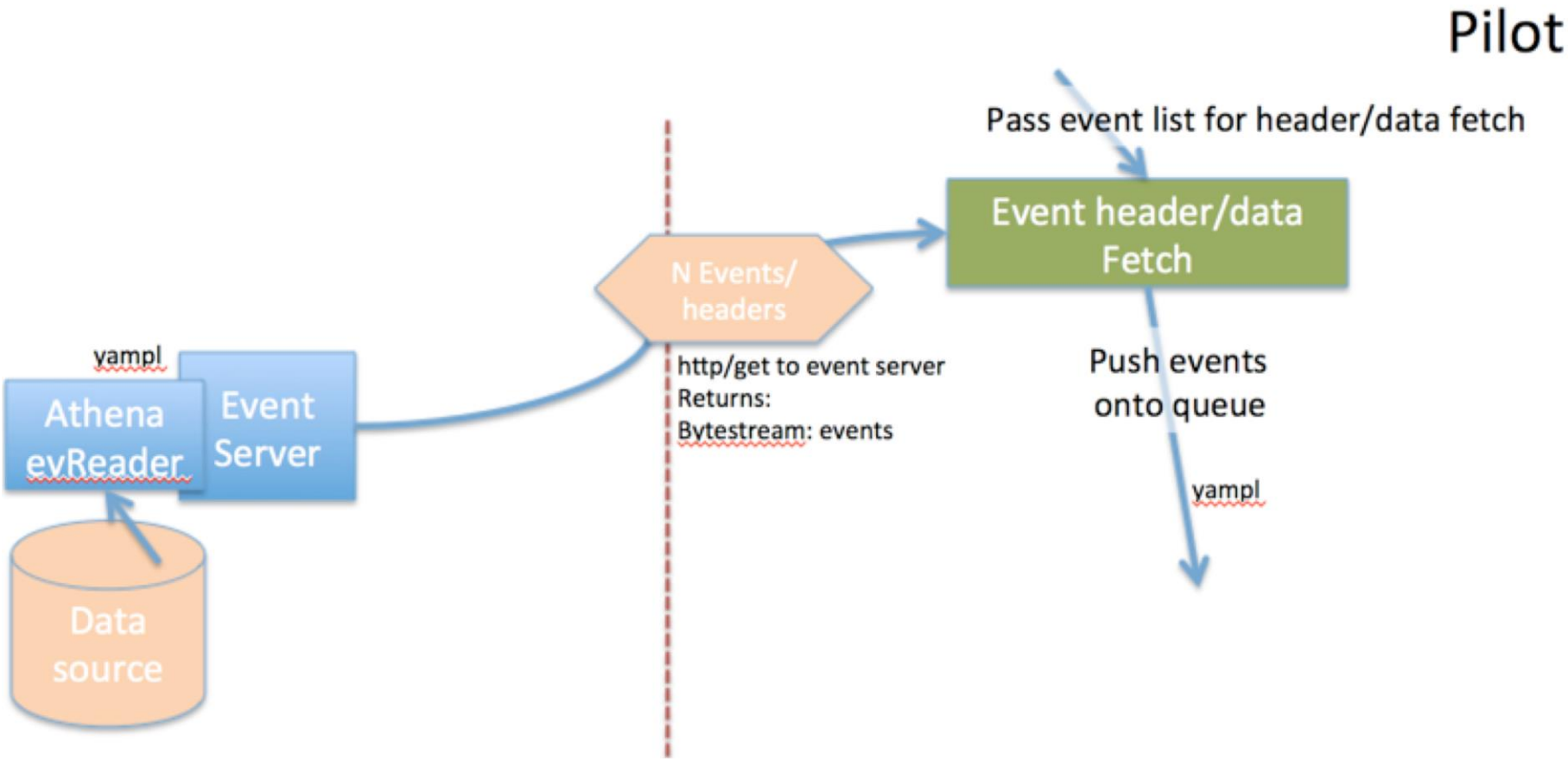
Developing the scheme



- Initial scheme from Borut and Andrej, 6/2011
- Agreed to be integral to Prodsys2 design at Ljubljana meeting 6/2012
- Informal discussions in subsequent months: ADC, core framework, event I/O
- Design V1 sketched out at 3/13 S&C week
- Subsequently elaborated, latest discussion last week


Event server schematic V2
May 2013
Pilot
Event list Request
Event list
Pass event list for header/data fetch
JEDI Event table
JEDI
Event list Dispatcher
http/get to dispatcher
Event header/data Fetch
Event index
N Events/ headers
Athena evReader
Event Server
http/get to event server
Returns:
Bytestream: events
Other: event header tokens
Push events onto queue
yampl
N events complete notification
SE
Data sources
Event data pull with read-ahead if possible
Asynchronous event queue
JEDI creates merge job
AthenaMP
Worker
Worker
Job payload
SE
Data sink
File Merge PanDA job
WebDav?
Event cluster staging
Async output stager
Worker Out
N events
Output aggregation site
BROOKHAVEN
Remote WN

Prototype Status (reported last week)



V. Tsulaia, R. Vitillo, P. Van Gemmeren

Prototype Status

- Prototype of the event reader and the event server web service
- Event reader reads (currently) bytestream events
- Web server is stateless Apache python/WSGI service with RESTful API
- At startup the event reader creates a YAMPL (message passing) channel to the event server web service, listening for requests
- When request is received from web service, it reads the requested event data from file and constructs a buffer sent over YAMPL to the web service, and thence to the requestor (pilot)
- Pilot client is emulated by curl

V. Tsulaia, R. Vitillo, P. Van Gemmeren

Issues – How to send events

- Send full events (as in prototype) or just event addresses (headers/tokens) from the event server?
  - Full events makes for fewer client/server interactions. But…
  - Increases substantially the scaling requirements for the event server (moving MBs instead of kB)
  - Loses opportunity for dispersing data retrieval for scalability, and leveraging direct access (FAX)
- For now at least, agreed to send event addresses; events will be retrieved from the client (pilot+athena) in a subsequent step

Issues – Scope/scale/role of event server

- Event reader has to resolve an event ID from JEDI into headers/tokens rendering the event retrievable
- Obvious (current) way is for event reader to read the actual file (so should be colocated with the data or will expose WAN latencies in ID->token resolution)
- Colocation is rather a big demerit – would be preferable to have many fewer event service instances than there are data repository Ses
- Colocation & file open unnecessary if we have an Event Index
  - Direct, remote, fast ID->token resolution
  - Lookup could even be done by the pilot, eliminating need for event reader service?

Issues – Managing outputs

- Outputs must be managed in a near real time, granular way just as inputs are, in order to reap full benefits
  - Minimal losses in case of sudden eviction from or disappearance of opportunistic resource
  - Output aggregation and monitoring can proceed concurrently with task processing
- Present concept:
  - athenaMP workers write few-event files to WN output directory
  - Watching monitor thread picks up event files and sends them to (webDAV?) aggregation point associated with the WN's CE (may not be local; e.g. may be at T1). On successful transfer, notifies JEDI that those events are processed; JEDI so records in the bookkeeping
  - JEDI knows state of completion and decides when to submit PanDA job for merging of event files

Issues – Pilot aspects

- Adapt present ‘job dispatch’ stage to retrieve the job configuration to be used (determining what sort of event processing athenaMP is to be configured for)
  - A given pilot+payload will not be reconfigurable for a different processing type, at least initially
- New ‘event dispatch’ stage to iteratively request events
  - A runEvent module analogous to existing runJob
- Adapt RunJob to new event consumer role
- Output watcher/completion reporter as a new ‘monitoring’ thread parallel to payload execution
- Need testing scheme – HC jobs + dev pilot?

Prototype next steps (Vakho, Roberto)

1. Agree on message content, format between JEDI, event server for communicating event IDs, e.g. include file ID in the request
2. Start implementing a new AthenaMP tool, which will work with events received from the pilot and pass the events to the workers
   - NB! Presently we cannot run Athena job with no input file. This would require serious cleanup/modifications in various places.
   - So, for the time being the AthenaMP job will be given some input RAW file only to start, initialize and then switch to the "waiting" mode for events from the pilot
3. Look at some scalability aspects of the event server/reader
   - Currently the Apache server dynamically tunes the number of processes, which encapsulate the web service, based on the number of pending requests. We need to see how this scales for hundreds or thousands of requests
   - Will need to see how many reader applications we can afford running simultaneously. Just one reader serving thousands of services will clearly create a bottleneck, running thousands of readers accessing disk simultaneously is not going to work either. [Considerations that make using Event Index attractive.]

Conclusion

- Prodsys2 is designed to enable efficient & flexible resource usage in the (even more) heterogeneous computing environment we're entering – multicore, serial, opportunistic, HPC, …
- Together with concurrency-directed core software work, it enables event servers for efficient use of these resources
- Leverages existing work in athenaMP, event I/O optimization, and FAX
- Effective WAN direct access to data is an enabler and a prerequisite
- Prototyping and development under way (as other priorities permit)
- Plenty of issues to address (only a subset discussed here)

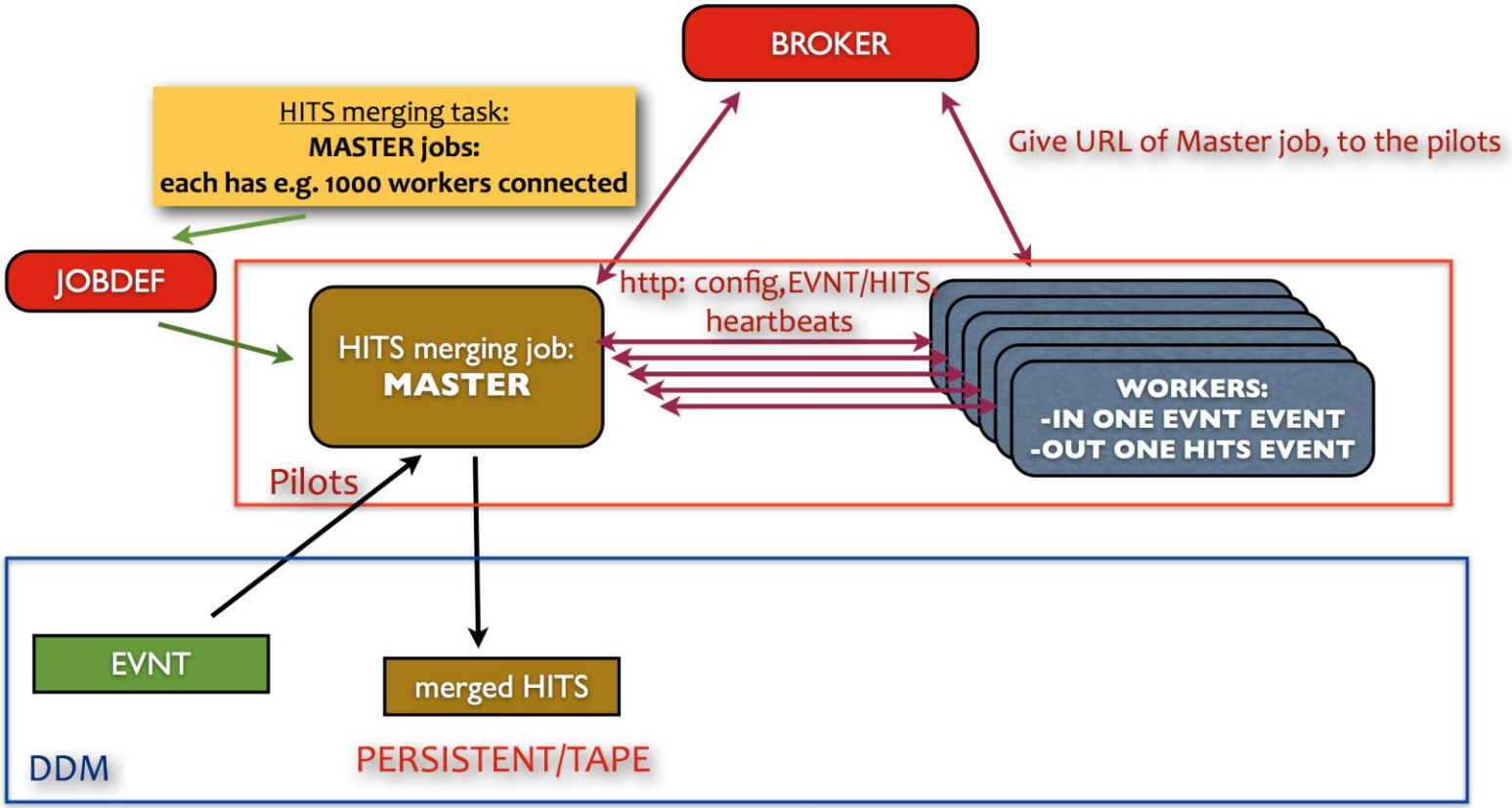
More Information

- “ADC issues in concurrent processing and maximizing computing throughput”, T. Wenaus, March 2013 S&C Week

Supplemental

Somewhat crazy idea

- I had a series of chats with Andrej, a brief chat with Simone and even shorter one with Rod - they don't think it's total nonsense so here goes in a diagram, please read the next slide for details.



Somewhat crazy idea (2)

- 1) the 'master' is submitted via panda as a regular 'HITS merging' job, with an EVNT file as input and e.g. 1000 evts/job . What the transform (and athena) inside it do:
  - athena sets up an http server (for example, the one built-in into python)
  - it reads the EVNT file and offers the events to 'workers' which query it
  - it gets the HITS events posted from workers and writes them to a file (sort of an SFO).
  - it offers the G4 configuration to run if workers ask (based on the prod. tag, so easy).
  - it does some bookkeeping of the events (needs to know something about errors and timeouts on the workers).
  - when 1000 events are reached (or some events abandoned due to crashes) it finishes writing the HITS file and stops like a regular job.
- 2) the workers are submitted as modified (extended) pilots, long lived:
  - i.e. the pilot asks for a job from the broker and if it gets the job ID/URL of a master (somehow it needs to know it's not a regular job) it connects to the master and asks for setup.
    - these jobs not defined by jobdef, their number variable.
  - it initializes the G4 (which takes time so we save time with long-lived workers)
  - asks master for events (one at a time) and sends hits back.
  - when the master has gone it asks the broker for a new master with the same configuration (G4 AMI tag), if there are none it shuts down (or re-inits with a different config, as we want it).
- 3) the broker needs to know that the merging/master jobs are special and pass the info about them to slaves (e.g. job id + some additional flags, maybe G4 AMI tag) but otherwise nothing special (I think).
- **Advantages:** no small HITS files (DDM simplified); if worker crashes/runs out of time we loose 1 event; simpler to use on non-dedicated resources.

- **Caveats:** Not sure how much work would take to make athena do I/O via http, according to Rolf not insourmountable. No clue how many changes the panda (broker) etc would require..

Borut, Andrej 6/2011