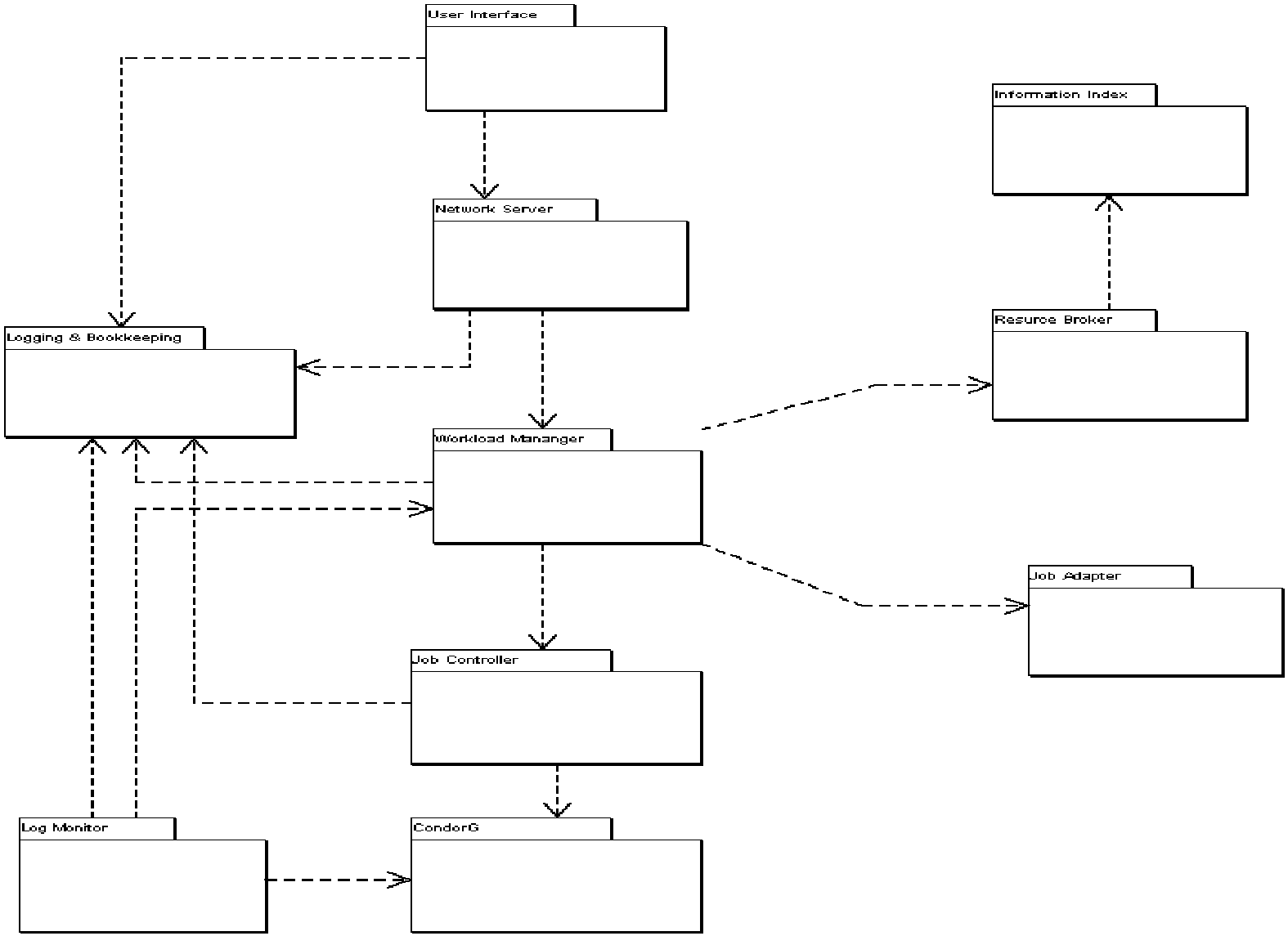
The EDG Workload Management System: Architecture for Release 2.1

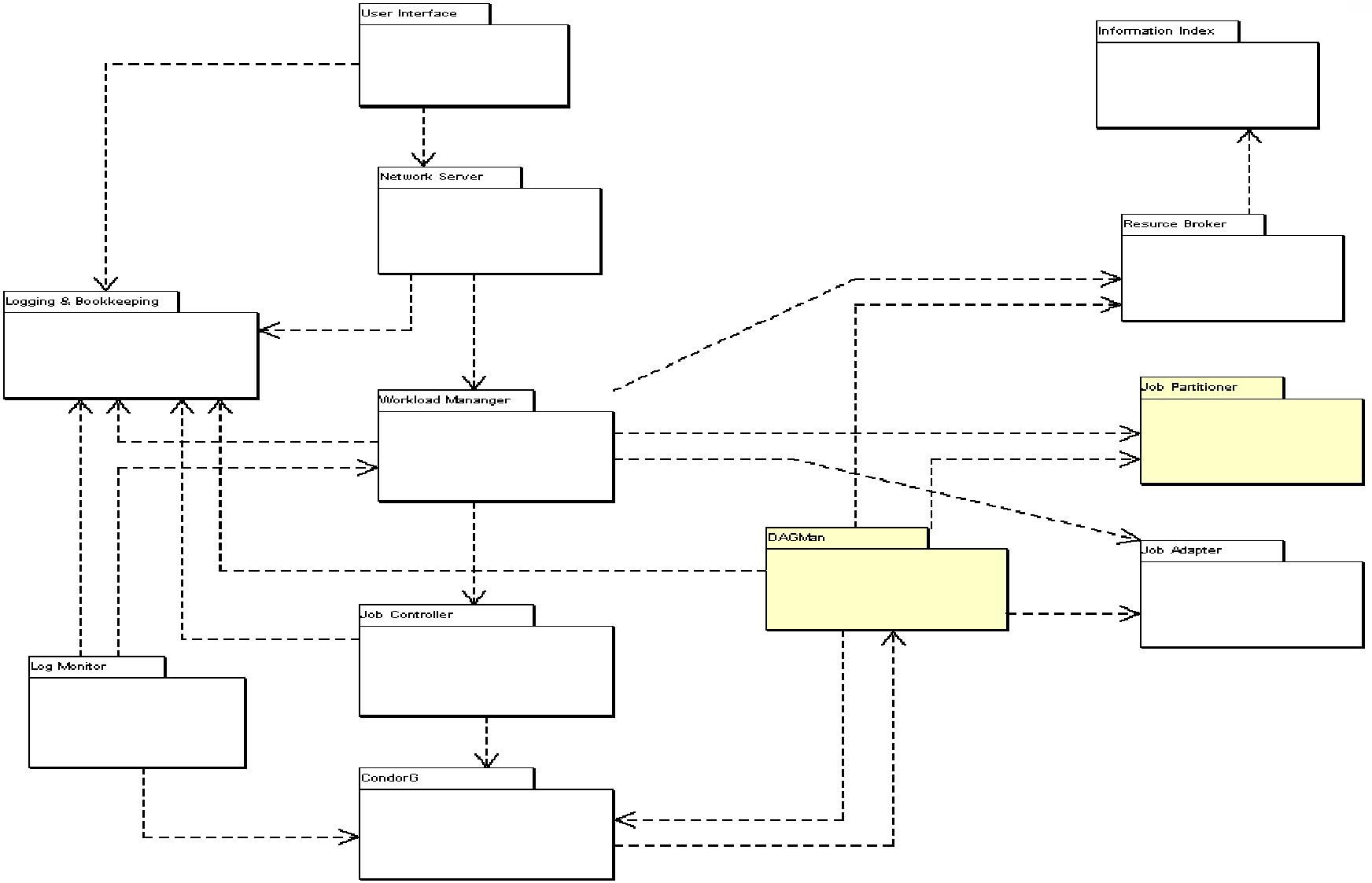
Fabrizio Pacini
Datamat SpA - EDG WP1

fabrizio.pacini@datamat.it

WMS architecture for rel 2.0



WMS architecture for rel 2.X



DAGMan (1/7)

- DAGMan stays for "DAG Manager"
  - A DAG is a Directed Acyclic Graph whose nodes are job and whose arcs represent dependencies (temporal) between them.
- DAGMan can be seen as an iterator on a graph of jobs. It periodically checks the new entries in the Condor log file and acts accordingly
  - review dependencies
  - when a job is free of dependencies it can be submitted to CondorG for its execution.
  - choice of the resource where to submit has to be done
- DAGMan Provides a recovery mechanism
  - if a DAG fails, a recovery file is created. This recovery file stores the so-called *rescue DAG*: this is the original DAG, where the already completed jobs have been marked as done

DAGMan (2/7)

- Scheduling Approaches
  - **Eager**: resolve each job specification at DAG submission time
  - **Lazy**: resolve a job specification at job submission time
  - **Very lazy**: resolve a job specification at run-time, i.e. the job is bound to a resource where the resource is already acquired.
- We feel that the later a job is bound to a resource the better it is but:
  - The "very lazy scheduling" would heavily affect the current WMS design
  - We have adopted the "lazy scheduling"
- The implementation of the lazy model implies that a job is bound to a resource just before it is passed to CondorG.
  - To support this, DAGMan has to call one or more Helper modules, along the same lines adopted within the Workload Manager

DAGMan - the whole path (3/7)

- We provide a way to specify DAGs with the same Job Description Language (JDL) used for normal jobs.
  - a DAG consists of a ClassAd, where the attribute “Type” is set to "DAG"
  - it contains a set of ClassAd attributes, each one representing a job
  - Each job is specified using the usual attributes
- Apart from the temporal dependencies, the jobs represented by the various nodes are independent: each one has its own executable, input, output, requirements, preferences, etc.
- The JDL received from the UI, representing the DAG, is first passed by WM to the Job Adapter to create the Condor DAG submit file
- Then the DAG is submitted to CondorG that invokes DAGMan to handle the DAG

DAGMan - the whole path (4/7)

- DAGMan navigates the graph, determines which nodes are free of dependencies and performs the following steps for each dep-free job composing the DAG:
  - calls the Resource Broker, in the same way as it is usually called by the WM for "single" jobs, to perform the match making
  - the result of the call is the original JDL augmented with the resource choice
  - DAGMan then also calls the Job Adapter to make the final adjustments to the job description and generate the JobWrapper script
  - Lastly DAGMan submits the job to CondorG

DAGMan (5/7)

- A DAG is considered as a single unit of work
  - It is therefore addressable with an identifier
  - The identifier is assigned by the WMS and used like a single job identifier
  - It is represented (having its own entries) in the LB
  - It can be managed (e.g. killed or monitored) like a single "normal" job
- The "visibility" of the single jobs (i.e. nodes) composing a DAG is also provided, and therefore users can, for example monitor such single jobs
- When a DAG node fails the DAG is considered as failed
- A DAG goes through a number of states during its lifetime, similar to the single job ones. Some of the job states are not meaningful since for example there isn't scheduling for a DAG as a whole, but only for its member jobs.

DAGMan (6/7)

- DAG State Machine:

  - SUBMITTED the user has submitted the DAG, using the UI
  - WAITING WM has accepted the job, the Job Adapter is called
  - READY the DAG has been submitted to CondorG, but a DAGMan process has not been started yet to manage it
  - RUNNING a DAGMan has been started for the DAG
  - DONE the DAG has been completed (this does not mean that all the jobs have completed successfully) or the DAG has been explicitly terminated by the user. If at least one job in the DAG did not complete successfully, a rescue DAG has been produced
  - ABORTED the DAG has been aborted for external reasons
  - CLEARED the user has retrieved successfully all the output files. Bookkeeping information is purged some time after the job enters this state.

DAGMan (7/7)

- DAGMan provides a recovery mechanism …
  - if a DAG fails, a recovery file is created. This recovery file stores the so-called *rescue DAG*: this is the original DAG, where the already completed jobs have been marked as done.
  - when resubmitted, the rescue DAG will be executed only for the jobs that are still to be run (i.e. the ones not marked as done)
- … the WMS exploits it
  - Upon DAG failure, the submission of the rescue DAG is triggered automatically (without requiring user intervention) by the Workload Manager, if requested to do so.

Job Partitioning (1/5)

- The idea:
  - the processing of a job should be described as a set of independent steps/iterations.
  - It must be stressed that the hypothesis is that the various steps are independent,
- The goal:
  - exploit this characteristic, considering different, simultaneous, independent sub-jobs, each one taking care of a step or of a sub-set of steps, and which can be executed in parallel.
  - The partial results (that is the results of the various sub-jobs) can be represented by job states (the final job checkpoint states of the various sub-jobs), which can then be "merged" together by a *job aggregator*
- Details in D1.4

Job Partitioning (2/5)

- When a user submits a partitionable job to the Grid,
  - he must specify, using a specific JDL attribute, which are the independent steps/iterations "composing" the job.
  - Besides specifying the characteristics and the requirements of the partitionable job, the user can also specify the "job aggregator" (the job responsible for collating and merge together the results for the various sub-jobs in which the original job is being partitioned)
  - and a "pre-job", that is a job which must be executed before the various sub-jobs:
- the JDL expression representing a partitionable job is transformed in a JDL expression representing a Directed Acyclic Graph (DAG) of jobs

Job Partitioning (3/5)

```
JobType = "Partitionable";
Executable = ...;
JobSteps = {"step_1",……,"step_N"};
StepWeight = {weight1,……,weight_N};
Requirements = ...;
        ...
Prejob = [
         Executable = ...
         Requirements = ...;
         ... ]
Aggregator = [
         Executable = ...
         Requirements = ...;
         ...  ];
```

- **JobSteps** is used to list the independent steps

- **StepWeight** is used to specify the "weights" for the various elements of the list; this is a "hint" given by the user to help the Grid middleware in performing a good partitioning.

Job Partitioning (4/5)

- The component converting the partitionable job into a DAG is the "Job Partitioner" which is called by the WM when it receives a request of type "partitionable job"
  - the initial list of steps will be partitioned in m sub-lists, where m represents the total number of available CPUs for all the CEs satisfying the requirements for this job
  - each sub-list will have *n/m* elements (unless weights are specified).
  - a DAG composed by *m*+2 nodes is defined: *m* nodes representing the m sub-jobs (which can be executed in parallel) and 2 other nodes, representing the pre-job (which is executed before the *m* sub-jobs), and the job aggregator (which can start its execution when the other *m* sub-jobs have terminated their run).
- The DAG returned by the Job Partitioner is treated by the WM as described in the previous slides (DAGMan)
  - The DAG is submitted to CondorG … …

Job Partitioning (5/5)

- Each sub-job will have the same *Requirements* and *Rank* of the initial partitionable job
- Both the *m* sub-jobs and the job aggregator will rely on the WP1 checkpointing API
- Before exiting, each sub-job will have to save, using the *save_state* method, its final state.
- When all the sub-jobs have completed their execution, and they all have saved their final state, the job aggregator can start its execution
- This job will be given (through the proper JDL attribute) as the initial set of iterations the list of the identifiers of the final states of the various sub-jobs. It will have to "retrieve" these final states of the sub-jobs, and then perform some processing to collect together these partial result

Conclusions

- Very likely DAGMan and Partitioning are the only new functionalities that will be introduced in WMS for September release (together with VOMS integration)
  - Final decision will be taken at next WP1 meeting
- Some work already done for DAGMan
  - Last issues will be addressed at next WP1 meeting (begin July)
- Job Partitioning strongly depends on DAGMan integration
- Both components are pluggable:
  - No impact on the control flow for "normal" jobs