Understanding and mastering dynamics in computing grids: processing moldable tasks with user-level overlay

Mościcki, J.T.

Publication date
2011

CHAPTER 4

Development of the User-level Overlay

Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.

E.W. Dijkstra

Make everything as simple as possible, but not simpler.

A. Einstein

In this Chapter[1] we explain the development strategy and architecture of a User-level Overlay which implements the late-binding task processing model described in the previous Chapter. The User-level Overlay consists of loosely-coupled, reusable and customizable services and tools based on GANGA and DIANE software packages. Such a strategy allows to easily develop high-level, application-specific functionalities such as heuristic resource selection and adaptive task scheduling.

[1]The results described in this Chapter formed the basis of the following papers: J. Mościcki, F. Brochu, J. Ebke, U. Egede, J. Elmsheuser, K. Harrison, R. Jones, H. Lee, D. Liko, A. Maier, A. Muraru, G. Patrick, K. Pajchel, W. Reece, B. Samset, M. Slater, A. Soroko, C. Tan, D. van der Ster, and M. Williams. Ganga: A tool for computational-task management and easy access to Grid resources. *Computer Physics Communications*, 180(11):2303 – 2316, 2009; V. Korkhov, J. Mościcki, and V. Krzhizhanovskaya. The user-level scheduling of divisible load parallel applications with resource selection and adaptive workload balancing on the Grid. *IEEE Systems Journal*, 3:121–130, March 2009.

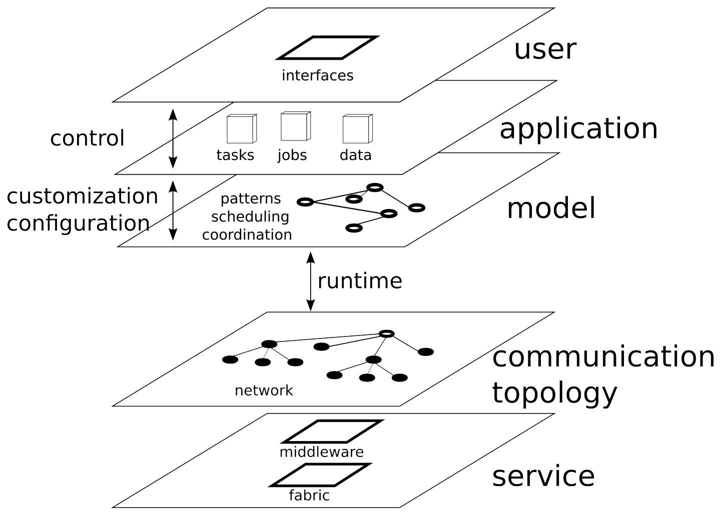

Figure 4.1: Layered model for grid computing includes both system and application domains. Separation between application and model is to emphasize the fact, that many applications follow common scheduling patterns and coordination methods and thus may take advantage of tools which support generic application execution models.

4.1 Vision

Emergence of large-scale production grids has changed the way computing capacities and capabilities are provided to the applications, simply because resources are not placed in a unique administrative domain and the scale and complexity of these systems tends to be overwhelmingly large. For example, traditional techniques developed for clusters, such as administrative fine tuning of scheduling, or prioritization of access to resources on a basis of individual applications, are not anymore practical. Instead, grid environment naturally falls into two somewhat distinct domains: a system layer which includes infrastructure and generic middleware services and an application layer which includes user jobs, applications and tools (Fig. 4.1). An attempt of mastering grid dynamics inevitably shifts our focus onto the application layer as a most likely place where such issues may be addressed, also because there are many grids and many middleware stacks, and we may sometimes need to use them simultaneously.

We use the term *overlay* to emphasize the fact that we build a layer above existing middleware services and across available grids and other distributed infrastructures without modifying them. Overlay in this context refers to a model layer in Fig. 4.1, which consists of interconnected software elements which span *above* the system domain. Layering is a key for overlays: grid users do not control resources and they have little influence on choice, life-cycle, upgrades and availability of middleware services. In many cases it is important that application-specific needs are accommodated at timescales convenient to the application community. These timescales are often not correlated at all with the timescales convenient to the resource and middleware providers. Moreover, deployment and operation of large-scale *production* grids is radically different from practices in grid computing *research* which does not make it easier to apply new scientific concepts in production environments[2].

With grids being multi-institutional federations of resources, trust relationships between resource providers and consumers are a delicate issue. Resource providers are not easily convinced to install application-specific services in their sites, as it often requires privileged access and special procedures negotiated as Memoranda of Understanding or Service Level Agreements [4]. Therefore, we use the term *user-level* to emphasize the fact that tools and services provided by the User-level Overlay live entirely in the user-space and do not require special access rights for installation and operation. Moreover, we require that all components of the User-level Overlay should use single user credentials. This is to be completely compliant with the grid security model which assumes accounting and traceability at the user level.

For this strategy to be effective only minimal assumptions on available grid services and infrastructures should be made. We envisage the User-level Overlay as a self-contained bundle of tools and services which travel together with the application code to be run by remote execution services. At runtime the User-level Overlay transforms into a layer of interconnected components, spanning multiple grid sites. To achieve this we rely on outbound network connectivity to be available on the worker nodes in a grid. Outbound connectivity is the least common denominator of virtually all distributed infrastructures which are currently in production.

Late-binding overlays raise concerns of resource owners about traceability and fair-share use of the system. For example, VO-centric overlays sometimes allow mixing of worker agents and tasks of different users, which may result in problems of distinguishing who is the owner of the application running on a particular resource. On the other hand, this feature allows the VOs to manually control resource sharing policies for their members, however, it requires a special agreement between a VO and resource providers which is only feasible for large user communities. A User-level Overlay is free of this limitation as the worker agents may only execute tasks which belong to the same user. In addition, running another set of tasks typically requires another set of worker agents jobs to be submitted and therefore the User-level Overlay does not bypass the fair-share mechanisms provided by grid middleware and underlying batch/fabric layers.

It is clear that no single turnkey solution may fit all applications across all distributed infrastructures. Support of flexible and incremental development of application-specific functionalities and abstractions and easy adaptation and customization by a user is the key. Flexibility is needed at the application level as well as at the user level to support increasingly complex use-cases and scientific workflows. Therefore, in our vision the system becomes an application-hosting environment where software plugins may be composed to best fit the purpose of a specific application. At the same time default

[2] Anecdotal evidence of this has been given by one of our operations/deployment colleagues at a coffee break. He was once talking to a known researcher in grid computing who claimed that "to solve your problems with the middleware component X you should *upgrade your testbed* to take advantage of this entirely new middleware architecture that we recently developed". The only possible answer was "Well, the fact is that we don't have a testbed but a production system with 10^5 CPUs, 300 sites, 3×10^5 jobs and several thousand users every day."

plugins should be provided off-the-shelf to the end users for standard computing tasks.

With the appearance of modern scripting languages such as PYTHON or Ruby the choice of implementation language becomes an essential part of the design [123]. As observed in [148], "with additional basic tools, PYTHON transforms into a high-level language suited for scientific and engineering code that's often fast enough to be immediately useful but also flexible enough to be sped up with additional extensions". PYTHON primarily acts as glue, or steering layer, for extension modules which allow to use high-performance compiled code. On the other hand, PYTHON features, such as dynamic typing, runtime decoration, introspection and extension of objects allow to achieve enormous degree of flexibility. Clear syntax allows to express ideas and algorithms as human-readable programs even with limited prior knowledge of the language. Therefore, this particular choice of the implementation language is an important part of our strategy. To emphasize this fact we chose to present algorithms using PYTHON syntax.

4.2 Functional breakdown and architecture

The requirements to be addressed, presented in Sec. 1.4, include: 1) improvement of the Quality of Service, 2) cross-infrastructure interoperability, 3) easy access and transition between different distributed environments for end users and 4) application-specific scheduling and coordination. These requirements may be mapped into the following high-level functional areas:

1. *late-binding task scheduler* for optimization of workload distribution and task coordination,

2. *resource selection algorithms*,

3. *resource access interface* for uniform access to remote resources,

4. *end-user environment* for easy application configuration and management of user jobs.

Fig. 4.2 shows the high-level outline of these functional areas where late binding is a key element. Resource access interface enables to allocate computational resources and to create dynamic resource pools which may span multiple distributed infrastructures. A resource pool consists of processes running on remote worker nodes which are directly controlled by the task scheduler. Resource selection component allows to control the number and quality of the resources in collaboration with the task scheduler. End-user environment provides easy ways for application configuration and management.

A relation between an application and the late-binding scheduler is one of the key issues: a scheduler may be embedded into the application or external to it. A scheduler embedded into the application is developed and optimized specifically for a given application, typically by re-factoring and instrumenting the original application code. It allows fine tuning and customizing the scheduling according to the specific execution patterns of the application. Such a scheduler is intrusive at the application source code

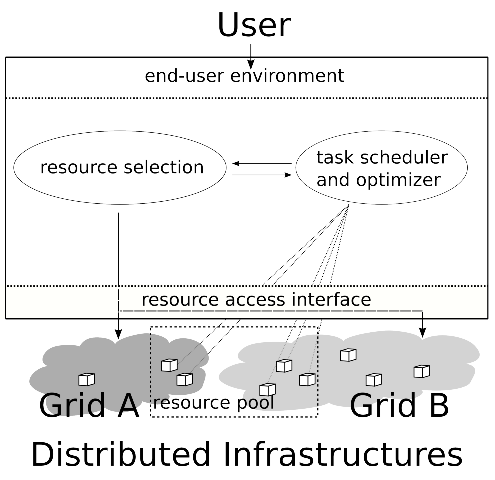
User
end-user environment
resource selection
task scheduler and optimizer
resource access interface
Grid A
resource pool
Grid B
Distributed Infrastructures

Figure 4.2: Main functional components of User-level Overlay. Resource selection component is used to construct a resource pool through a uniform resource access layer. Task scheduler and optimizer uses the late-binding method to operate directly on a dynamic resource pool. Upper layer provides end-user environment shielding users from underlying complexity of distributed infrastructures.

level which means that the code reuse of the scheduler is reduced and the development effort is high for each application. A scheduler external to the application relies on the general properties of the application such as a particular parallel decomposition method (e.g. task parallelism, iterative decomposition, geometric decomposition or divide-and-conquer). An application adapter connects the external scheduler to the application at runtime. Depending on the decomposition method, the application re-factoring at the source code level may or may not be required. The disadvantage of external schedulers is that it may be very hard to generalize execution patterns for irregular or speculative parallelism. In this case, which occurs in various situations ranging from medical image processing to portfolio optimization [172], a development of a specialized embedded scheduler may be necessary. Any particular solution is a trade-off between usability, flexibility and efficiency. Our User-level Overlay adopts an approach which is mid-way between a generic scheduling service and an embedded scheduler tightly-coupled with the application code.

4.3 DIANE and Ganga software packages

The User-level Overlay consists of loosely-coupled components which have been developed as two separate software products:

- Diane – providing a task scheduler,
- Ganga – providing a resource access interface and an end-user environment.

Diane [133, 134] is a task coordination framework which exploits the late-binding method. Diane provides an application-aware scheduler which may be extended by a system of plugins to support master/worker workloads such as task farms and bag of tasks. Plugins for DAGs [82] and data-oriented workflows [76] have been implemented as third-party contributions by interested user communities. The framework also supports customized, application-specific processing methods and failure-management strategies.

Ganga [136] is a job submission interface to access distributed computing resources in an easy and uniform way. It is an open and extensible framework which allows to submit jobs through a system of plugins and it currently supports Portable Batch System (PBS) [88], Load Sharing Facility (LSF) [163], Sun Grid Engine (SGE) [72], Condor [170], gLite [113], ARC [58], Globus/GRAM [48], GridWay [91] and the SAGA API [77] standard. Ganga has a double role in the User-level Overlay. Firstly, it provides a scripting API to programmatically access remote resources and may be used as a job management abstraction layer. Secondly, Ganga provides a user-oriented working environment for application configuration and management of computational tasks.

4.4 Operation of the User-level Overlay

In the Ganga/Diane overlay user workload is split dynamically in a number of tasks which are scheduled for execution to a set of worker agents (Fig. 4.3). A scheduler is a part of the master server which is managed by the user as a personal, transient service. Worker agents communicate directly with the master which, by default, automatically stores the results in the local storage area directly accessible by the user. When processing is terminated the overlay is automatically destroyed and resources released.

Ganga interface is used to control the worker agents which may be submitted, killed and removed as simple batch jobs. When started, worker agents pull the tasks from the master which controls the distribution of work. The system is fault-tolerant and may run autonomously: a worker agent which fails to complete the assigned calculations is replaced by another worker agent. Late-binding allows to reduce the scheduling overhead and to dynamically adapt to changing workload and evolving pool of available resources.

Resource selection is controlled by the user with the scripting interface of Ganga. Several simple submitter scripts are provided which allow the user to submit a specified number of worker agent jobs directly from the system shell. Worker pool may also be controlled automatically by Agent Factories – specialized submitter scripts which provide streams of worker agent jobs over longer time periods. Agent Factories may implement advanced resource selection schemes based on application feedback and in response to the performance requirements of the application which are provided by the task scheduler. At the conceptual level, loose coupling between resource selection and task scheduling allows the quality, provenance and number of computational resources (worker agent jobs) to be controlled independently of the number of tasks at the application level.

At the software level, loose coupling between Ganga and Diane is also beneficial

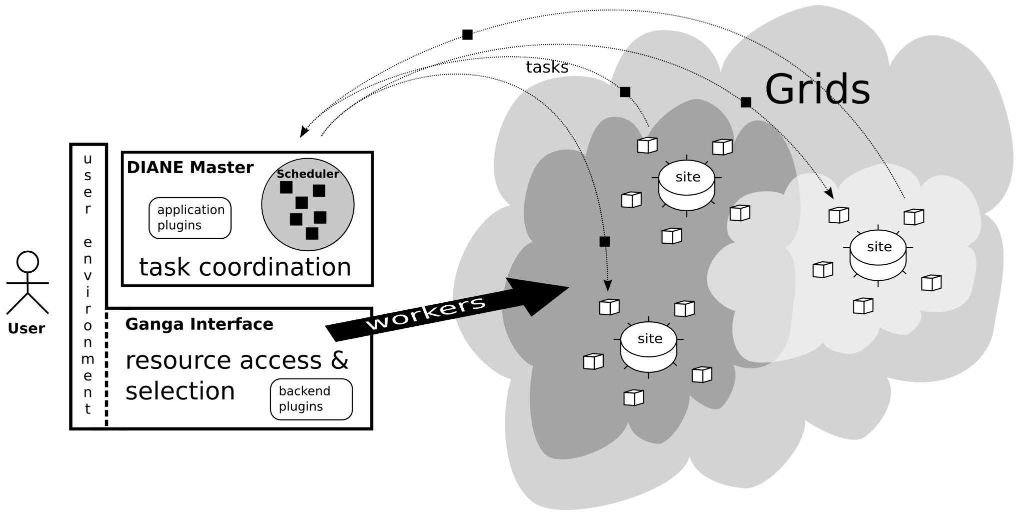

Figure 4.3: Architecture of the User-level Overlay operating on multiple grids at the same time. Task coordination in a Master/Worker model is provided by Diane. Resources are acquired by sending worker agent jobs using Ganga interface. Multiple clouds represent different computing environments and grids. User has a direct control over Ganga and Diane components and plugins.

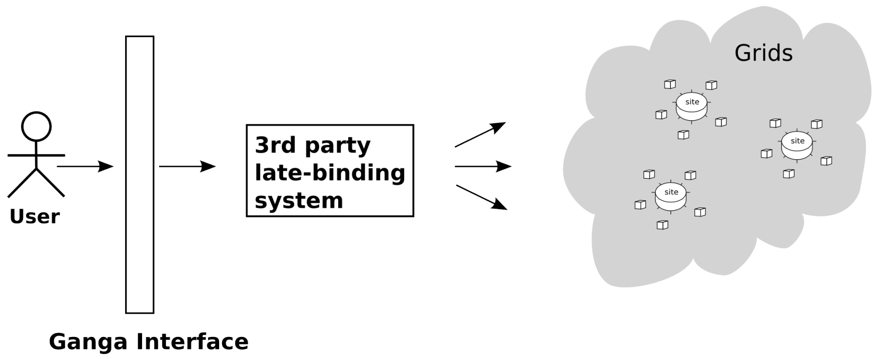

Figure 4.4: User-level components in an alternative configuration: third-party late-binding task management system replacing Diane. Ganga provides an interface which hides complexity of actual implementation of task management system and underlying distributed infrastructures.

as it allows to replace specific functionalities with other implementations, thus making integration into existing domain-specific frameworks more easy. A user may choose a component to be used separately or to be replaced by third-party solution as shown in Fig. 4.4 where DIANE is replaced by a third-party task management system. For example, this configuration is used in large High Energy Physics communities, where third-party task management is implemented as VO-centric overlay (discussed in the next Chapter). In this case users are interacting with the task management system via GANGA.

4.5 The DIANE task coordination framework

The DIANE task coordination framework is based on a simple distributed programming model which follows the Master/Worker pattern. The implementation uses PYTHON and a high-performance Object Resource Broker, omniORB[3].

DIANE model relies on an ability to partition the computational problem into parametric tasks which are executed independently as presented in Fig. 4.5. Tasks are defined as basic execution blocks which convert a set of input parameters into a set of output parameters. Execution block may be a sequence of programming instructions, an invocation of an executable, a set of OpenMP threads or MPI processes to accomplish the task. The tasks may be created dynamically at runtime or arranged in more complex structures to express data flow, execution dependency or specific ordering in the task execution model.

The ensemble of computation consists of many tasks and it is called *a run*. The system consists of many worker processes which communicate with one master process (the worker processes do not need to share the filesystem nor memory). A task operation is specified by a set of parameters which are produced by the RunMaster (running on a master node) and consumed by the WorkerAgent (running on a worker node). The worker produces the output and sends it back to the master. The RunMaster keeps track of tasks in order to react to failures in the task execution, spawns and coordinates new tasks.

A built-in FileTransferService provides a simple but reliable way of transferring files between RunMaster and the WorkerAgents. It provides a simple PYTHON-based API which allows to download and upload files without file-size restrictions and may be readily used in the DIANE component plugins. FileTransferService may be used to overcome limitations of certain middleware implementations, such as gLite WMS service which typically restricts a single file size to 10 MB.

4.5.1 Layered architecture of DIANE

The framework consists of three layers, as shown in Fig. 4.6:

1. application and scheduling layer,

2. core framework,

[3]http://omniorb.sourceforge.net

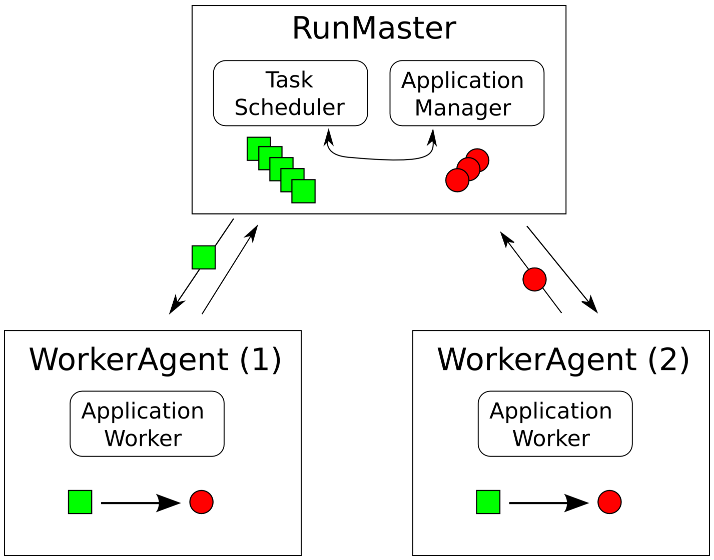

Figure 4.5: Task coordination framework architecture. RunMaster schedules tasks for processing by WorkerAgents using the TaskScheduler component (task inputs are shown as squares). Output of task execution is automatically postprocessed by the RunMaster using the ApplicationManager component (task outputs are shown as circles).

3. transport and networking layer.

Transport and networking layer provides the communication protocol between remote processes and handles other low-level communication aspects such as TCP connection management and server thread management. The core framework implements the Master/Worker processing logic which transparently handles the interaction between the RunMaster and WorkerAgents. It provides a *software bus* which connects components in the application and scheduling layer.

The layered architecture allows to decouple the physical networking topology from the application processing logic. This is convenient because 1) it allows to transparently replace the transport and networking layer by another implementation if needed and 2) it allows to transparently adapt the physical communication model to other topologies such as multi-tier masters. Multi-tier masters would allow the coordination of physically co-located worker nodes to be managed by a local sub-master to increase scalability. Currently in the EGEE Grid this requires setting up dedicated services (so-called VO-boxes) in the grid sites. VO-boxes are used by large VOs to orchestrate critical activities, such as data transfers in High Energy Physics. However, this option is generally not available to individual users. In the future, communication which is now routed via a central point could be short-circuited using multi-tier masters, with advantages for scalability and network traffic control.

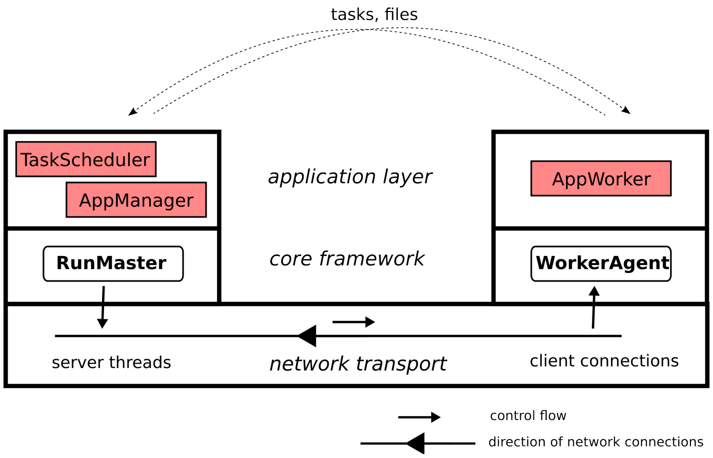

Figure 4.6: Layered architecture of Diane. Network transport layer handles communication details and is encapsulated and invisible to the application layer which defines application processing logic. Framework layer defines interaction between distributed processing elements.

4.5.2 Application and scheduling layer

Application plugins are implemented as Python classes as shown in Fig. 4.7. TaskScheduler keeps track of processed task entries (which also includes task input and output parameters) and is responsible of mapping tasks to workers and handling the task-failure policies. SimpleTaskScheduler is provided for bag-of-task applications to be used out-of-the box and it uses flat task queues for book-keeping. Alternative implementations of the TaskScheduler allow the task mapping and coordination logic to be easily plugged in. A developer of a specific TaskScheduler implementation is free to choose the most appropriate data structure to keep the task entries for the bookkeeping.

TaskScheduler is an active object (thread) which runs in the master process, alongside the RunMaster thread (core service). The RunMaster notifies the scheduler of the events such as task and worker status changes. The scheduler may schedule/unschedule tasks and remove workers by calling appropriate methods of the RunMaster object. The communication between RunMaster and TaskScheduler is fully asynchronous and implemented through thread-safe in-memory queues.

ApplicationManager handles task creation, pre- and post-processing of task input and output such as merging of output for completed tasks. Task scheduler forwards event notifications to the ApplicationManager. The separation between the application manager and the task scheduler enables to isolate scheduling aspects from application actions (such as merging outputs of successfully completed tasks).

Clear separation of concerns and asynchronous callback architecture provide a flexible and open environment. It is free of a priori constraints of approaches which rely on generalized, abstract representations of application and control-flow models which are often expressed through specialized markup languages such as GSFL [111]. Plugin

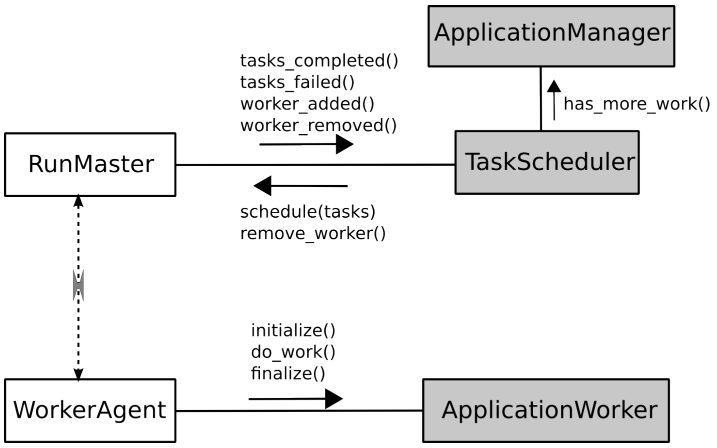

Figure 4.7: Collaboration diagram showing the interaction of RunMaster and WorkerAgent components. Plugin classes (shown in gray) allow to customize the scheduling and application-specific actions of the framework.

components may implement specific application functionality or be generalized for a subset of applications sharing similar logic. For example, a DAG scheduler allows to specify job dependencies using the DAGman [55] representation. DAG scheduler may be easily combined with application-specific ApplicationManager, or even programmatically extended to include application-specific features. This strategy allows to use at the same time the declarative and imperative styles for application modeling, scheduling and control in a flexible, extensible and efficient way.

ApplicationWorker is responsible for actual processing of the task on the worker node. The WorkerAgent process, passes the task parameters received from the RunMaster and waits for output produced by the application worker to send it back to the RunMaster. When a WorkerAgent joins or leaves the worker pool, it is initialized (or finalized) by calling appropriate methods of the ApplicationWorker.

By default ApplicationWorker module is loaded in-process by the WorkerAgent (imported as a Python module), however, a configuration option exists to load the ApplicationWorker as a separate process and use local IPC channel to communicate with the WorkerAgent. This is useful if the ApplicationWorker uses compiled extension modules which may crash the hosting process. With out-of-process option enabled, the WorkerAgent may cleanly report the application crash to the RunMaster.

4.5.3 Core framework

The core framework implements the communication logic of the Master/Worker pattern using CORBA objects. RunMaster keeps track of registered workers in a list. The worker parameter list includes the worker status (idle, busy, dead or lost), history of status update timestamps, processed task identifiers etc. The RunMaster effectively manages a pool of worker nodes – it provides mechanisms to launch new tasks on remote

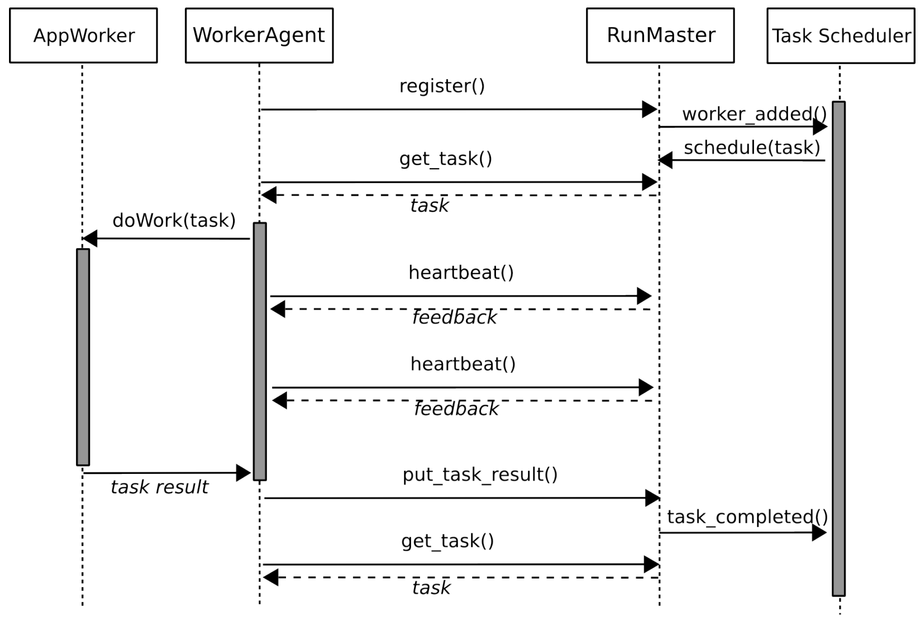

Figure 4.8: Core framework sequence diagram showing a Master/Worker interaction and unidirectional communication model with heartbeats.

resources and to report finished tasks to the application layer, as shown in Fig. 4.8.

The RunMaster detects worker node or network failures through a configurable heartbeat mechanism. WorkerAgents send periodic confirmations of their status to allow the RunMaster to detect crashed or idle workers. If a worker fails to report its status within a specified time it is marked as *lost* and it is up to the components in the application layer to implement a suitable recovery policy (typically rescheduling tasks to another WorkerAgent). Similarly, if a worker remains idle for a specific time then it may be automatically removed from the pool. The framework guarantees consistent handling of the worker pool: once a worker is removed, all further connection attempts of that worker are ignored by the master.

WorkerAgent bookkeeping entries which are used by the RunMaster may be easily extended with arbitrary attributes, for example data tags. Such application-specific information may be used by the TaskScheduler or ApplicationManager to make scheduling decisions based on the tag information derived at runtime by the worker nodes. In addition, the Core framework automatically collects benchmarking and system information about the worker nodes (hardware characteristics, operating system and environment) which may be easily accessed by the scheduler.

4.5.4 Transport and networking layer

The underlying transport and networking layer is based on omniORB with TCP/IP and is invisible to the application layer. This layer is responsible for reliable messaging between the master and the workers, scalable connection management and high-performance server thread management.

From the networking point of view the workers are clients of the master server and the communication is always unidirectional: from the workers to the master. Therefore, the connectivity requirements are minimal: outbound connectivity from the worker nodes and at least one open port for inbound connectivity (through a potential firewall) on the master node.

Bi-directional communication is based on periodic polling through heartbeats. The RunMaster effectively sends feedback to the WorkerAgents in a reply to a heartbeat as shown in Fig. 4.8. This model comes as an improvement of previously used bi-directional communication scheme in which the RunMaster could actively send messages to a WorkerAgent by reusing existing connections to the WorkerAgent. WorkerAgent was effectively acting as a server which proved to be hard to implement, susceptible to deadlocks and induced scalability limitations.

omniORB allows to easily configure the automatic shutdown of connections after a period of inactivity for incoming and outgoing connections by setting `inConScanPeriod` and `outConScanPeriod` options. The heartbeat polling rate combined with the connection shutdown timeout allows to effectively switch from connection-oriented system (where the connections remain open at all times) to connection-less system with short-lived TCP/IP connections (similarly to classic HTTP requests). For example, if average task duration is 15 seconds, it makes sense to keep the connections open all the time to avoid the overhead of frequent connection trashing. On the other hand, if average task duration is 1 hour, it may make sense to set heartbeat rate to 30 minutes and `outConScanPeriod`,`inConScanPeriod` to 1 second to immediately shutdown the connection (which would otherwise remain idle). This may be easily configured on per-run basis and it allows to control the number of concurrently open TCP/IP connections and increase scalability.

Sophisticated server-side thread management options are provided by omniORB to control efficiency and resource usage trade-offs. Concurrent calls from the same client may be multiplexed using the same connection. A separate server thread may be dedicated to each incoming connection or a thread pool of configurable size may be used. The server may also automatically switch to a thread pool mode if too many connections arrive. For example, if a worker pool is small, the requests may be efficiently handled by dedicated threads. When a worker pool grows, the requests are queued and served in a thread pool. The configuration and runtime handling of threads and connections is completely transparent to the upper layers of the system.

4.5.5 Security

Diane provides flexible authentication and authorization options. Through omniORB options the RunMaster may be configured to advertise its service using Unix domain or

TCP/IP sockets. Standard transport protocols may be easily combined with SSL and Grid Security Infrastructure (GSI) [65]. With the GSI-based security, the RunMaster creates GSI-secured listening endpoint using user's grid proxy certificate, while the Worker Agents submitted to the grid retain their copy of a proxy certificate as provided by standard middleware. This mechanism is fully compliant with recommended security practices in the grid.

Additionally, multiple network transports may be specified for the same RunMaster service. This allows the WorkerAgents to connect from a local, trusted network with different authentication and authorization rules than are applied to the WorkerAgents running in remote, untrusted locations. Rules for accepting incoming connections may also include the IPv4 address specification with subnet mask bit selection. The endpoint configuration may also be used to prioritize the networks, such that connections over a fast network may be preferred to other networks.

4.5.6 Pre-defined components

Several application-level components which encapsulate most common use-cases are available for a direct use. They include SimpleTaskScheduler, SimpleApplicationManager and Executable application handler. We provide a brief functional overview of these components to illustrate the architecture of the system and existing configuration options.

SimpleTaskScheduler provides scheduling capabilities for sets of independent tasks processed in a self-scheduling mode. The scheduler uses a flat task queue to dispatch the tasks on the first-come first-served basis. When the queue is empty, the run terminates. The scheduler may automatically reschedule failed or lost tasks. No assumptions are made about the order in which these tasks are rescheduled. A worker time limit may be set, to control the worker node time usage. A particular worker which consistently fails to execute the tasks may be automatically black-listed and removed from the pool. The detailed list of policy options may be found in online reference documentation of DIANE.

SimpleApplicationManager allows to specify application-specific splitting and merging capabilities and customize the run termination criteria. In the simplest case the `initialize()` method is used to split the initial workload by creating a static list of tasks which is then passed on to the scheduler. The `tasks_done()` method is called whenever tasks complete and it may implement on-the-fly post-processing such as output merging. The `has_more_work()` method is called periodically and it may be used to determine application-specific conditions to terminate the run. The `finalize()` method may be used for final post-processing and cleanup.

An advanced application may take advantage of dynamic task splitting at runtime. This may be achieved by deriving a specific ApplicationManager from a BaseThread class, which automatically turns the ApplicationManager into an active object with its own control thread. ApplicationManager may take advantage of parallel post-processing in a thread-pool in the cases when sequential post-processing of tasks would lead to bottlenecks.

ExecutableApplication component is provided for easy integration of legacy appli-

```
# this file is given to the RunMaster as argument
# SimpleTaskScheduler is used by default.

# tell DIANE that we are just running executables
from diane_test_applications import ExecutableApplication as application

# the run function is called when the master is started
# input.data stands for run input parameters
def run(input,config):
  d = input.data.task_defaults # a convenience shortcut

  # all tasks will share these default parameters
  d.input_files = ['hello']
  d.output_files = ['message.out']
  d.executable = 'hello'

  # here are tasks differing by arguments to the executable
  for i in range(20):
    t = input.data.newTask()
    t.args = [str(i)]
```

Figure 4.9: Listing of a run file with ExecutableApplication. A static list of tasks is created at the beginning of a run in a user-defined loop in the `run()` function.

cations. It allows to specify executable tasks with input and output files and variable executable arguments. It extends SimpleApplicationManager and may be directly used with the default scheduler. It provides resource usage monitoring on the worker nodes, including CPU and wall-clock times. It handles transfer of files of arbitrary sizes using a built-in FileTransferService, including the standard output and standard error files. The usage of ExecutableApplication is illustrated in Fig. 4.9.

4.6 The Ganga resource access API and user interface

Ganga is a user-centric tool that allows easy interaction with heterogeneous computational environments, configuration of the applications and coherent organization of jobs [136]. The implementation uses an object-oriented design in Python. Ganga provides uniform access to remote resources and high-level user interface for application configuration and job management. In the context of Diane, Ganga is used to submit and manage WorkerAgent jobs. It may also be used as a simplified bookkeeping environment for Diane tasks and assist user in application configuration.

Ganga provides a simple but flexible programming interface that can be used either interactively at the Python or IPython [151] prompt, through a Graphical User Interface (GUI) or programmatically in scripts. This reflects the different working styles in different user communities and addresses various usage scenarios such as using the GUI for training new users, the command line to exploit advanced use-cases, and scripting for automation of repetitive tasks.

The concept of a *job* component is essential as it contains the full description of a computational task, including: the code to execute; input data for processing; data produced by the application; the specification of the required processing environment; post-processing tasks; and metadata for bookkeeping. Ganga keeps track of all jobs and their status through a repository that archives all information between independent Ganga sessions. It is possible to switch between executing a job on a local computer and executing on the Grid by changing a single parameter of a job object. This simplifies the progression from rapid prototyping on a local computer and small-scale tests on a local batch system, to the analysis of a large dataset using Grid resources.

It is possible to make Ganga available to a user community with a high level of customization. For example, a domain expert can implement a custom application class describing the specific computational task. The class will encapsulate all low-level setup of the application, which is always the same, and only expose a few parameters for configuration of a particular task. The plugin system provided in Ganga means that this expert customization will be integrated seamlessly with the core of Ganga at runtime, and can be used by an end user to process tasks in a way that requires little knowledge about the interfaces of Grid or batch systems. Issues such as differences in data access between jobs executing locally and on the Grid are similarly hidden.

Ganga is a user- and application-oriented layer above existing job submission and management technologies, in Globus[4], Condor [170], Unicore [168] or gLite [113]. Rather than replacing the existing technologies, Ganga allows them to be used interchangeably, using a common interface as the interoperability layer.

Ganga may be used as a job management system integrated into a larger system. In the case of User-level Overlay Ganga acts as an API for job submission and control.

4.6.1 Architecture and functionality

Fig. 4.10 shows the architecture of Ganga where the three user interfaces are built on top of the Ganga Public Interface (GPI) which in turn provides access to the Ganga core implementation.

A job in Ganga is constructed from a set of components. All jobs are required to have an application component and a backend component, which define respectively the software to be run and the processing system to be used. Many jobs also have input and output dataset components, specifying data to be read and produced. Computationally intensive jobs may have a splitter component which provides a mechanism for dividing into independent subjobs, and a merger component, which allows the aggregation of subjob outputs. The overall component structure of a job is illustrated in Fig. 4.11.

By default, the GPI exposes a simplified, top-level view suitable for most users in their everyday work, but at the same time allows the details of underlying systems to be exposed if needed. Ganga monitors the evolution of submitted jobs and categorizes them into the simplified states *submitted*, *running*, *completed*, *failed* or *killed*.

All job objects are stored in a job repository database, and the input and output

[4]The Globus Alliance, `http://www.globus.org`

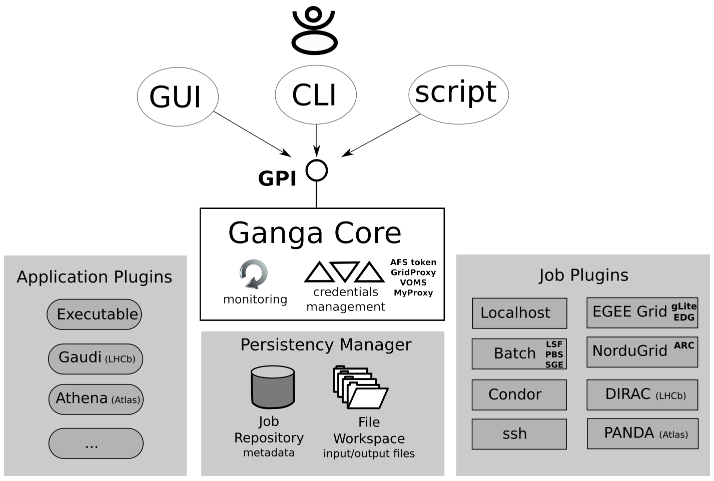

Figure 4.10: Architecture of Ganga. The user interacts with the Ganga Public Interface (GPI) via the Graphical User Interface (GUI), the Command-Line Interface in Python (CLI), or scripts. Plugins are provided for different application types and backends. All jobs are stored in the repository.

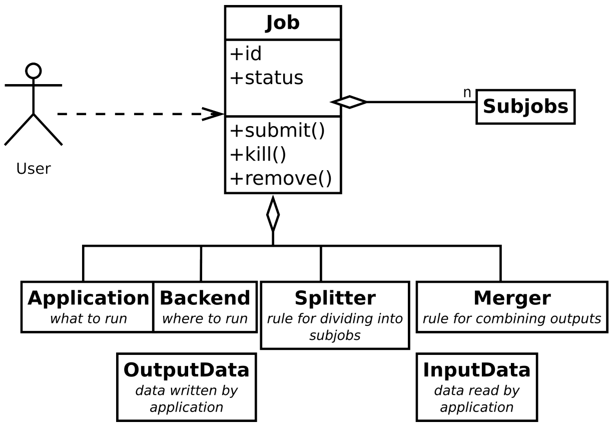

Figure 4.11: A set of classes in Ganga can be combined to form a complete job. The application to run and the backend where it will run are mandatory while all other components are optional.

files associated with the jobs are stored in a file workspace. Both the repository and the workspace may be in a local filesystem or on a remote server.

A large computational task may be split into a number of subjobs automatically according to user-defined criteria and the output merged at a later stage. Each subjob will execute on its own and the merging of the output will take place when all have finalized. The submission of subjobs is automatically optimized if the backend component supports bulk job submission. For example, when submitting to the gLite workload management system [113] the job collection mechanism is used transparently to the user.

Job splitting functionality provides a flat list of subjobs suitable for parallel processing of fully independent workloads. However, certain backends allow users to make use of more-sophisticated pluralization schemes, for example the MPI. In this case, Ganga may be used to manage collections of subjobs corresponding to MPI processes. Similarly the Diane RunMaster may be specified as a Ganga backend with the server-side splitting and merging capabilities.

Ganga has built-in support for handling user credentials, including classic Grid proxies, proxies with extensions for VOMS (Virtual Organization Management Service) [11], and Kerberos [144] tokens for access to an Andrew filesystem (AFS) [132]. A user may renew and destroy the credentials directly using the GPI. Ganga gives an early warning to a user if the credentials are about to expire. The minimum credential validity and other aspects of the credential management are fully configurable.

Ganga supports multiple security models. For local and batch backends, the authentication and authorization of the users is based on the local security infrastructure including user name and network authentication protocols such as Kerberos. GSI provides for security across organizational boundaries for the Grid backends. Different security models are encapsulated in pluggable components, which may be simultaneously used in the same Ganga session.

Application components

The application component describes the type of computational task to be performed. It allows the characteristics and settings of some piece of software to be defined, and provides methods specifying actions to be taken before and after a job is processed. The pre-processing (configuration) step typically involves examination of the application attributes, and may derive secondary information. For example, intermediate configuration files for the application may be created automatically. The post-processing step can be useful for validation tasks such as determining the validity of the application output.

The simplest application component (`Executable`) has three attributes:

`exe` : the path to an executable binary or script;

`args`: a list of arguments to be passed to the executable;

`env` : a dictionary of environment variables and the values they should be assigned before the executable is run.

The configuration method carries out integrity checks – for example ensuring that a value has been assigned to the `exe` property.

Backend components

A backend component contains parameters describing the behavior of a processing system. The list of parameters can vary significantly from one system to another, but can include, for example, a queue name, a list of requested sites, the minimum memory needed and the processing time required. In addition, some parameters hold information that the system reports back to the user, for example the system-specific job identifier and status, and the machine where a job executed.

A backend component provides methods for submitting jobs, and for cancelling jobs after submission. It also provides methods for updating information on job status, for retrieving output of completed jobs and for examining files produced while a job is running.

Backend components have been implemented for a range of widely used processing systems, including: local host, batch systems (Portable Batch System (PBS) [88], Load Sharing Facility (LSF) [163], Sun Grid Engine (SGE) [72], and Condor [170]), and Grid systems, for example based on gLite [113], ARC [58] and OSG [155]. Remote backend component allows jobs to be launched directly on remote machines using ssh. DIANE backend enables the RunMaster server to be started up directly from GANGA.

As an example, the batch backend component defines a single property that may be set by the user:

`queue` : name of queue to which job should be submitted, queue being used if this left unspecified,

and defines three attributes for storing system information:

`id` : job identifier;

`status` : status as reported by batch system;

`actualqueue`: name of queue to which job has been submitted.

In addition, a remote-backend component allows a job defined in a GANGA session running on one machine to be submitted to a processing system known to a remote machine to which the user has access. For example, a user who has accounts on two clusters may submit jobs to the batch system of each from a single machine.

Dataset components

Dataset components generally define attributes that uniquely identify a particular collection of data, and provide methods for obtaining information about it, for example its location and size. The details of how data collections are described can vary significantly from one problem domain to another, and the only generic dataset component in GANGA represents a null (empty) dataset. Other dataset components are specialized for use with a particular application, and so are discussed later.

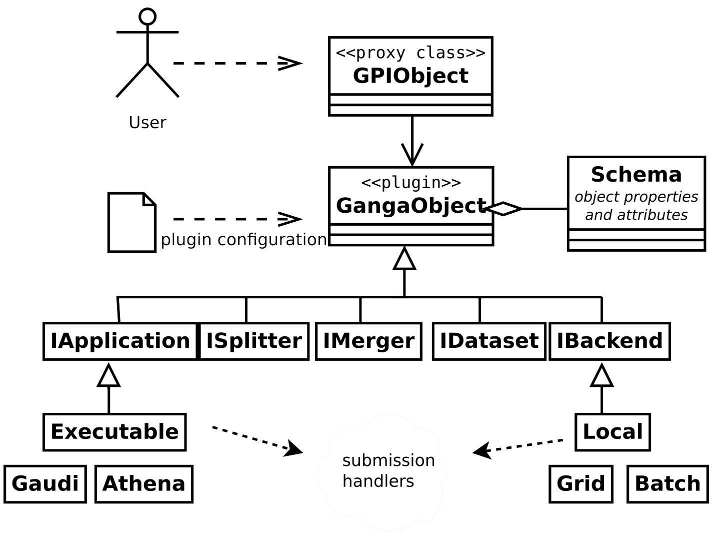

Figure 4.12: A component class implements one of the abstract interfaces corresponding to the different parts of a job. Runtime dependencies between application and backend classes are provide by submission handlers.

A strict distinction is made between the datasets and the sandbox (job) files. The former are the files or databases which are stored externally. The sandbox consists of files which are transferred from the user's filesystem together with the job. The sandbox mechanism is designed to handle small files (typically up to 10 MB) while the datasets may be arbitrarily large.

4.6.2 Implementation

Here we provide details of the actual implementation of some of the most important parts of Ganga.

Components

Job components are implemented as plugin classes, imported by Ganga at start-up if enabled in a user configuration file. This means that users only see the components relevant to their specific area of work. Plugins developed and maintained by the Ganga team are included in the main Ganga distribution and are upgraded automatically when a user installs a newer Ganga version. Currently, the list includes around 15 generic plugins and around 20 plugins specific to HEP data analysis. Plugins specific to other user communities need to be installed separately but could easily be integrated into the main Ganga distribution.

Plugin development is simplified by having a set of internal interfaces and a mechanism for generating proxy classes [71]. Component classes inherit from an interface class, as seen in Fig. 4.12. Each plugin class defines a schema, which describes the plugin attributes, specifying type (read-only, read-write, internal), visibility, associated user-convenience filters and syntax shortcuts.

The user does not interact with the plugin class directly but rather with an automatically generated proxy class, visible in the GPI. The proxy class only includes attributes defined as visible in the schema and methods selected for export in the plugin class. This separation of the plugin and proxy levels is very flexible. At the GPI level, the plugin implementation details are not visible; all proxy classes follow the same design logic (for example, copy-by-value); persistence is automatic, session-level locking is transparent. In this way the low-level, internal API is separated from the user-level GPI.

The framework does not force developers to support all combinations of applications and backends, but only the ones that are meaningful or interesting. To manage this, the concept of a *submission handler* is introduced. The submission handler is a connector between the application and backend components. At submission time, it translates the internal representation of the application into a representation accepted by a specific backend. This strategy allows integration of inherently different backends and applications without forcing a lowest-common-denominator interface.

Most of the plugins interact with the underlying backends using shell commands. This down-to-earth approach is particularly useful for encapsulating the environments of different subsystems and avoiding environment clashes. In verbose mode, Ganga prints each command executed so that a user may reproduce the commands externally if needed. Higher-level abstractions such as JSDL [126] or OGSA-BES [16] are not currently used, but specific backends that support these standards could readily be added.

Job persistence

The *job repository* provides job persistence in a simple database, so that any subsequent Ganga session has access to all previously defined jobs. Once a job is defined in a Ganga session it is automatically saved in the database. The repository provides a bookkeeping system that can be used to select particular jobs according to job metadata. The metadata includes such parameters as job name, type of application, type of submission backend, and job status. It can readily be extended as required.

Ganga supports both a local and a remote repository. In the case of the former, the database is stored in the local file system, providing a standalone solution. In the case of the latter, the client accesses an AMGA [104] metadata server. The remote server supports secure connections with user authentication and authorization based on Grid certificates. Performance tests of both the local and remote repositories show good scalability for up to 10 thousand jobs per user, with the average time of individual job creation being about 0.2 seconds. There is scope for further optimization in this area by taking advantage of bulk operations and job loading on demand.

The job repository also includes a mechanism to support schema migration, allowing for evolution in the schema of plugin components.

Input and output files

Ganga stores job input and output files in a *job workspace*. The current implementation uses the local file system, and has a simple interface that allows transparent access to job files within the Ganga framework. These files are stored for each job in a separate directory, with sub-directories for input and output and for each subjob.

Users may access the job files directly in the file-system or using Ganga commands such as `job.peek()`. Internally, Ganga handles the input and output files using a simple abstraction layer which allows trivial integration of additional workspace implementations. Tests with a prototype using a WebDav [181] server have shown that all workspace data related to a job can be accessed from different locations. In this case, a workspace cache remains available on the local file system.

The combination of a remote workspace and a remote job repository effectively creates a roaming profile, where the same Ganga session can be accessed at multiple locations, similar to the situation for accessing e-mail messages on an IMAP [1] server.

Monitoring

Ganga automatically keeps track of changes in job status, using a monitoring procedure designed to cope with varying backend response times and load capabilities. Each backend is polled in a different thread taken from a pool, and there is an efficient mechanism to avoid deadlocks from backends that respond slowly. The poll rate may be set separately for each backend.

The monitoring sub-system also keeps track of the remaining validity of authentication credentials, such as Grid proxies and Kerberos tokens. The user is notified that renewal is required, and if no action is taken then Ganga is placed in a state where operations requiring valid credentials are disabled.

4.7 Heuristic resource selection

4.7.1 Submitters

Worker agents are managed as Ganga `Executable` job wrappers. The job wrapper downloads and installs Diane on-the-fly in the worker node and spawns a Worker-Agent process which connects to the RunMaster. A `Submitter` helper class handles preparation and submission of worker agent wrappers, including low-level details such as inclusion of the master server address, which is encoded as CORBA IOR, into the worker wrapper input sandbox. The helper makes the system not only easy to use but also easy to extend with customized submitter scripts. Fig. 4.13 shows a complete implementation of a Ganga submitter script for the LSF batch system. A simple substitution of a backend object allows to reuse the same script to transparently submit worker agents to the EGEE Grid or any other infrastructure supported by Ganga, including explicitly selected worker nodes in a local computing center via ssh. Shell commands are available to a user for easy submission of WorkerAgents via a selected Ganga backend.

```
#!/usr/bin/env ganga
#-*-python-*-

from diane.submitters import Submitter

submitter = Submitter()
submitter.download=False
submitter.parser.add_option("--delay",...)
submitter.parser.add_option("--queue",...)
submitter.initialize()

import time

for i in range(submitter.options.N_WORKERS):
    j = Ganga.GPI.Job()
    j.backend=Ganga.GPI.LSF(queue=submitter.options.queue)
    submitter.submit_worker(j)
    if submitter.options.delay:
        time.sleep(submitter.options.delay)
```

Figure 4.13: LSF submitter helper (`LSFSubmitter.py`). The actual worker agent submission is handled by Ganga, according to the specified backend.

4.7.2 Simple Agent Factory

In typical cases of Capacity Computing, the lifetime of individual worker agent is much shorter than the makespan of a run. As the number of available resources in the pool decreases with time the submission of new worker agents is needed in the course of the run to maintain the worker pool.

Workers may be easily submitted by hand by a user, but that may require frequent manual operation and may be time consuming and inefficient. Agent Factory is a component which automates the submission process.

The aim of the Agent Factory is to maintain the number of active workers as high as possible but not greater than a specified number. As an example, in case of a sudden drop in the number of workers one would like to react by submitting new workers to replenish the worker pool. On the other hand the submission of new worker agents should be kept under control and on par with the number of available resources in the grid at a given moment. In particular, if not enough free resources are available, then an excessive, continuous submission would have little effect on the speedup of the system but could lead to overloading of grid services.

Simple Agent Factory, shown in Fig. 4.14, periodically checks the size of the worker pool reported by the RunMaster and fills up the pool by submitting more workers to the grid. User defined job resource requirements, such as specified in the GLUE schema [17], may be used to guide the resource selection which is delegated to the grid WMS.

This very simple solution may, however, lead to suboptimal resource selection as it does not take into account runtime failures in the worker nodes. Another common problem is that users are often not able to efficiently define detailed job resource requirements due to abundance of configuration parameters and heterogeneity of the grid environ-

```
# worker agent submitter helper with the user-defined
# resource requirements. For historic reasons
# EGEE Grid is aliased to LCG.
submitter = LCGSubmitter(requirements)

# Ganga job repository (list of worker agents in the pool)
jobs = Ganga.GPI.jobs

while 1:
   # current size of the worker pool
   # obtained via a RPC query to the RunMaster
   current_pool_size=runmaster.get_worker_pool_size()

   # submit the workers to fill up the pool size
   # choice of CE is left to Grid WMS, based on resource requirements
   for i in range(N_WORKERS-current_pool_size):
     submitter.submit_worker(Ganga.GPI.Job())
```

Figure 4.14: Simple Agent Factory algorithm implemented in Ganga. Simple Agent Factory uses Ganga GPI interface to manipulate worker agent jobs and Diane RPC interface to query the run master. `N_WORKERS` is a size of the worker pool set by the user.

ment. Specific requirements may include hardware and software parameters, resource allocation policy constraints and software dependencies which may also be site-specific and thus impossible to define globally.

4.7.3 Heuristic Agent Factory

Heuristic Agent Factory (HAF) component was designed to optimize the resource selection, based on system feedback using recent performance data. This is accomplished through a resource selection algorithm in which the grid sites are ranked based on their reliability and performance over time. The algorithm is designed to cope with the observed dynamics of the grid where the number of available computing resources is variable in time. Typical cases include sites entering downtime periods and stopping accepting jobs, or sites going back into production after configuration fixes, hardware upgrades etc. The selection algorithm gives more weight to more recent performance data and eventually "forgets" old data. It allows to increase the submission success rate and to maintain the number of worker agents on a predefined level.

Selection algorithm

In EGEE Grid, distributed computational clusters or batch farm are represented by *Computing Elements* (CEs). CE is the smallest management unit for the resource selection algorithm. HAF submits jobs to the CEs via the *Workload Management System* (WMS) which is used as a grid gateway.

The core of HAF is a non-deterministic selection procedure based on a fitness algorithm commonly found in genetic algorithms/evolutionary strategies [131]. When a new worker agent is submitted to the grid, a CE is chosen randomly with probability proportional to its fitness. CEs are selected by the HAF and the WMS simply forwards the job submission requests. The outline of the algorithm is presented in Fig. 4.15 and it shows the actual PYTHON code from the HAF implementation.

For a Computing Element with n total jobs, r jobs currently running and c jobs completed without errors, the $fitness(\mathrm{CE})$ is defined as

$$fitness(CE) = \frac{r + c}{n} \tag{4.1}$$

The fitness value lies in the [0..1] interval. The $fitness = 1$ represents a reliable CE with all workers either running or finished cleanly. If all workers are queuing in a CE or if a CE is unable to correctly execute any jobs for the application then $fitness = 0$.

A *generic CE slot* corresponds to a random CE selected by the WMS and is used for discovery and adaptive ranking of CEs. The $fitness$ of the generic slot is always 1. At bootstrap the list of known CEs is empty and all jobs are submitted via the generic slot. As the list of known CEs grows the HAF keeps on using the generic slot to submit a small fraction of jobs (`MAX_PENDING` parameter) random sites to detect the availability of new resources or an improvement in the performance of CEs with low fitness.

For each CE the probability to be selected by HAF is

$$P(\mathrm{CE}) = \frac{fitness(\mathrm{CE})}{1 + \sum fitness(\mathrm{CE}_i)} \tag{4.2}$$

The denominator is the total fitness of the population of known CEs, where 1 represents the generic CE slot.

Resources discovery and balancing

Initially the list of known, available CEs is empty. While it is possible to directly query the grid information services to obtain a list of all available Computing Elements, this is of little help since the selection algorithm needs historical data to calculate the fitness. However, this problem does not exist in algorithms which uses the generic slot where the WMS makes a few initial CE choices.

At the beginning, the generic slot is the only resource available and the only candidate for selection. Computing Elements are chosen automatically by the WMS and HAF stores this information. Eventually, with enough data collected, it takes over the decision process. As the number of discovered Computing Elements grows, the likelihood of the generic slot being selected for submission decreases. However, the fitness of the generic slot is permanently fixed at 1, and thus the probability of selecting it never reaches 0. This is important to maintain balance: when known Computing Elements perform well, few or no workers are sent to the grid via the generic slot and new workers are distributed to known Computing Elements. When known Computing Elements start under-performing and the fitness of the population decreases, we can expect the generic slot to be chosen more often, giving WMS a chance to find new resources. This way, we avoid overloading of Computing Elements with excessive resource requests.

```
# worker agent submitter helper
submitter = LCGSubmitter()

# Ganga job repository (list of worker agents in the pool)
jobs = Ganga.GPI.jobs

# return jobs which belong to a given CE only (note: by lexical closure)
def filter_CE(job_list):
  return [j for j in job_list if j.backend_actualCE == CE]

while 1:
    # current size of the worker pool
    # obtained via a RPC query to the RunMaster
    current_pool_size=runmaster.get_worker_pool_size()

    # build the set of all known CEs and prune old jobs
    knownCEs = set()

    for j in jobs:
      if too_old(j):
       j.remove()
      else:
       knownCEs.add(j.backend.actualCE)

    # calculate fitness
    for CE in knownCEs:
      n = len(filter_CE(jobs))
      c = len(filter_CE(jobs.select(status='completed')))
      r = len(filter_CE(jobs.select(status='running')))
      fitness[CE] = float(c + r) / n

    total_fitness = sum(fitness)

    #allow a small oversubmission of workers (MAX_PENDING)
    total_pending = jobs.select(status='submitted')

    # submit workers if pool size smaller than N_WORKERS
    while current_pool_size<N_WORKERS and len(total_pending) < MAX_PENDING:

      #select the CE using fitness-proportional method
      r = random.uniform(0, total_fitness + 1)
      for i in range(len(fitness)):
        r -= fitness[i]
        if r <= 0:
          CE = knownCEs[i]
          break
      submitter.submit_worker(Ganga.GPI.Job(), CE)
```

Figure 4.15: Heuristic Agent Factory (HAF) algorithm implemented in Ganga. HAF uses Ganga GPI interface to manipulate worker agent jobs and Diane RPC interface to query the run master. `N_WORKERS` is a size of the worker pool set by the user. `MAX_PENDING` controls the job submission redundancy.

Discussion

The added value of the HAF is that it ranks available resources as a function of the current performance for the *specific application*. The HAF is an efficient way to automatize resource provisioning without overloading the system with unnecessary submissions, hence to maximize the overall duty cycle of user's application.

It is important to note that functionality of the HAF and that of WMS are complementary. WMS provides global workload balancing between Computing Elements while the role of the HAF is the selection of the most compatible resources in a context of a particular application.

An arbitrary subset of job resource requirements may be easily defined by the user for the generic grid slot of the HAF to constrain the set of Computing Elements for the resource selection algorithm. HAF works on a higher level than the WMS itself and thus all WMS configuration parameters and job requirements are automatically taken into account. However, if job requirements are absent or poorly defined the HAF is still able to dynamically adapt in a non-parametric way to operating within the constraints of the application.

HAF algorithm is also application-agnostic as it does not require any application-specific knowledge and relies solely on the worker agent job status information provided by the underlying grid system.

The evaluation of HAF is provided in the context of a specific application and is presented within Capacity Computing Case Study in Chapter 6.

4.8 Adaptive workload balancing

In Chapter 3 we showed that the late-binding U_0 model defines the maximal theoretical speedup with ideal load balancing, which is also a limit for the U_1 discrete model for small task sizes. We also concluded that the runtime overheads should not exceed 10% of the task execution time. This constraints the task splitting granularity for applications where communication overheads become large. Additionally, in grid environment, worker nodes may be shared among multiple jobs. This may result in varying performance characteristics (for example the CPU load) of the processors and, for longer tasks and simple self-scheduling strategy, may lead to suboptimal schedules.

A number of semi-automatic load balancing methods have been developed (e.g. diffusion self-balancing mechanism, genetic networks load regulation, simulated annealing technique, bidding approaches, multi-parameter optimization, numerous heuristics, etc.), but most of them suffer one or another serious limitation, most noticeably the lack of flexibility, high overheads, or inability to take into consideration specific features of the application. To overcome these limitations we have developed a hybrid approach where the balancing decision is taken in interaction of the application with the execution environment. In this approach the adaptive workload balancing algorithm (AWLB), developed in [105], is applied at the task scheduling level. The resource pool is maintained by a Hybrid Agent Factory, which uses the information provided by the scheduler and application benchmarks, to rank and select the resources.

4.8.1 The AWLB algorithm

In self-scheduling all the workload is divided into tasks of equal size. As soon as a worker becomes available, it is assigned the next task from the task queue. In AWLB, the size of the task assigned to each worker is calculated by the heuristic algorithm, using the resource and application characteristics. Applying dynamic splitting, a scheduler may dynamically choose the task size such that the sum of task execution time and communication overheads is equal for all tasks executed on all processors.

The AWLB provides an optimal distribution of the divisible workload between participating processors according to the computing environment characteristics and the application requirements. The suitability of resources is determined by the application requirements; for traditional parallel computing applications considered here as a test case, it depends on the processing power and network connectivity correlated with the application communication to computation ratio. Thus the main parameters that define a parallel application performance are:

- The application parameter $f_c = N_{comm}/N_{calc}$, where N_{comm} is the total amount of application communications, i.e. data to be exchanged (measured in bit) and N_{calc} is the total amount of computations to be performed (measured in Flop);
- The resource parameters $\mu_i = p_i/n_i$, where p_i is the available performance of the i_{th} processor (measured in Flop/s) and n_i is the network bandwidth to this node (measured in bit/s).

The AWLB algorithm is based on the benchmarking of the available resources capacity, defined as a set of individual resource parameters $\mu = \mu_i$, and experimental estimation of the application parameter f_c. The value of the application parameter f_c is determined by running through the space of possible f_c and finding the value f_c^* which provides minimal runtime of the application on this set of resources. The algorithm is described in more detail in [106, 108].

4.8.2 Hybrid approach to workload balancing

The application may comprise heterogeneous tasks executed at different times with different performance characteristics so the total application performance requirements (f_c) may vary at runtime. Similarly the capacity of grid resources (μ) may vary with time due to inherent grid dynamics. The scheduler may respond to changing application or resource conditions and more suitable resource set may be selected to execute the application at a given time. This may happen at the individual task execution boundary or at other natural boundaries specific to the application model (for example at each iteration for the iterative simulations). This is a distinctive feature, in contrast to the static parallel programs where resources are allocated once and fixed during the execution unless special migration libraries are used such as the Dynamite [95].

We illustrate the hybrid workload balancing approach for an iterative computation pattern where each iteration consists of a set of independent tasks and output of one iteration is used as an input for the next iteration. This pattern is typical for many

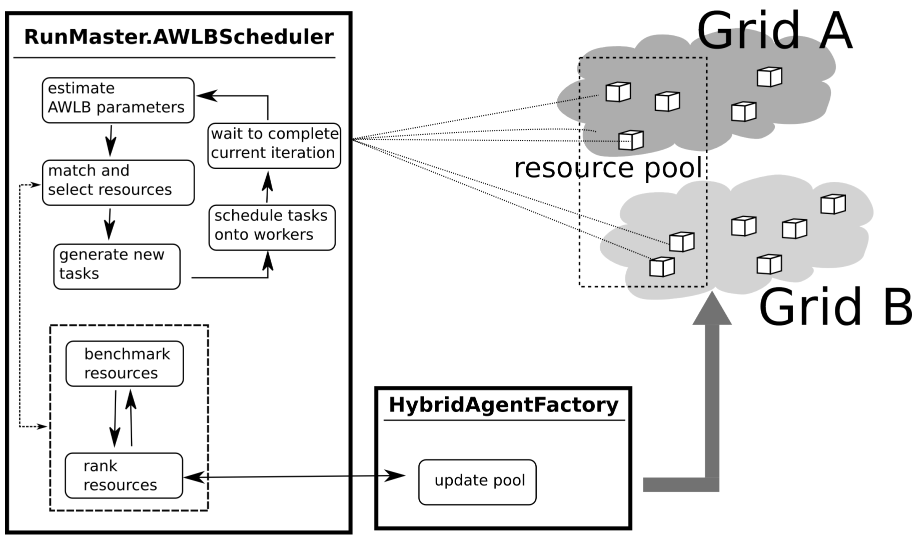

Figure 4.16: Iterative execution of parallel application in the integrated AWLB in User-level Overlay with dynamic resource pool.

time-based optimization algorithms such as SWAT-CUP [5] used for calibration and uncertainty analysis of distributed watershed models in Earth Science applications.

Hybrid workload balancing approach is shown in Fig. 4.16. The application consists of a set of parallel tasks that process the workload scheduled by the RunMaster. The RunMaster collects the information about the available resources and monitors the application responses. At each iteration of the application algorithm the distribution of the workload is re-evaluated on a updated set of resources, and the AWLB parameters are re-estimated. The scheduler uses AWLB benchmarks to rank the resources and generates tasks which duration is proportional to the capacity μ of the resources to minimize the execution time of the iteration. If more resources are available in the pool than required by the optimal schedule only the best subset is selected and used. If runtime conditions, such as CPU load or network bandwidth, change on the worker node, the resource rank in the subsequent iteration is modified.

The selection of suitable resources from available worker pool is performed by the AWLB scheduler. The same information may be used to guide the process of updating the pool which is performed asynchronously by the Agent Factory. For example, the Heuristic Agent Factory described in previous section may be extended such that a *Hybrid* Agent Factory may use the resource ranking information provided by the AWLB scheduler to complement the fitness-proportional ranking of computing elements. The fitness of each computing element may be first determined by the plain heuristic algorithm described in the previous section and then be linearly scaled by the AWLB rank to select the most efficient subset of CEs. This approach is most effective when a benchmark of a single worker node is representative for other nodes in the same farm. This is to a good approximation true for CEs which consist of homogeneous nodes.

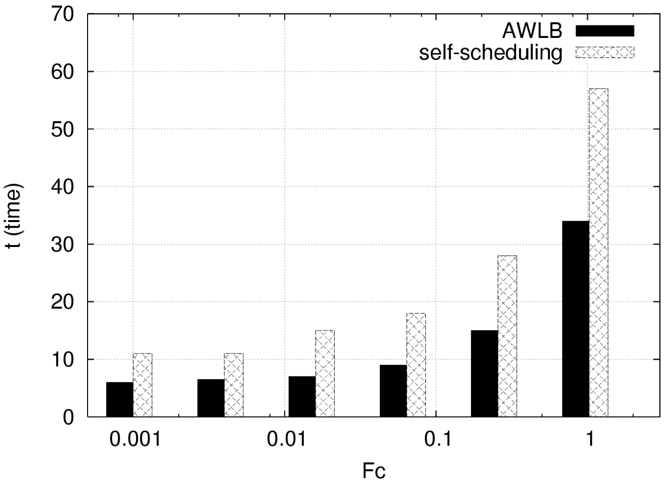

Figure 4.17: Execution time with self-scheduling and AWLB load balancing for six applications characterized by different communication/computation ratio Fc.

4.8.3 Experimental results

The experimental results were achieved for a model application with a synthesized workload and tunable parameter f_c [107], where the task input and output consisted of sequences of random bytes, and the task execution consisted of loops performing a set of standard floating point and integer operations to generate the CPU load. The number of loop cycles was used to control the N_{calc} parameter, whereas the length of input and output sequences was used to control the N_{comm} parameter. The f_c ratio was chosen correspondingly to typical values observed in computing-intensive calculations of Feynman-loops in theoretical physics. A pool of EGEE worker nodes was fixed in each test run.

Fig. 4.17 demonstrates the results for six application workloads, characterized by different communication/computation ratio Fc. In these experiments the computational load (N_{calc}) was kept constant on a fixed set of 16 processors, while the amount of data transferred between the master and the workers (N_{comm}) was varied. The execution time with the AWLB algorithm was up to 2 times lower than self-scheduling for all types of simulations.

Thorough testing of different applications on various sets of resources showed a strong influence of the level of resource heterogeneity on the results achieved. A series of targeted experiments was performed varying the resource heterogeneity in the processor power and the network links bandwidth to calculate the load-balancing speedup θ defined as

$$\theta = \frac{T_{non-balanced}}{T_{balanced}} \tag{4.3}$$

where $T_{non-balanced}$ is the execution time of the parallel application without the load-

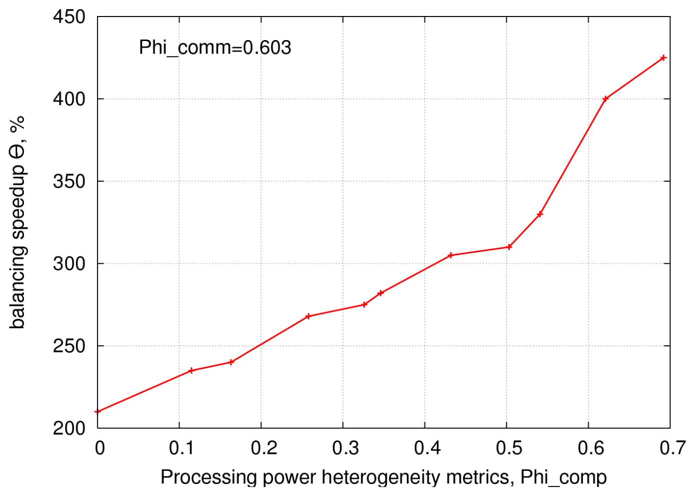

Figure 4.18: Dependency of the load balancing speedup on resource heterogeneity metrics Φ.

balancing, and $T_{balanced}$ is the execution time using load-balancing on the same set of resources. As a sample of these tests, Fig. 4.18 shows the dependency of the load balancing speedup on the processing power heterogeneity metrics Φ, defined as a standard deviation of a set of normalized parameters μ

$$\Phi = \sqrt{\frac{1}{N-1} \sum 1 - \frac{\mu}{\mu_{avg}}}. \tag{4.4}$$

As it may be seen, the speedup grows super-linearly with the heterogeneity level, thus indicating that this approach is especially beneficial for strongly heterogeneous resources, such as grids.

4.9 Summary

We outlined the key architectural principles and functional areas of the User-level Overlay, which are essential to tackle the main challenges of using grids and other distributed environments for processing of scientific tasks. While late binding is the key to mastering of the inherent dynamics in large computing grids, its efficiency may be further improved with strategies such as heuristic resource selection and adaptive workload balancing. This is more easily achieved in an architecture which decouples resource selection from scheduling.

We presented the detailed design of essential software components, and the key features of the implementation in an attempt to create a blueprint with software design and engineering guidelines for building support systems for processing of distributed tasks in a general context. One of the key aspects is the development of the User-level Overlay as a *hosting environment* for application components, and for scheduling and resource selection algorithms. This enables incremental adaptability to increasingly complex requirements.

The key features of the User-level Overlay strategy proposed in this Chapter include:

- late-binding task processing model,
- ability to interface to a wide range of distributed systems,
- ability to extend and customize the system to cover application-specific scheduling and processing patterns,
- ease of use and lightweight deployment in the user space.

The User-level Overlay concept is universal and may be used above a variety of distributed infrastructures. Software plugins to more than 10 different computational backends have been so far implemented, what provides a good indication of practical importance of this approach. An essential question at this point is, whether the ideas and concepts used to develop the User-level Overlay may stand up to expectations with real-life applications. This is considered in the next Chapter.