Scalable Computing: Practice and Experience
Volume 10, Number 4, pp. 413–418. http://www.scpe.org
ISSN 1895-1767

MULTI-APPLICATION BAG OF JOBS FOR INTERACTIVE AND ON-DEMAND COMPUTING

BRANKO MAROVIĆ, MILAN POTOČNIK, AND BRANISLAV ČUKANOVIĆ*

Abstract. A generic infrastructural grid service for submission, management, and access to computing resource is described. It is suitable for interactive applications and on-demand computing with frequent usage of short jobs. The improved access to computing resources is achieved through allocation and maintenance of a pool of ready jobs, the size of which is dynamically adjusted to demand produced by applications. A set of APIs facilitates communication between the clients and worker jobs through a simple programming model.

Key words: cluster and grid computing, distributed computing, pilot job infrastructures, interactive applications, programming API

1. Introduction.

The grid infrastructures provides access to enormous computing resources to its users. There are many grid sites dispersed throughout the world and they are grouped into several more or less independent infrastructures. Obtaining these resources can be difficult, especially for users that are not experts in the usage of grid, but are rather scientists or users of specific applications running on the grid. Various grid middlewares employ different operational schemes, authorisation policies and access interfaces. Aside from the complexity of the grid infrastructures, certain limitations are also present, notably the need for users to wait, sometimes for a significant time, until their requests for computing resources (also called job submissions) are processed and the lack of good support for interactive applications.

The use of pilot jobs [1] is gaining increasing popularity among various groups of grid users. With pilot-based infrastructures, users submit their jobs to a centralized queue or repository. These jobs are handled by grid computing resources executing asynchronously started pilot jobs. Pilot jobs communicate with a pilot aggregator, which allocates user jobs from the repository. Once started, the jobs get the actual work to be performed from the aggregator. This late binding of user jobs and resources greatly improves the user experience, as it hides broken grid resources and provides more accurate information about available resources.

On the other hand, there are many implementations of user interaction with jobs running on the grid, most of which provide communication between the user and job through text terminal-like user interface. However, none of them addresses one of key problems related to the interactive jobs—the startup time for grid jobs that is often unacceptably long for interactive use.

Work Binder is a generic service developed in Java programming language whose main purpose is to quickly allocate new jobs for grid users. Through its simple API, it also has support for interactivity between the end user and the job being run on the remote server located on the grid. With gLite middleware [2], this server is called a worker node (WN) and it is one computing node belonging to the computing element (CE), which is usually a cluster administered by some grid site. However, this service can be easily adapted to any other infrastructure that allow applications to submit code to computing resources.

Work Binder [3] is aimed to be used with three specific types of grid jobs:

- Interactive jobs—jobs executing on a remote worker node that require communication (interaction) with the user that started them (in gLite support for these types of jobs is very poor).
- Jobs with relatively short execution time that have a good chance of being resubmitted or participate in a workflow.
- Jobs with critical demand for startup time.

All of these job types have one thing in common, the need for short startup time. Work Binder is designed in such a way that it enables almost instant allocation of jobs to the users. In effect, Work Binder acts as a mediator between client and server components of an arbitrary application in the grid environment.

2. Design of work binder.

Work Binder consists of software components distributed in three tiers, at the client, worker, and central service, as shown in Figure 2.1:

- Various application-specific client programs interact with end user and invoke remote processing. They request jobs from the Work Binder service and communicates with obtained worker processes.

*University of Belgrade Computer Centre

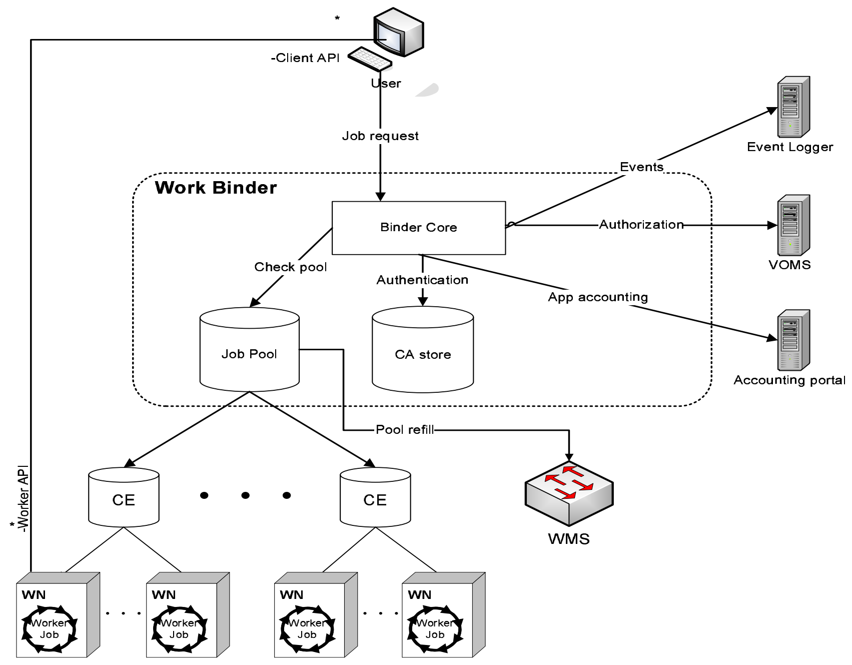

Fig. 2.1. *Components of Work Binder environment*

- Application-specific programs perform computation on grid worker nodes. Each program execution is called worker, and is being run within a grid job. It starts the actual application code (as a class or separate external program) after pairing with some client.
- Work Binder service is a mediator that maintains the pool of free workers and matchmakes them with clients. It is capable of simultaneous handling of several applications.

Key features that distinguish Work Binder, giving it a significant advantage over other related pilot infrastructure solutions [4] [5], include:

- Java API which allows quick & simple way to integrate new applications into the Binder & grid environment;
- Almost instant allocation of new jobs for users.
- Support for any number of different applications which will share the same pool of jobs;
- Adaptive dynamic allocation of new jobs;
- Work Binder hides the complexity of grid infrastructure from the user;
- Possibility to incorporate application-specific intercepting handlers into the mediator.

2.1. Submission of New Jobs. Work Binder submits worker jobs to the grid, maintains a pool of ready jobs, and mediates between clients and workers. It pairs clients and worker jobs from which it has received connections. A small dispatcher program started by the Worker Job then starts an application-dependent Java class or executable file installed on the targeted grid site. Further communication (what kind of communication will be used is specified by the client) between the client and the worker is continued using one of three possible methods:

- Via binder—All communication will go through the binder with the option to have a specific code on the binder for processing the data transferred between the client and the worker.
- Built-in direct communication—The worker connects directly to the client after the client has been matched to the worker and they continue their communication without going through the binder; network performance will probably be better, however not all clients are capable of accepting incoming connections.
- Custom communication—Defined using arbitrary application-specific mechanism; it is up to the client and the worker to implement it, the service only provides exchange of communication descriptors.

2.2. Management of the Job Pool. A pool of ready worker jobs is maintained using sophisticated algorithms with the goal of always having enough worker jobs in the pool for incoming clients as well as minimizing the stress on the grid infrastructure (the pool changes its size according to user demand). Having one pool of jobs for many different applications is an efficient way of conserving precious grid resources. When client connects to Work Binder and asks for a worker job, it will acquire a job almost instantly, providing there is one available. If not, all jobs in the pool are used up, and the client has an option to try again later.

The suitable size of the pool of jobs may vary significantly, depending on the current load and dynamics of user and job arrivals. For example, in order to handle interaction with individual end users, the interactive applications often need many processes for short periods of time. Such pattern of job allocation may be also attractive to some non-interactive applications that repeatedly run short jobs. However, such a behavior is extremely unsuitable for the current grid infrastructure. With Work Binder, a near-optimal usage of grid resources and high availability of ready jobs are archived by implementing an adaptive job submission and pool management policy. The policy shares jobs among several applications, reuses finished jobs, and adapts to the characteristics of grid sites and present workload.

Two modes of mediator operation, called idle and active, determine different job pool sizes:

- Idle mode of operation is used when there are no active clients. In this mode, pool of ready jobs is kept at a predefined minimum.
- Active mode of operation is used when the number of active clients is above zero. The desired pool size in this operation mode is determined using a number of estimation functions.

In order to calculate the desired pool size, it is necessary to take a few things into account. The first one is previously mentioned minimum pool size. The second is the amount of busy jobs in the pool—the more the pool is used, the more it is expected that it will be used in the near future. The third is the arrival frequency of new clients—if a lot of clients come in a short period of time, pool of jobs may be quickly used up.

In order to keep the pool of ready jobs at a desired size two job creation (job refill) strategies are used: regular and full throttle.

2.3. Regular Refill Strategy. Regular strategy is used for each CE individually and works over longer periods of time that are proportional to the response time of each particular CE. Response time in this case means the time in which the submitted job on the CE will become available to Work Binder.

The sum of available (ready) jobs for each CE used by Work Binder is the total number of ready jobs in the binder's pool. The goal of the regular refill strategy is to keep the number of ready jobs on each CE at the appropriate level by analyzing its current status.

The target pool size for each CE depends on the minimal and maximum pool size for the CE agreed between Work Binder administrator and the administrators of the particular CE and its load. The load preference is the measure of how heavily some CE is currently used by the binder.

The amount of jobs that will submitted on some CE is calculated from the target pool size for the CE and the current amount of ready jobs, but it also takes into account already submitted jobs that are expected to become ready (since it takes time for a submitted job to become available in the grid environment) and busy jobs that will finish soon and possibly become reusable (this is one of Work Binder's advantages, not available in the regular grid environment). It is important to note that the calculated number of jobs will not be submitted instantly, but rather gradually over time to compensate for the non-instant responsiveness of the computing elements.

2.4. Full Throttle Refill Strategy. Full throttle (panic) strategy is used to allocate jobs globally on Work Binder. Unlike the regular strategy which works slowly over periods of time, full throttle strategy submits jobs instantly as soon as it detects that the desired total amount of jobs is greater than the actual amount of ready jobs (and the jobs that are expected to become ready in a relatively short time). This can happen if the clients' arrival frequency becomes too high or if there are many busy jobs that saturate some of computing elements.

If panic refill strategy decides that some new jobs need to be submitted, it is necessary to make these jobs available as soon as possible. Work Binder keeps track of each CE's response time, and in order to provide new ready jobs quickly, it picks the group of computing elements with lowest response times and proportionally submits jobs to them. However, Work Binder will not submit and maintain more than the maximum of jobs agreed between the binder administrator and a CE administrator.

If fastest computing elements have reached their maximum pool sizes and there is a further need for panic refill, the amount of remaining refill jobs will be submitted to the rest of computing elements, by submitting as much as possible of new jobs on the fastest CEs first.

2.5. Work Binder Java API. The Work Binder Java API allows application users to easily integrate Work Binder with their existing applications. This integration can be achieved by using the appropriate API wrapper interfaces. Wrapper interfaces exist in all three tiers, and their purpose is to hide the complex internal communication from application users and provide them with simple interfaces and methods, so that they can focus on their own application specific problems and not bother with grid and binder infrastructure related issues. The API is designed in such a way, that the applications that already have some sort of client-server design can be easily adapted to use it (even if they were not gridified previously).

2.6. Client Interface. At the client side, *ClientConnector* is the wrapper interface through which users can connect to the Binder and obtain a Worker Job. Wrapper interface can be initiated with many options, some of them being:

- Binder Address—IP address of the Work Binder instance
- Binder Port—port on which the Work Binder instance is listening for incoming workers instances
- Application ID—unique name identifying which application supported by the binder will the client use
- Required Wall Clock Time—how long will the client use the Worker Job
- Access String—string describing the connection that will be used between the client and the worker
- Client's Certificate—user's grid certificate needed for authentication & authorization (every grid user has its own certificate that can be used to access various grid services)

Once a Worker Job has been obtained, the user can communicate to the Worker Job through *ClientConnector* interface methods or get the reference to the actual socket being used to communicate to the Worker job (this feature is not a preferred way of communication, but is provided in order to make life easier for applications that already have some sort of client-server communication implemented).

2.7. Binder Interface. Work Binder intercepts communication between the client and worker. One of the purposes of this intermediate level of communication, among other reasons, is to provide a way for clients and workers to connect to each other in environments that would not allow them to establish direct communication due to firewall or other network issues (Binder acts as a proxy server in this case). Another possible use is to add some application logic to the binder level where some processing can be performed on the Binder instead of on the remote worker job. The client chooses whether it wants to connect to the worker job via binder or directly (the details of direct connection can be exchanged via binder). If direct communication is chosen, this entire level of communication is skipped.

If communication via binder is used, the users have an option to implement application specific handlers that run at the binder by implementing the provided *BinderHandler* interface. If there is no application-specific implementation on Work Binder, a simple built-in proxy implementation is used where Work Binder acts as a proxy between the client and the Worker Job.

From within *BinderHandler* users are given access to an instance of the *BinderConnector* interface. This instance represents the wrapper interface at the binder. Similarly to the wrapper interface on the client, this class gives access to the communication primitives to the client on one and the worker on the other side, as well as direct access to the actual sockets used to communicate to the client and the worker.

2.8. Worker Interface. Worker level interface provides communication from within a worker job. Application users must implement their custom handlers by implementing the provided *WorkerHandler* interface. Users then get access to an instance of the *WorkerConnector* interface, which is also a wrapper interface. With this interface, as with the other two wrappers, users get access to communication primitives as well as low level access to the actual socket used for communication.

Application-specific *WorkerHandler* implementations are executed on remote grid sites, so they must be preinstalled and available to Work Binder in order to be executed successfully. Binder will detect the installed applications on each supporting site and keep track of them allowing application users to transparently add new applications to the Binder.

If an application user just wants to execute some external program located on the WN, instead of writing her own *WorkerHandler* implementation, there is an option to use one of built-in implementations. This way, an easy integration of programs written in various programming languages is possible, by providing execution

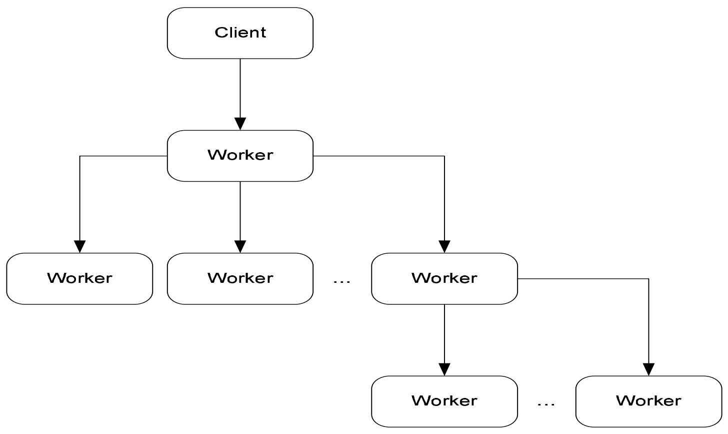

FIG. 2.2. *An example of dynamic job graph*

of application-defined programs or scripts, as well as assistance in establishing communication with the client. Two built-in implementations for two distinct use cases scenarios are available:

- *ExternalExecutor* is a *WorkerHandler* implementation that is intended for applications that want to establish direct communication between the worker and the client, where the worker is an external program located on the remote worker node. This implementation simply closes the connection to the binder and towards client, and executes a command line string on the worker with arguments provided by the client. It is up to the external program and the client to reestablish communication, if needed. For example, the client may inform the worker (using the program invocation parameters) about the address the worker should connect to.
- *ExternalListener* is the other *WorkerHandler* implementation intended for applications that want to use communication via binder, but the worker is an external program located on the remote WN. This implementation preserves the connection to the binder and towards client, starts listening on a local socket, and executes the external program to which it provides an address and the port on which it is listening as addition to command line arguments. The worker program is expected to connect to this socket (instead of connecting to the client) and communicate with the client normally using its own application-defined protocol as if there are no mediators between the worker and client. The client therefore does not notice any interruption of the communication and can start using the application-defined protocol after successful execution of *ClientConnector.connect()* method. Another benefit of this approach is that all communication will go through the binder. This scenario is useful in cases when the worker and the client cannot establish direct communication or when there is a need to have a specific code on the binder itself through which all the data transferred between the client and the worker will go.

An interesting feature is the ability of users to allocate several worker jobs by using the ClientConnector, not only from client machines, but also from workers. This way complex dynamic graph of jobs can be implemented, as shown in Figure 2.2. It is up to the application to orchestrate obtained workers that run in parallel.

3. Conclusion.

The described approach will allow applications to use the grid in a new way, with a minimal additional load on the infrastructure. Users will have improved experience by having immediate availability to new jobs and the ability to uniformly access jobs from several grid sites using a simple programming model.

Three applications are currently being adapted to use this service, which will help its further evaluation and improvement. Also, a simplified interface for communication between the processes that use the described architecture is being implemented.

This research was supported by SEE-GRID-SCI project funded by the European Commission under the FP7 Research Infrastructures contract no. 211338.

REFERENCES

[1] *Pilot Jobs*, HEP Sysadmin Wiki, `http://www.sysadmin.hep.ac.uk/wiki/Pilot_Jobs`

[2] S. Burke, S. Campana, P.M. Lorenzo, C. Nater, R. Santinelli, A. Sciabí, *gLite 3.1 User Guide*, Tech. Rep. CERN-LCG-GDEIS-722398, Worldwide LHC Computing Grid, April 2008, `https://edms.cern.ch/document/722398/`
[3] *Work Binder Application Service*, EGEE SEE ROC and SEE-GRID Wiki `http://wiki.egee-see.org/Work_Binder_Application_Service`
[4] J. T. Moscicki, *DIANE—Distributed Analysis Environment for GRID-enabled Simulation and Analysis of Physics Data*, Nuclear Science Symposium Conference Record, 2003 IEEE Volume 3, 19–25 Oct. 2003, pp.1617–1620 Vol. 3.
[5] *DIANE: Distributed Analysis Environment*, `http://it-proj-diane.web.cern.ch/it-proj-diane/`

Edited by: Dana Petcu
Received: May 22nd, 2009
Accepted: Nov 3rd, 2009