Load-balancing EU-DataGrid Resource Brokers

William Lee Steve McGough Steven Newhouse John Darlington

London e-Science Centre, Imperial College London, South Kensington Campus, London SW7 2AZ, UK
Email: lesc-staff@doc.ic.ac.uk

Abstract

The European DataGrid (EDG)[17] project aims to provide a platform to satisfy the ever-growing demand of high computation and storage requirements across scientific disciplines. Its resource broker is the gateway to a managed set of compute elements for handling the job submission and accounting. In this paper, we present an infrastructure layering on top of the resource brokers for load-balancing job submission requests. The resource broker is abstracted as OGSI compliant services to provide interoperability to heterogeneous submission clients. We demonstrate the use of Javaspace as an information distribution framework for different strategic elements to co-operatively load-balance job submission requests.

1 Introduction

The European Data Grid (EDG)[17] is a platform to support intensive computational analysis of extremely large-scale distributed datasets across widely distributed scientific communities.

The Resource Broker (RB) in the EDG Work Package 1[1] is the gateway for submitting job to a managed set of compute elements. End-users wishing to submit jobs to the EDG platform have access to a collection of command-line and graphical tools. Application Programming Interface (API) for C++, Python or Java are available for customisable clients. By hiding the network protocol details under the well-defined set of programmable interfaces, it allows progressive refinement to the underlying protocols without requiring clients to be rewritten to deal with changes.

The Resource Broker is based on the traditional client/server model[16]. The RB daemon listens to a well-known port and uses the Grid Security Infrastructure (GSI) [7] to ensure client authenticity. The daemon passes incoming job request to a set of multi-threaded agents to communicate with the EDG subsystems to submit and manage jobs. Therefore, the stability and scalability of the resource broker is essential for the adoption of the infrastructure.

We recognise the need to allow heterogeneous clients to interact with the resource broker in any imaginative way. Clients can range from end-user submission interface, or autonomous agents managing large batch jobs on behalf of users. In order to free the clients from the programming language dependent API, we leverage the Open Grid Services Infrastructure (OGSI)[18] to provide a web service environment to support client interaction.

The objectives of the load-balancing infrastructure presented in this paper are to pool and encapsulate a set of resource brokers as co-operative OGSI services.

- Flexibily couple a dynamic set of Resource Brokers to form a submission pool
- Provide a well-defined OGSI-compliant service interface for submitting and managing EDG jobs.
- Job management service is distributed across co-operative OGSI service containers to reduce load and prevent single-point of failure.

2 Background

2.1 The European Data Grid Resource Broker

The Resource Broker is a middleware responsible for carrying out a set of tasks related to user job submission. These tasks include interacting with Replica Catalog (RC) to resolve logical data set names, finding preliminary set of sites for data transfer, job submission and management by interacting with EDG sub-systems such as the Job Submission Service (JSS).

The resource broker is a daemon process that listens to a TCP/IP socket for client requests. Upon

client interaction, a thread is spawned to handle the client messages using a new available port. The RB master daemon acts as the preliminary broker for agent thread to handle individual client. The agent process acts as the hub for carrying out tasks by communicating with the JSS, RC and the local Job Registry (JR) database through the DBMS interface to ensure job state consistency and persistence of queued jobs.

The design suffers from problems of the traditional client-server model.

- Clients address the RB they are wishing to use by the host and port. Reliability and availability of the resource broker is directly presented to potential users.
- The Resource Broker is a single-point of failure and well-known RB might become biased with high loads.
- Under high load, available ports become scarce and the master daemon becomes the bottleneck of the resource broker. Scalability is limited by the machine architecture and operating system.

2.2 Open Grid Services Infrastructure

The Open Grid Services Infrastructure (OGSI)[18] has brought about a convergence of the grid and web services communities. It leverages commercially supported web services protocols to build a Grid infrastructure. OGSI adopts the general web service approach for describing the abstract interface and the implementation details of all Grid services by using WSDL. The network binding and the messaging layer are interchangeable. This flexibility allows existing protocols and future standards to be described through a unified interface language.

The core contribution of the OGSI specification is the standardisation of a set of core service types that are essential for distributed computing. The service port types include *Factory* for creating new service, *Notification* related port types for managing subscription and receiving notification, etc.. OGSI introduces the notion of a Grid Service Handle (GSH), which acts as a globally unique pointer for locating service through a handle resolver. A GSH is resolved by a *HandleResolver* service into a Grid Service Reference (GSR). The GSR acts as the binding-specific network pointer to the service. In the case of a GSH resolvable to a GSR expressed as WSDL. The WSDL description would contain the networking location, protocol and the messaging characteristics of the service, which the client can access. By differentiating the service identity from the network details, it decouples the client from the locality of the service as well as providing an opportunity for the service to handle migration or fault recovery.

3 Load-balancing Resource Broker

The resource broker can be considered as a load-balancer for compute elements. It decides on a preliminary set of compute elements where the jobs can be launched based on the user requirements, such as priority, architecture and the current loads. In order to avoid loading a particular instance of resource broker, our approach is to layer on top a collection of resource brokers with a meta load-balancer. The meta load-balancer acts as the gateway for parallelising requests to resource brokers acting as backends.

Several options have been considered;

- By emulating the resource broker master daemon, one can delegate requests to backend resource brokers using a simple round-robin strategy. The agent thread is spawned on the delegated resource broker, and the URL for the client to contact the agent thread is returned to the user through the load-balancing daemon. Later conversation with the agent thread is performed directly with the RB host. This scheme allows a dedicated machine to be used to delegate request to backend resource brokers, reducing the load of the master daemon load on each RB. However, it still presents a single point of failure. Since the network protocol is well encapsulated by the API, emulating the resource broker daemon demands reverse engineering the protocol, which is error-prone and suffered from later change in the protocol.
- Hardware load-balancers have been widely used in client-server environment[6] . Instead of exposing backend resources through delegation, hardware load-balancer hides the backend resources behind a single network identity. It provides an efficient mean for forwarding network packets to resources based on their operational metrics and loads.

The load-balancer will keep track of session information based on some rules, such as client IP address, protocol content, etc. to ensure later conversational packets will be routed to the same backend resource to ensure consistency. However, this approach is mainly used in a cluster environment with high-speed network within one organisation. Multiple resource brokers in a geographically dispersed virtual organisation might not be suitable for this use. Also the lack of protocol transparency presents difficulty in establishing session rules based on the packet content.

4 Load-balancing Architecture

The Load-balancing Architecture depicted in figure 1 is constructed from a clear separation between information distribution and decision-making. We believe load-balancing strategies are pluggable entities that are interchangeable based on the usage pattern as well as organisational policies.

The information distribution channel is termed *Information Tuplespace*, which serves as the core of the infrastructure. Client interaction takes place in the OGSI and web-services layer. The OGSI services provides an entry point to the system. They introduce job submission request information into the Tuplespace in the *Pull* strategy, or optionally *Push* a request to the *Resource Broker Agent*. Each agent represents a Resource Broker instance. It strategically pulls request from the Tuplespace and instruct the Resource Broker to submit the job through the RB Client API. The agent is also responsible to introduce response tuple into the Tuplespace to acknowledge a submitted job. This provides a natural load-balancing scheme for any participating OGSI containers to provide the job management function.

4.1 Characteristics

The load-balancing architecture presented here exhibits the following characteristics

- *Interoperability* - In order to increase adoption of the EDG platform, functionalities provided by the resource broker is encapsulated as OGSI-compliant services. Job submission and management capabilities are abstracted as two port types;

1. *JobSubmissionFactory* is an extension of the OGSI *Factory* port type that is responsible for instantiating new instance of JobManager service for a given job submission request. The JobSubmissionFactory will use the Resource Broker client API to contact a Resource Broker selected by the load-balancing infrastructure. JobSubmissionFactory service instances are expected to be advertised and discovered by clients through the multitudes of web service discovery mechanisms, such as UDDI directory[14], GT3 Index Service[4], etc..

2. *JobManager* is a service accessible only by the user who instantiate this service through the *JobSubmissionFactory*. Each instance of JobManager represents a single job submission. It encapsulates the management API for querying status and cancelling the job. The state of the JobManager service is held by the Resource Broker who is managing the job, therefore the JobManager instance is free to migrate from container to container as long as the resolver service would resolve the GSH of the JobManager instance to a service running among a set of co-operative containers. The JobManager instance is a transient service. It's lifetime is determined by the status of the job it is representing as well as the client's desire to terminate the job. When the job has reached the *JOB-OUTPUT-TRANSFERED* state, the JobManager instance will terminate after a configurable period to reclaim used resources. The JobManager port type extends the OGSI *NotificationSource* port type. Clients can register *NotificationSink* instances to the JobManager to receive important notification, such as the change of job state service data.

Our implementation is based on the Globus Toolkit 3.0 (GT3)[3] core distribution. This is a Java based reference implementation for the OGSI specification. We have enhanced the GT3 service container by allowing instance created by the Factory port

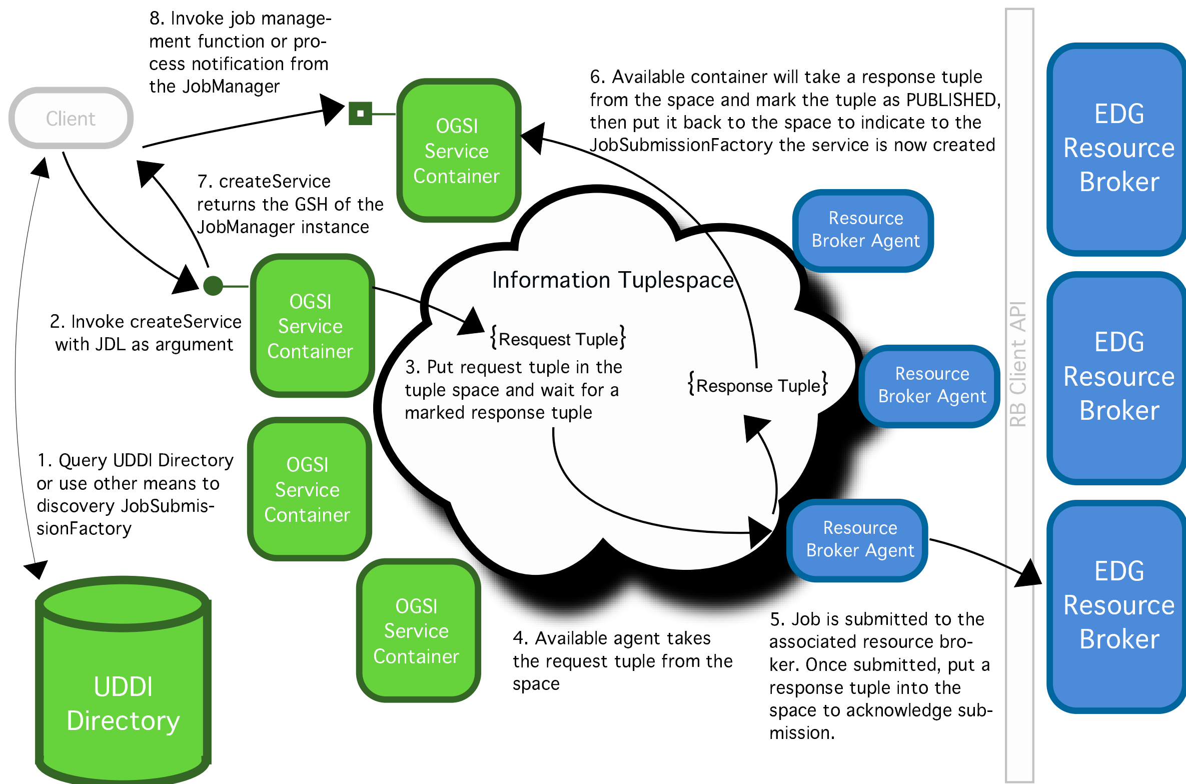

Figure 1: The Load-balancing Infrastructure

types to be instantiated on a foreign containers through our information distribution framework.

- *Transparency* - The OGSI service encapsulation hides the Resource Broker. Clients are not aware of the Resource Broker the job is submitted through. The port type definitions are arguably generic to any job submission system. At the time of writing, the job description language supported is the European Data Grid JDL[13]. However, through the use of extensiblity element and service data description, a JobSubmissionFactory instance can publish the supported set of job description languages, such as RSL[2], Condor ClassAds[15] or SGE execution scripts[12]. By publishing these meta-data about the capabilities of JobSubmissionFactory, clients can search and adapt these services in a wider communities that satisfy their needs.

- *Decentralised* - JobSubmissionFactory instances taking part in the same load-balancing pool are considered equal. Clients can submit job through any of the JobSubmissionFactory services hosted across organisations that span geographical boundaries. The *createService* request on the factory will result in a JobManager instance being created on any of the co-operating service containers that later serves the long-running management function. The instantiation is govern by the load-balancing strategy employed by the infrastructure.

- *Separating Information and Decision* - Efficient load-balancing demands intelligence on the state of the resources. Load-balancing strategies utilise these information to make informed decision on resource assignment. This paper concentrates on describing a framework for information to be distributed and strategy modules to be plugged in at different part of the system to support a wide variety of load-balancing strategies. We will demonstrate the framework with a *Push* and *Pull* hybrid approach.

4.2 Information Tuplespace

The Information Tuplespace resembles the Linda[8] system that consists of a small num-

ber of operations (e.g. *take*, *read*, *write*, *notify*) for manipulating tuple persistently stored in the space. We have chosen to use the reference implementation of Javaspace[10] provided by the Java JINI[11] technology. The Javaspace technology has posed many advantages;

- Javaspace is an API for a Java LINDA system. Multiple implementations are available with different operational characteristics, such as persistence method, security and performance. By developing against the API, interchanging implementation for scalability purposes would only result in minimal change in deployment.
- Javaspace operations are transaction aware. By using the Java Transaction API (JTA)[9], the system can ensure ACID property for job submission.
- The Javaspace implementation persists tuple states into long-term storage. Failure in the Javaspace node can be recovered by replaying the persistence log. Javaspace clients refer to the tuplespace by name, therefore transparent to network migration.
- Most importantly, tuple published to a Javaspace not only carries information presented as states of a Java object. The Java object can also carry executable content inherited from the JINI technology. We have used this characteristic to implement the Job Courier Tuple that routes job request to other spaces.

4.2.1 Job Request Tuple

A Job Request Tuple *Req* consists of

$$Req = \{GUID, GSH, UID, JDL\} \tag{1}$$

- *GUID* - A globally unique identifier generated per job request to identify a job submission through the system
- *GSH* - The Grid Service Handle of the JobSubmissionFactory that handles the user request
- *UID* - The credential of the user who submits the job. This is a *org.ietf.jgss.GSSCredential* object representing the GSI X.509 certificate of the user.
- *JDL* - The job description document specifying the user intent.

This tuple is generated by the JobSubmissionFactory when a *createService* request is received. The factory introduce this tuple to the space and wait for a Job Response Tuple *Resp* that satisfies the template tuple with the PUB field marked as true and matching GUID and UID.

$$Resp = \{GUID, *, UID, true, *\} \tag{2}$$

The JobSubmissionFactory will returns the JobManager locator denoted by the MGSH field of the tuple to the client.

4.2.2 Job Response Tuple

A Job Response Tuple *Resp* denotes a successful submission through a Resource Broker Agent. The Agent who has taken a Job Request Tuple {GUID, GSH, UID, JDL} from the Tuplespace will insert a tuple {GUID, null, UID, false, JOBID} to the space to indicate the job is begin processed by the EDG sub-system.

$$Resp = \{GUID, MGSH, UID, PUB, JOBID\} \tag{3}$$

- *GUID* - A globally unique identifier matching the GUID field of a previously published Job Request Tuple *R*
- *MGSH* - The Grid Service Handle of the JobManager that handles the management request. A null value indicates it has not been managed by any JobManager service.
- *UID* - The credential of the user matching the UID field of request tuple *R*
- *PUB* - A boolean field indicating whether a JobManager service is bound to this job.
- *JOBID* - The Job ID returned by the EDG Resource Broker for establishing management conversation.

Once the Job Response Tuple is introduced into the space. An available OGSI container will take the tuple off the space and instantiate a JobManager service instance in the local container. The JobManager will use the JOBID field to converse with the RB Client management API. Upon creation of the service, the tuple is written back to the tuplespace with the MGSH field set to the GSH of the JobManager instance and PUB set to true. The JobSubmissionFactory that initiates the request will now be unblocked with the locator available to be returned to the client.

4.2.3 Job Courier Tuple

The Job Request and Response Tuples provide the ingredients for a *Pull* load-balancing strategy. An agent *pulls* an available request tuple from the space when it is free to do so. However, this creates an environment where tuples will be left in the space for their lease to be expired and reclaimed when all the participating agents are busy. Although it provides *natural* load-balancing amongst all the EDG Resource Brokers, since the *take* operation does not guarantee any ordering, a *Push* strategy is needed to create multiple priority queue.

A Job Courier Tuple is an executable entity that represents a slot an agent offers for immediate execution. A JobSubmissionFactory takes a courier tuple from the space when a job needs to be pushed to an agent as quickly as possible. By invoking the *submit* method on the courier tuple, the tuple will invoke the resource broker agent by its internal means to execute the job. The JobSubmissionFactory can decide from the return value whether it will creates the JobManager service locally or delegate it back to the tuplespace as discussed in the *pull* strategy. Once a JobSubmissionFactory has finished using the courier, it can choose to release the courier tuple back into the courier tuple for others to use or retain it in a greedy fashion.

4.3 Strategic Elements

In the last section, we have demonstrated the components that enable the *Push*, *Pull* and hybrid strategies. The infrastructure provides three distinct points for inserting different strategies.

- *JobSubmissionFactory* - The JobSubmissionFactory is responsible for either inserting a request into the tuplespace for an agent to be retrieved, or it can look for courier tuples for executing high-priority jobs. A well-known JobSubmissionFactory can aggressively retain a high numbers of courier tuples in advance. By reserving these available slots, the factory sustaining a high rate of submission can push jobs to the agents as quickly as possible.

- *Resource Broker Agent* - The Resource Broker Agent implementation takes request tuples from the space and introduce response tuple when a job is passed to the EDG sub-system. The rate of retrieval is a variable in the agent strategy. This is typically controlled by the failure submission rate to the underlying resource broker. Under high load, we observe connection refusal by the resource broker master daemon. By throttling the retrieval rate, the agent attempts to minimize and adapt to the failure rate accordingly.

- *JobManager* - The instantiation of JobManager is controlled by a manager thread running inside each participating OGSI containers. The manager will takes response tuple from the space and create JobManager in the local container. The decision for creating new instances is govern by a configurable maximum count, as well as the sum of invocation activity to all the local JobManager instances.

5 Discussion and Future Works

The load-balancing framework presented can thus be considered as generic job submission and management services. The port types are designed to have no reference to the European Data Grid platform. OGSI Service Data allows us to advertise information related to the underlying job submission system the service supports. Job description is passed to the JobSubmissionFactory through the *creationParameters* extensibility element. The service data description advertises the support of the EDG JDL.

The JobManager contains operations for querying job status. However, different underlying submission systems have varying model of job states. In order for generic tools to understand state changes to take appropriate actions. One possible solution is to use service data description to advertise the possible state transition for a given model as a directed graph. Graph state described ontologically can be inferred to establish understanding between different models.

The load-balancing infrastructure can scale to use any numbers of JobSubmissionService instances running on dedicated machines. Hardware load-balancers can be used to cluster multiple containers. Dedicated containers can be configured to serve as hosting environments for JobManager instances. They sustain higher activity than JobSubmissionService. Although the multi-layered approach will increase latency for a job request-response compared to the direct use of the Resource Broker Client API, clients are opened up to a dynamic collection of resource brokers that can handle their requests. It potentially reduce failure rate considerably because of the transparent retry

semantics. The potential bottleneck of the system is the scalability of the information tuplespace. Multiple open-source as well as commercial implementations of Javaspace are available with different performance characteristics. Further performance testing is essential for determining different combination of strategy as well as implementation in different load pattern.

6 Conclusion

This paper presents a secure framework that abstracts the European Data Grid Resource Broker as a set of OGSI job submission and management services. The generic abstraction allows the infrastructure to be applied to a variety of job submission system. Moreover, the OGSI based service oriented architecture opens a wide integration avenue to heterogeneous clients. Performance profiling of resource brokers under high submission rate have shown a degradation of throughput and build-up of long submission queues[5]. By hiding the monolithic Resource Broker as a component of a distributed job submission system, load can be shared between resource broker instances. This paper has demonstrated the design and implementation of the information tuplespace and the strategic elements that makes use of these information to distributed job requests. A LINDA inspired tuplespace based on the Javaspace API is an ideal candidate for publishing job requests to a bulletin-board where agents respond and execute the request based on its strategy. The three tuple types are the ingredients for a push-pull hybrid approach for load-balancing.

Acknowledgement - The work has been carried out as part of the EPSRC project on Effective Multi-User and Multi-Job Resource Utilisation (GR/R74505/01)

References

[1] The European Data Grid work package 1. http://server11.infn.it/workload-grid/.

[2] The Globus Resource Specification Language RSLÊv1.0. http://www.globus.org/gram/rsl_spec1.html.

[3] The Globus Toolkit 3.0. http://www-unix.globus.org/toolkit/download.html.

[4] GT3 Index Service overview. http://www.globus.org/ogsa/releases/final/docs/infosvcs/indexsvc_overvi%ew.html.

[5] G. Avellino et al. The first deployment of workload management services on the eu datagrid testbed: feedback on design and implementation. In *Computing in High Energy and Nuclear Physics (CHEP03)*, March 2003.

[6] F5. Big-IP hardware load-balancer. http://www.f5.com/f5products/bigip/.

[7] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998.

[8] D. Gelernter. Generative communication in linda. In *ACM Transactions on Programming Languages and Systems, No.1, pp. 80-112*, January 1985.

[9] Sun Microsystems. Java transaction api specification v.1.0.1. http://java.sun.com/products/jta/index.html.

[10] Sun Microsystems. Javaspace service specification. http://java.sun.com/products/jini/2.0/doc/specs/html/js-spec.html.

[11] Sun Microsystems. Jini(tm) Network Technology. http://java.sun.com/jini/.

[12] Sun Microsystems. Sun one grid engine software. http://wwws.sun.com/software/gridware/.

[13] F. Pacini. Job Description Language How-to. http://server11.infn.it/workload-grid/docs/DataGrid-01-TEN-0102-0_2-Doc%ument.pdf.

[14] UDDI Project. Universal Description, Discovery and Integration (UDDI), September 2002. Available at http://www.uddi.org.

[15] M. Solomon R. Raman, M. Livny. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, Chicago, IL, USA, July 1998.

[16] S. Monforte S. Cavalieri. Resource Broker Architecture and API. http://www.infn.it/workload-grid/docs/20010613-RBArch-2.doc.

[17] The DataGrid Project. http://www.eu-datagrid.org/.

[18] Foster I. Frey J. Graham S. Kesselman C. Snelling D. Vanderbilt P. Tuecke S., Czajkowski K. Open Grid Service Infrastructure (OGSI) v.1.0 Specification, Feburary 2003.